UNIVERSIDADE DO VALE DO RIO DOS SINOS
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
PROGRAMA INTERDISCIPLINAR DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA

Programa Interdisciplinar de Pós-Graduação em

Computação Aplicada

Mestrado Acadêmico

Giovane Oliveira de Barcelos

inSOA - Uma Linguagem de Composição de Serviços para Dispositivos Móveis

UNISINOS

inSOA - Uma Linguagem de Composição de Serviços para Dispositivos Móveis

por

Giovane Oliveira de Barcelos

Dissertação apresentada à Universidade do Vale do Rio dos Sinos como requisito parcial para a obtenção do título de Mestre em Computação Aplicada.

Orientador: Prof. Dr. Sérgio Crespo Coelho da Silva Pinto

São Leopoldo

2009

B242i Barcelos, Giovane Oliveira de
inSOA: uma linguagem de composição de serviços para dispositivos móveis / por Giovane Oliveira de Barcelos. -- 2009.
128 p. : il. ; 30cm.

Dissertação (mestrado) -- Universidade do Vale do Rio dos Sinos, Programa de Pós-Graduação em Computação Aplicada, São Leopoldo, RS, 2009.
"Orientação: Prof. Dr. Sérgio Crespo Crespo da Silva Pinto, Ciências Exatas e Tecnológicas".

1. Computação Móvel - Linguagem de Composição de Serviços. 2. SOA - Composição de Serviços. 3. Linguagem de Domínio Específico. 4. DSL. I. Título.

CDU 004.75.057.5:004.43

Catalogação na Publicação:
Bibliotecária Eliete Mari Doncato Brasil - CRB 10/1184

Dedico este trabalho às pessoas mais importantes da minha vida: minha esposa Jane, meus pais Flávio e Antônia e, meus companheiros Lelo e Mitcha.

AGRADECIMENTOS

A DEUS pela vida, saúde, sabedoria e por estar sempre ao meu lado.

A minha esposa Jane, por seu amor, paciência e apoio em todos os momentos.

Aos meus pais, Flávio e Antônia, pela educação e incentivo.

Aos meus irmãos Vinicius, Fabiane e Alexandre, pelo apoio.

Ao professor Sérgio Crespo: pela orientação, confiança e incentivo; por conduzir meu desenvolvimento com muita sabedoria e paciência; pela oportunidade, pela bolsa e por possibilitar o meu retorno ao mestrado.

Aos professores Marco Gerosa e João Gluz pela participação na banca e pelas valiosas contribuições para melhoria do trabalho.

Aos meus colegas do projeto *U-SOA*, Luciano Zanuz e Alexsandro Filippetto, pelo companheirismo e parceria no desenvolvimento dos trabalhos.

Aos alunos do curso de Engenharia da Computação, Cládio, Douglas e João, que compraram a idéia e nos auxiliaram no projeto *U-SOA*.

Aos professores e funcionários do *PIPCA* que sempre estiveram disponíveis para ajudar e ensinar.

Ao meu amigo Alexandre Weissmuller, pelos conselhos e sempre valiosas informações.

A *CAPES* pelo apoio financeiro.

A todos que direta ou indiretamente contribuíram para a realização deste trabalho.

"Mas nada há encoberto que não haja
de ser descoberto; nem oculto, que não haja
de ser sabido.
Porque, onde estiver o vosso tesouro,
ali estará também o vosso coração."
Lucas, 12:2 e 12:34

RESUMO

Composição de serviços em dispositivos móveis está se tornando indispensável para as atuais e futuras necessidades dos usuários de *TI*. Estas necessidades são suportadas pela convergência de arquiteturas centralizadas para arquiteturas distribuídas que estimulam a composição de serviços *web*. Estes serviços *web*, motivados pela arquitetura *Web Services*, logo evoluíram para *SOA*. Esta arquitetura por sua vez, no contexto de orquestração de serviços, tem exigido linguagens de composição para sua efetiva implementação. Linguagens estas, que têm progredido para linguagens interpretadas e declarativas.

Neste contexto de evolução, este trabalho apresenta, uma linguagem declarativa de composição de serviços chamada *inSOA*. Esta linguagem foi desenvolvida para ser executada em dispositivos móveis como parte de um projeto de uma arquitetura ubíqua colaborativa chamada de *U-SOA*. *inSOA* é uma solução focada em *Web Services* com vocação *SOA* e aderente aos *patterns* de composição de serviços, além de poder ser embutida em linguagens de propósito geral e leve no melhor estilo das linguagens *DSL*.

Como forma de testar e avaliar a linguagem, foi realizado um estudo de caso com dois cenários, onde se procurou desenvolver composições com o maior número de *patterns* possível. Estas composições foram executadas em um ciclo de desenvolvimento completo envolvendo outras ferramentas da arquitetura *U-SOA*.

Palavras-chaves: *SOA*, Composição de Serviços, Linguagem de Composição de Serviços, Computação Móvel, Linguagem de Domínio Específico, *DSL*.

ABSTRACT

Services composition on mobile devices is becoming essential for current and future needs of IT users. These needs are supported by the convergence from centralized to distributed architectures that encourage the web services composition. These web services, motivated by the Web Services architecture, soon evolved into an SOA. This architecture in turn, in the services orchestration, has required composition language for their effective implementation. These languages, has advanced to interpreted and declarative languages.

In this evolution context, this work presents, a declarative language of services composition services called inSOA. This language was developed to execute on mobile devices as part of a project of a collaborative ubiquitous architecture called U-SOA. InSOA is a solution focused on Web Services and SOA adherent to the patterns of services composition, and can be embedded in a language of general purpose and light in the better DSL languages style.

As a way to test and evaluate the language, a case study was developed with two scenarios, which sought to develop compositions with the largest number of possible patterns. These compositions were performed in a full development cycle involving other tools of the U-SOA architecture.

Key-words: *SOA, Services Composition, Language of Services Composition, Mobile Devices, Domain-Specific Language, DSL.*

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

3G	Third Generation
ANTLR	ANother Tool for Language Recognition
API	Application Programming Interface
BNF	Backus–Naur Form
BPEL	Business Process Execution Language
BPEL4WS	Business Process Execution Language for Web Services
BPM	Business Processes Management
BPML	Business Process Modeling Language
BSD	Berkeley Software Distribution
CBPEL	Common Business Process Execution Language
CDATA	Character Data
CDC	Connected Device Configuration
CEP	Código de Endereçamento Postal
CERN	European Organization for Nuclear Research
CIL	Common Intermediate Language
CLAM	Composition Language for Autonomous Megamodules
CLDC	Connected Limited Device Configuration
CLR	Common Language Runtime
CORBA	Common Object Request Broker Architecture
CPF	Cadastro de Pessoa Física
CSS	Cascading Style Sheets
D-U-N-S	Data Universal Numbering System
DCOM	Distributed Component Object Model
DSL	Domain-Specific Languages
DVM	Dalvik Virtual Machine
EAN	European Article Number
EBNF	Extended Backus–Naur Form
FCL	Framework Class Library
GML	Geography Markup Language
GMT	Greenwich Mean Time
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
InSOA	Invoke Service Oriented Architecture
ISO	International Organization for Standardization
IT	Information Technology
JME	Java Micro Edition
JSR	Java Specification Requests
JVM	Java Virtual Machine
MIDP	Mobile Information Device Profile
MVC	Model-view-controller
OWL-S	Ontology Web Language for Services
QoS	Quality of Service
REST	Representational State Transfer
RFC	Request for Comments
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SGML	Standard Generalized Markup Language

SHA	Secure Hash Algorithm
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SQLSOA	Structured reQuest Language for Service Oriented Architecture
TI	Tecnologia da Informação
U-SOA	Ubiquitous Service-Oriented Architecture
UDDI	Universal Description, Discovery and Integration
UML	Unified Modeling Language
URI	Uniform Resource Identifier
WS	Web Services
WSCI	Web Service Choreography Interface
WSDL	Web Services Description Language
WSFL	Web Services Flow Language
WWW	World Wide Web
XML	Extensible Markup Language
XPATH	XML Path Language
XPDL	XML Process Definition Language
XSD	XML Schema Definition
XSL	Extensible Stylesheet Language
XSLT	XSL Transformations

SUMÁRIO

1 INTRODUÇÃO

A *Web* foi criada em 1989 nos laboratórios do *CERN*[1] (Conselho Europeu para Pesquisa Nuclear) com o objetivo de facilitar a colaboração dos pesquisadores de seus laboratórios no desenvolvimento de pesquisas e projetos [Berners-Lee 1989]. A implementação inicial deu-se pelo compartilhamento de documentos de pesquisa e desenvolvimento de software em hipertexto utilizando um navegador numa arquitetura em rede. Desde então o uso da *web* tem crescido muito. Este crescimento foi facilitado pelo aumento da velocidade das redes de comunicação, novas tecnologias e formas de uso da plataforma *web*, bem como, a demanda de troca de informações motivada pela globalização mundial. Entre estas novas tecnologias podemos destacar *Web Services* (*WS*), que permitiu o uso e reuso de aplicações por meio de serviços disponíveis na *web*.

A arquitetura *WS* rapidamente evoluiu para *SOA* (*Service Oriented Architecture*) que permitiu construir aplicações mais dinâmicas com composição de aplicações em tempo de execução [Nakamura 2004]. A composição e orquestração de serviços tornaram os benefícios da arquitetura *SOA* [IBM 2007; Microsoft 2006] mais visíveis, pois possibilitaram o reuso, colaboração, interoperação e produtividade no desenvolvimento de software. Estes conceitos beneficiaram a antiga tecnologia *workflow,* também chamada de *BPM* (*Business Processes Management*) [Havey 2005]. *BPM* associado com *SOA* estimulou novas tecnologias de composição e orquestração de serviços. Estas novas tecnologias foram concretizadas com o desenvolvimento de linguagens de composição. Dentre estas linguagens de composição a mais influente é a *BPEL* (*Business Process Execution Language*) [Nakamura 2004; Hackmann 2006], que implementa composição e orquestração de serviços a partir de um modelo baseado em *XML*.

Recentemente as tecnologias relacionadas à *web*: *WS* [Srirama 2006], *XML* [Balani 2003A], *SOA* [Balani 2003B] e *BPEL* [Hackmann 2006; Kalasapur 2005], foram migradas para dispositivos móveis, iniciando um processo de convergência tecnológica do *desktop* para os dispositivos portáteis. Isto se deve a demanda crescente de aplicações com mobilidade, que possibilitem a comunicação e integração com aplicações legadas, e também ao apelo da computação ubíqua deste ambiente. O grande desafio deste ambiente computacional é o

1 *CERN - http://public.web.cern.ch/public/*

desenvolvimento de soluções que sejam do tamanho das limitações de recursos impostas por estes dispositivos. Desenvolver linguagens de composição para dispositivos móveis, apesar de desafiante, é necessário, considerando que estes dispositivos dependem muito de integração e uso de serviços disponíveis externamente.

Tabela 1: Cenário de convergência e evolução tecnológica.

Cenário de Convergência e Evolução		
Arquiteturas centralizadas	→	**Arquiteturas distribuídas**
***Web Services* (*WS*)**	→	***SOA* (*Service Oriented Architecture*)**
Business Process Management* + *SOA	→	**Linguagens de composição de serviços**
Computador pessoal	→	**Dispositivos móveis**
Dispositivos móveis clientes	→	**Dispositivos móveis servidores**
Linguagens compiladas	→	**Linguagens interpretadas, declarativas , embutidas e leve**

As tecnologias que estão sendo migradas para os dispositivos móveis devem ser repensadas. A mobilidade e a possibilidade de disponibilizar serviços remotos localmente com baixo custo de processamento é o grande apelo da computação móvel. Entretanto, para que isto seja possível é necessário o desenvolvimento de ferramentas adequadas às limitações destes dispositivos. Além disto, estas ferramentas devem ser distribuídas, orientadas a serviços e aptas a compor serviços, ser embutidas e leves. Dentro deste contexto e cenário de convergência relacionado na tabela 1, este trabalho vem propor uma linguagem de composição de serviços para dispositivos móveis chamada de *inSOA* (*invoke SOA*). Esta linguagem visa aproveitar alguns esforços já empreendidos na migração de tecnologias *WS* para dispositivos móveis e prover uma linguagem de composição de serviços declarativa, leve e de fácil edição, que acesse de forma transparente *WS* e facilite a declaração de composições.

1.1 Motivação

O mundo está se tornando *wireless* e os dispositivos móveis são a sua motivação. Segundo [IBM 2008], as vendas de dispositivos móveis no mundo têm crescido na taxa de 20% ao ano, em média, nos últimos 8 anos, com previsão de crescimento de 5,8% ao ano em média até 2012. Conforme este estudo, existem alguns países em que o número de dispositivos móveis é superior ao número de pessoas. Considerando que a maioria destes dispositivos possui recursos de comunicação via *web* e tecnologia *3G* embutida, existe uma previsível explosão de consumo por serviços que exijam banda larga nos próximos anos.

Nos últimos anos os dispositivos móveis têm experimentado profundas mudanças, passando de dispositivos com poucos recursos para verdadeiros computadores pessoais de bolso, capazes de transmitir dados e voz de forma interativa, abrindo muitas oportunidades de desenvolvimento de *software*. Com isto iniciou-se um novo modelo de negócios, que vislumbra a troca de informações, o uso de bases de dados remotas e principalmente o uso de aplicações legadas via *web*. Como exemplos deste novo modelo, temos: comércio eletrônico, serviços de localização e *home-bank*, dentre outros. Entretanto, nem todos os dispositivos móveis podem acessar *WS*, devido à quantidade de memória e processamento que podem ser demandados [Oscar 2005]. Devido a estas limitações, as aplicações devem ser projetadas e desenvolvidas especificamente para realidade dos dispositivos móveis. Esta realidade vai além das limitações físicas, envolvendo também características de mobilidade, intermitência na comunicação e disponibilidade dos serviços.

A demanda por aplicações focadas em dispositivos móveis também é motivada pela crescente necessidade de serviços que são compostos por mais de um serviço. Este agrupamento de serviços em um único serviço, também chamado de composição de serviço, torna-se visível em aplicações como por exemplo, consulta de viagens, processo de compra com pagamento em *home banking,* rastreamento com localização em mapas, entre outros. Estas necessidades exigem que aplicações atuais para dispositivos sejam aptas a compor e agrupar serviços facilmente.

Recentemente uma série de trabalhos [Srirama 2006; Balani 2003A; Hackmann 2006; Kalasapur 2005] estão migrando as tecnologias de serviços (*XML, WS, SOA, BPEL, etc.*) do *desktop* para dispositivos móveis. Estas tecnologias utilizam as mesmas estruturas *XML*, conceitos e modelos do *desktop*, porém eliminam excessos e resolvem problemas típicos de dispositivos móveis, tais como, a intermitência na comunicação gerada pela mobilidade. Entretanto, os esforços empreendidos na otimização das linguagens de composição para dispositivos móveis, como a linguagem *BPEL* [Hackmann 2006; Kalasapur 2005], não resolvem a complexidade e a imprecisão desnecessária existente nas linguagens de composição [Staab 2003]. Esta complexidade exige o uso de ferramentas poderosas para se criar uma simples composição e executá-la, não sendo possível a criação desta manualmente, de forma simples e leve. Segundo [Wohed 2002], esta complexidade deve-se a dois motivos principais: muitas construções sobrepostas e a semântica não é clara.

Assim, criar uma linguagem para dispositivos móveis que permita o uso e a composição de serviços de forma simples e leve atendendo os *patterns* de composição, é o que motiva a criação da linguagem *inSOA (invoke Service* Oriented *Architecture*).

1.2 Escopo do Trabalho

Este trabalho utilizará técnicas de engenharia de *software* e linguagens de programação com a finalidade de realizar composição de serviços em dispositivos móveis. Estas composições são definidas utilizando uma linguagem declarativa específica para esta finalidade. Para o desenvolvimento deste trabalho são necessários estudos sobre *XML*, *XPath*, *WS*, *SOA*, linguagens de composição, *ANTLR e StringTemplate*.

1.3 Problema

Entre os grandes avanços nos dispositivos móveis está a possibilidade destes conectarem-se na *web,* apesar das limitações na velocidade e os tipos de conexões disponíveis. Com isto, surge a oportunidade de desenvolver aplicativos que acessem e componham serviços disponíveis com tecnologia *SOA* na *internet*, tal qual, em um ambiente *desktop*.

Atualmente, existem uma série de trabalhos que visam migrar as tecnologias e ferramentas disponíveis no *desktop* para dispositivos móveis. Estes trabalhos, apesar de abordarem as limitações de memória e pouco processamento dos dispositivos móveis, limitaram-se a migrar as mesmas estruturas *XML*, conceitos e modelos do *desktop* para este ambiente. Contudo, para este ambiente, do ponto de vista de composição de serviços, deveríamos ter novas ferramentas de desenvolvimento, com vocação colaborativa, declarativa e aderente aos *patterns* de orquestração de serviços.

Dentro deste cenário, o presente trabalho busca respostas para as seguintes perguntas:

“Como acessar serviços *WS* e *SOA* em dispositivos móveis?”.

“Como possibilitar composição de serviços em dispositivos móveis?”.

“Como fazer composição de serviços utilizando uma linguagem declarativa?”.

Estas perguntas sintetizam o foco de pesquisa deste trabalho.

1.4 Questão de Pesquisa

A questão central deste trabalho é: como criar composições de serviços em dispositivos móveis utilizando uma linguagem de composição declarativa.

1.5 Objetivos do Trabalho

Este trabalho tem por objetivo desenvolver uma linguagem de composição de serviços para dispositivos móveis que seja declarativa, suporte composição de serviços, faça tratamento de fluxo de dados, seja leve, simples e esteja apta para ser embutida em linguagens de propósito geral. Tal objetivo é alcançado através da adequação de um *framework SOA* para dispositivos móveis e a criação de uma linguagem de composição específica para *SOA* chamada de *inSOA* (*invoke Service Oriented Architecture)*.

Para atingir o objetivo deste projeto serão implementados os seguintes componentes:

a) **Gramática da linguagem*:*** especificação e definição da linguagem utilizando a gramática do *ANTLR* [Parr 2007] que utiliza o padrão *BNF* (*Backus–Naur Form*) [Naur 1960] modificado.
b) **Análise léxica e *parser inSOA***: componente responsável pela análise léxica e parser da linguagem, gerando como resultado uma árvore sintática compilada da linguagem.
c) **Análise semântica e otimizador**: componente responsável pela análise semântica da linguagem e a otimização da execução dos *WS*. Este componente é responsável por assegurar a lógica de dependência entre os *WS* e a correta definição e uso das variáveis de ambiente. Ele também otimiza a execução da composição, ordenando os WS e definindo os paralelismos destes.
d) **Interface otimizada com o motor de composição *SmallSOA* e o analisador de impacto *gImpact***: este componente é responsável por gerar um arquivo *XML* que servirá de protocolo de comunicação com o motor *SmallSOA* e o analisador de impacto de alterações *gImpact* [Zanuz, Barcelos et al. 2008A]. Este protocolo tem por finalidade orientar o motor na execução da composição. Esta orientação inclui entre outras coisas a ordem de execução dos *WS,* a dependência entre eles, os paralelismos, os tratamentos de falhas, entre outros.

O resultado final será uma linguagem de composição de serviços para dispositivos móveis, com três componentes chaves: compilador *inSOA*, otimizador e uma camada de comunicação com o *SmallSOA*. Esta linguagem é parte integrante de um projeto de maior envergadura que trata da construção de uma meta-arquitetura baseada em serviços para ambiente colaborativos.

A solução proposta neste trabalho será avaliada com um estudo de caso. Esse terá como base uma aplicação para dispositivo móvel acessando *WS* públicos.

1.6 Organização da Trabalho

Esta proposta possui 7 capítulos, sendo que no primeiro encontra-se a introdução. Os demais capítulos são descritos a seguir:

- **Capítulo 2:** Tecnologias Relacionadas – descreve as tecnologias utilizadas no desenvolvimento do trabalho. O texto descreve a tecnologia *XML*, *XPath*, *WS*, *SOA*, linguagens de composição, *ANTLR, StringTemplate* e plataformas de desenvolvimento para dispositivos móveis.
- **Capítulo 3:** Trabalhos Relacionados – mostra alguns esforços no desenvolvimento de linguagens de composição de serviços relevantes ao contexto deste trabalho.
- **Capítulo 4:** *inSOA* – descreve a solução desenvolvida neste trabalho: a arquitetura, o compilador, o otimizador e a interface com o motor de composição *SmallSOA* e analisador de impacto *gImpact*.
- **Capítulo 5:** Estudo de Caso – apresenta um estudo de caso com dois cenários explorando as características da linguagem *inSOA*.
- **Capítulo 6:** Conclusão – discute as conclusões do trabalho ressaltando as contribuições, limitações e trabalhos futuros identificados no decorrer do desenvolvimento da *inSOA*.
- **Capítulo 7:** Referências Bibliográficas - lista as fontes de pesquisa utilizadas na realização deste trabalho.

2 TECNOLOGIAS RELACIONADAS

2.1 *eXtensible Markup Language* (*XML*)

Segundo [Dykes 2005], *XML* é uma linguagem de marcação que utiliza etiquetas (*tags)* para rotular, categorizar e organizar informações em um dado formato. As marcações descrevem documentos ou estrutura de dados e organização. Conteúdos, tais como, imagens, texto e dados, são parte do código que as etiquetas de marcação contém, além de conhecimento semântico inserido na estruturação dos dados. Esta capacidade de organização e armazenamento semântico de informações é o que torna a linguagem *XML* tão útil e poderosa. Isto é percebido no uso intenso da *XML* em tecnologias *web*, onde no contexto dos *WS*, a *XML*, não é apenas utilizada como um formato para troca de mensagens, mas também a forma através da qual os serviços são definidos [Scopel 2005].

XML é um subconjunto da *SGML* (*Standard Generalized Markup Language*). *SGML* é um padrão *ISO* (*ISO 8879:1986 SGML*) de metalinguagem, descendente da linguagem *GML* (*Generalized Markup Language*) da *IBM* [Harold 2002]. Ela fornece uma sintaxe abstrata que pode ser convertida em muitas sintaxes concretas diferentes. Diferentemente da *SGML,* a *XML*, possui uma sintaxe simples e de propósito geral, podendo ser facilmente analisada e implementada, por isto, atualmente esta linguagem tornou-se um padrão de fato. Conforme [Coyle 2002], *XML* tem sido largamente utilizada em uma variedade de aplicações, sendo percebida desde a indústria de vocabulários de dados até a indústria de protocolos, como apresentado na figura 1.

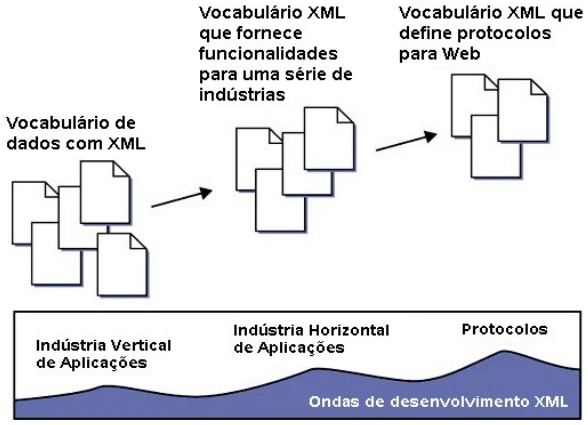

Figura 1: Extensão das aplicações da *XML* [Coyle 2002].

<?xml version="1.0" encoding="UTF-8"?>	Declaração XML
<!--Exemplo XML: Descrição de uma pessoa-->	Comentário
<?xml-stylesheet type="text/css" href="unisinos.css"?>	Prológo
<!DOCTYPE document [<!ENTITY br "Brasil">]>	Entidade
<pessoa id="١٣٠٨">	Elemento com Atributo
<nome>	Elemento
<primeiro>Giovane</primeiro>	Elemento e Texto
<sobrenome>Barcelos</sobrenome>	Elemento e Texto
</nome>	Fecha Elemento
<universidade>UNISINOS</universidade>	Elemento e Texto
<pais>&br;</pais>	Elemento e Ref. Entidade
<![CDATA[١ > ٠ e ٢ < ٣]]>	Seção CDATA
</pessoa>	Fecha Elemento

Figura 2: Seções da linguagem *XML*.

A linguagem *XML* é essencialmente dividida em sete seções [Holzner 2003], como descrito abaixo e ilustrado na figura 2:

- **Prólogo:** aparece no início do documento *XML,* e contém informações a respeito do restante do documento, tais como, a versão e os estilos suportados por este.
- **Declarações *XML*:** é uma seção obrigatória do *XML*, e contém informações sobre a versão do *XML*, os caracteres de codificação utilizados, entre outras definições importantes para o funcionamento do *parser XML*.
- **Instruções de processamento:** contém instruções direcionadas ao *parser XML* que orientam o funcionamento deste, tais como, a forma de tratamento de estilos *CSS* (*Cascading Style Sheets*) no *XML*.
- **Elementos e Atributos:** elementos, também chamados de *tags,* iniciam com o caractere '<' e terminam com '>' e são sensíveis a capitalização. Os atributos são definidos internamente nas *tags* e são compostos de nome e valor. É possível declarar mais de um atributo a uma única *tag*.
- **Comentários:** podem conter qualquer informação, sendo utilizados frequentemente para inserir descrições do documento *XML*. Da mesma forma que *HTML,* deve-se iniciar um comentário com os caracteres '<!--' e finalizar com '-->'.
- **Seções *CDATA*:** utilizadas quando deseja-se inserir dados no *XML* que o *parser* não deve interpretar.
- **Entidades:** representam itens de dados, tal como, uma seção de texto ou um dado binário. Elas podem ser ou não interpretadas pelo *parser XML*. No caso de ser interpretada pelo *parser* é possível declarar entidades por referência. As entidades

por referência permitem criar referências que tornam o código mais fácil de entender e alterar.

A figura 2 mostra um exemplo de um arquivo *XML* na primeira na coluna. Na segunda coluna são descritas cada uma das seções do arquivo *XML*. Como pode-se perceber, existe uma hierarquia entre as *tags* apresentadas no exemplo. A *tag* pessoa contém as *tags* nome, universidade e país dentro dela, enquanto a *tag* nome contém as *tags* primeiro e sobrenome. Estas seções representam a semântica dos dados do *XML*, enquanto o restante das seções são declarações e comandos direcionados ao *parser XML*.

Do ponto de vista dos *WS*, a sintaxe de *XML* especifica como os dados são representados, define como os dados são transmitidos, além dos detalhes de publicação e descoberta de serviços. Costumeiramente as implementações de *web services* decodificam os conteúdos das mensagens *XML*, que contém informações para interagir com as aplicações e seus domínios de *softwares* [Coyle 2002].

2.2 *XML Path Language* (*XPath*)

XPath é uma sintaxe utilizada para selecionar e descrever partes de um documento *XML*. Ela é uma linguagem utilizada para endereçar partes de um documento *XML* [W3C 2003]. Sua sintaxe é compacta e não é baseada na sintaxe *XML,* apesar de ser utilizada para selecionar e navegar em documentos *XML*. Foi desenhada para ser embutida em outras linguagens hóspedes, tais como, *XSLT* e *XQuery*. Sua versão atual é a 2.0, que além das características de seleção e navegação em documentos *XML,* disponíveis na versão anterior, possui instruções de iteração e formatação de saída.

A linguagem *XPath* trata um documento *XML* como uma árvore hierárquica constituída de nodos [Dykes 2005]. A árvore prevê todas as seções *XML* discutidas anteriormente com o acréscimo do conceito de nó raiz, que contém todo o documento e representa o início da árvore. O nó raiz é apenas conceitual, pois não existe um elemento visível ou texto no documento que represente o nó raiz. Entretanto, ele é muito importante pois serve de ponto de referência para o *XPath* navegar no documento *XML*, pois independente do que se deseja pesquisar ou selecionar este nó sempre será visitado.

A expressão mais útil do *XPath* é o caminho de localização. O caminho de localização identifica o conjunto de nós em um documento. Este conjunto pode ser vazio, pode conter um único nó, ou pode conter muitos nós. Ele pode representar nós de elementos, atributos, *namespaces*, texto, comentários, instruções de processamento, raízes, ou qualquer combinação

destes. Cada passo do caminho é avaliado relativo a um nó particular do documento chamado de contexto do nó.

Um caminho de localização produz um *node-set* [Skonnard 2002]. O *node-set* é o nome dado para o nó de contexto, que é o nó resultante ou nó atual do caminho. O caminho pode ser absoluto ou relativo como demonstrado na tabela 2. O caminho absoluto inicia com o caractere / e o relativo por qualquer caractere diferente de /. Um caminho consiste de um ou mais passos de localização, cada um deles separado pelo caractere /.

Tabela 2: Exemplo absoluto e relativo do *XPath*.

Xpath	Descrição	Tipo
/fatura/item	**Identifica os elementos filhos *item* elemento *fatura* raiz.**	**Absoluto**
item/quantidade	**Identifica os elementos *quantidade* dos nós de contexto dos elementos filho *item***	**Relativo**
Preco | quantidade | unidade	**Identifica os filhos dos nós de contexto dos elementos *preco*, *quantidade* e *unidade***	**Absoluto**

Os passos de localização consistem de um identificador *axis*, um nó de teste, e zero ou mais predicados, conforme figura 3.

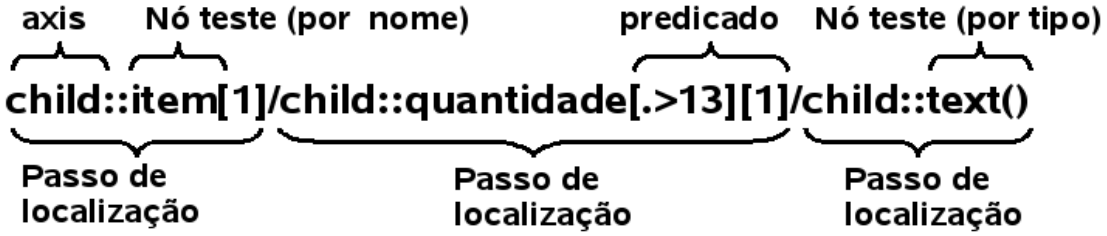

Figura 3: Exemplo de localização e seus elementos. Adaptado de [Skonnard 2002].

Cada passo de localização da figura 3 é avaliado considerando seu axis, nó de teste e predicado. Na tabela 3 podemos verificar os valores possíveis para cada elemento do passo de localização. Estes elementos são sempre avaliados para um nó ou conjunto de nós correntes que também é chamado de *node-set*.

Tabela 3: Elementos possíveis no *XPath*.

Axis	Descrição
self	Identifica o nó do contexto.
child	É o padrão do a*xis*. Identifica o filho do nó do contexto. Quando o a*xis* é omitido, o *child* é assumido.
parent	Identifica o pai do nó do contexto.

Axis	**Descrição**
descendant	Identifica os descendentes do nó do contexto.
descendant-or-self	Identifica o nó do contexto e o descendente do a*xis*.
ancestor	Identifica os antepassados do nó do contexto.
ancestor-or-self	Identifica o nó do contexto e os antepassados do a*xis*.
following	Identifica todos os nós que estão depois do nó do contexto no documento, excluindo descendentes, atributos, e nós de *namespaces*.
following-sibling	Identifica o mais próximo do nó do contexto do a*xis* seguinte.
preceding	Identifica todos os nós que estão antes do nó do contexto no documento, excluindo descedentes, atributos, e nós de *namespaces*.
attribute	Identifica os atributos do nó do contexto.
namespace	Identifica os nós do *namespace* do nó do contexto.
Nó de teste	**Descrição**
QName	Verdadeiro para todos os nós tem o nome do *namespace* especificado e são do tipo do nó do *axis* principal.
*	Verdadeiro para todos os nós que são do tipo do *axis* principal.
text()	Identifica o texto dos nós.
comment()	Identifica os comentários dos nós.
processing-instruction (target?)	Identifica os nós de instrução de processamento que combinam as *strings* especificadas.
node()	Identifica todos os nós no *axis* independente do tipo.
Predicado	**Descrição**
[...]	Os predicados são inseridos entre colchetes ([]) e filtram um *node-set* para produzir um novo *node-set*. Todas as operações básicas são suportadas, tais como, +, -, /, *, >, <=, >=, < e literais numéricos, bem como, um conjunto extenso de funções, por exemplo, count, sum, concat, ceiling, etc.

A *XPath* é uma linguagem poderosa para selecionar e navegar em documentos *XML*. Recentemente ela esta sendo embutida em linguagens que utilizam e manipulam *XML*. Isto inclui linguagens de composição de serviços *web* que lidam com documentos *XML* retornados dos *web services*. Como exemplo temos *BPEL e OWL-S* que incluem extensões *XPath* em suas especificações.

2.3 *Web Services* (*WS*)

Atualmente, o principal uso da *web* é o acesso a interação com documentos e aplicações. Na maioria dos casos, tais acessos são realizados por usuários humanos,

tipicamente trabalhando através de navegadores *web*, tocadores de áudio, ou outros sistemas clientes interativos. A *web* pode crescer significantemente em poder e escopo se isto for estendido para suportar comunicação entre aplicações, característica esta permitida pela arquitetura *Web Services (WS)*. *WS* é uma arquitetura que fornece uma forma padronizada de interoperação entre diferentes aplicações, rodando sobre uma variedade de plataformas e/ou *frameworks* [W3C 2004].

A arquitetura *WS* foi concebida como uma solução padronizada para utilização na integração de sistemas e na comunicação entre aplicações distintas. Estas aplicações podem estar disponíveis em linguagens e plataformas diferentes sem com isto gerar problemas na interoperação entre elas. Esta tecnologia possibilita que aplicações novas interajam com aplicações antigas, possibilitando deste modo o reuso e a construção de aplicações mais confiáveis e rápidas em rede.

Os *WS* combinam os melhores aspectos do desenvolvimento baseado em componentes na *web*. Assim como componentes de software reutiliáveis, os *WS* representam uma funcionalidade fechada que podem ser reutilizados sem a preocupação com a linguagem utilizada no seu desenvolvimento [Graham 2002].

Os serviços disponíveis em *WS* tornaram-se muito populares por permitir que aplicações antigas disponibilizassem serviços em rede sem que estas precisassem ser reescritas. Esta capacidade de disponibilizar serviços, logo exigiu que os serviços fossem integrados, possibilitando a transferência de informações e composição de serviços complementares. Com isto nasce à tecnologia *SOA*.

2.3.1 Arquitetura *Web Services*

A arquitetura *WS* é um sistema de *software* desenhado para suportar interoperabilidade da interação máquina-para-máquina sobre uma rede [W3C 2004]. A comunicação nestas interações dá-se a partir da troca de mensagens *XML*, a qual permite que as aplicações enviem e recebam dados de forma transparente e universal. Para tornar isto possível foi criada uma especificação que contém algumas tecnologias que tornam esta arquitetura factível. Estes componentes são: *XML*, *WSDL, UDDI*, *HTTP* e *SOAP*, conforme descritos abaixo:

- ***XML* (*eXtensible Markup Language*):** é uma linguagem de marcação que define um formato para a criação de documentos com dados organizados de forma hierárquica.

- ***WSDL* (*Web Services Description Language*):** é uma linguagem baseada em *XML* utilizada para descrever os *WS*. Esta descrição especifica como acessar *WS*, bem como, as operações e métodos disponíveis nos *WS*.
- ***UDDI* (*Universal Description Discovery and Integration*):** é a especificação do registro de serviços, ou seja, é uma espécie de páginas amarelas dos serviços *WS*, que contêm as informações para localização de um serviço e metadados dos *WS*. Estes metadados são as descrições *WSDL* comentadas anteriormente.
- ***HTTP* (*HyperText Transfer Protocol*):** é o protocolo padrão de aplicação do modelo *OSI* (*Open Systems Interconnection*) utilizado para transferência de dados na *Internet*.
- ***SOAP* (*Simple Object Access Protocol*):** é um protocolo para troca de informações estruturadas em uma plataforma descentralizada e distribuída utilizando tecnologia *XML*, que pode trafegar sobre outros protocolos, tal como, *HTTP*.

Considerando estas definições pode-se dizer que os serviços *WS* são aplicações que disponibilizam métodos e operações que são publicadas via *WSDL* e podem ser encontradas em um serviço *UDDI*. Estes *WS* possibilitam trocar informações com outras aplicações utilizando o protocolo *SOAP*, rodando opcionalmente, sobre *HTTP*. Neste contexto os serviços *WS* atuam como verdadeiros tradutores, conversando internamente com a plataforma padrão do seu legado e externamente com outras aplicações que falam a tecnologia *WS*.

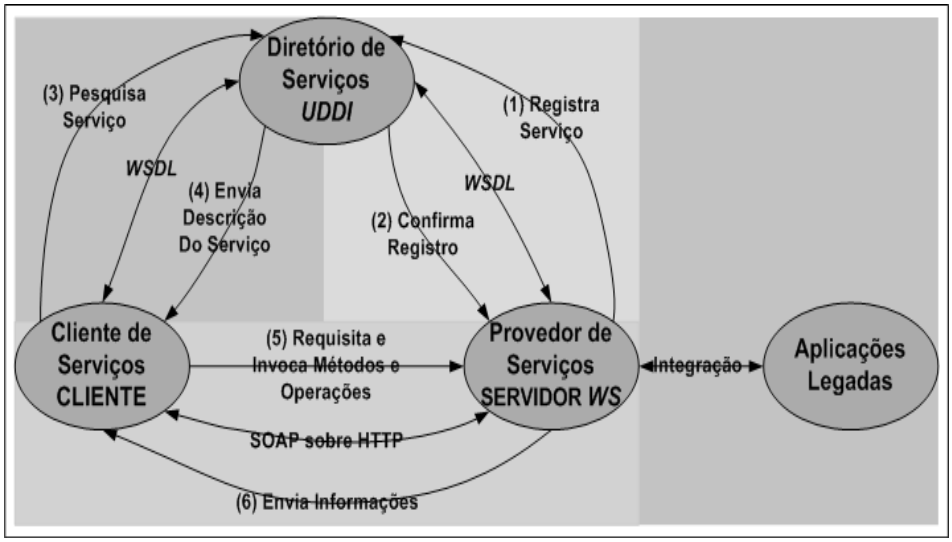

Figura 4: Modelo de referência *WS*. Adaptado de [W3C 2004].

O modelo de referência apresentado na figura 4 pode ser entendido como descrito abaixo:

1) O provedor de serviços (SERVIDOR) que contém os *WS* implementados registra um serviço no diretório de serviços (*UDDI*) utilizando o protocolo *WSDL*.

2) O diretório de serviços (*UDDI*) retorna uma confirmação de registro utilizando *WSDL*.

3) O cliente de serviços (CLIENTE) faz uma pesquisa de serviço utilizando *WSDL* no diretório de serviços (*UDDI*) passando parâmetros (categoria, tipo, nome, etc) que possibilitem localizar o serviço desejado.

4) O diretório de serviços (*UDDI*) retorna a descrição e detalhes do serviço pesquisado via *WSDL* para o cliente de serviços (CLIENTE).

5) O cliente de serviços (CLIENTE) faz uma requisição por invocação de método via *SOAP* ao provedor de serviços (SERVIDOR) que prontamente executa o *WS* que pode se integrar com o legado.

6) O provedor de serviços (SERVIDOR) envia as informações solicitadas ao cliente de serviço (CLIENTE).

Os passos descritos anteriormente permitem identificar os componentes *WS* e suas interações quando em execução. Estas interações também podem ser melhor entendidas quando visualizadas em uma estrutura em camadas, como proposto por [Cerani 2002] e apresentado na figura 5.

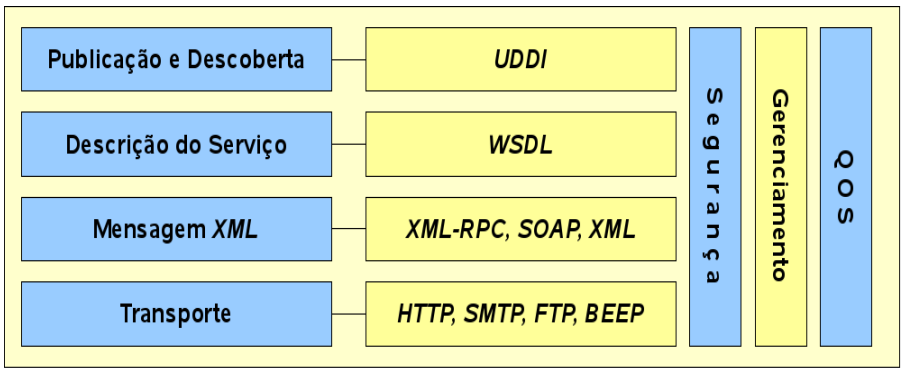

Figura 5: Modelo *WS* em camadas. Adaptado de [Cerani 2002].

2.3.2 Tecnologias da Arquitetura *Web Services*

Existem algumas tecnologias, além da *XML* discutida anteriormente, que merecem ser estudadas mais a fundo, devido a sua importância na arquitetura *WS*. Estas tecnologias são: *WSDL*, *SOAP*, e *UDDI;* as quais serão discutidas e exemplificadas nos próximos itens.

2.3.2.1 *Simple Object Access Protocol* (*SOAP*)

Segundo [Sharpe 2007], *SOAP* (*Simple Object Access Protocol)* é um formato *XML* para transmissão de dados via *WS*. Ele é um protocolo desenhado para trocar informações entre computadores. Embora *SOAP* possa ser utilizado em uma variedade de sistemas de mensagens e possa ser implementado em uma variedade de protocolos de transporte, o foco inicial de *SOAP* são as chamadas de procedimentos remotos transportados via *HTTP*. *SOAP* habilita aplicações clientes conectarem-se facilmente em serviços remotos e invocar métodos remotos. Conforme [Cerani 2002], outros *frameworks*, incluindo *CORBA, DCOM,* e *Java RMI*, fornecem as mesmas funcionalidades de *SOAP,* entretanto as mensagens *SOAP* são as únicas escritas inteiramente em *XML* e portanto é independente de plataforma e linguagem de programação.

A especificação *SOAP* define três partes principais [Cerani 2002]:

- **Envelope:** a especificação do envelope *XML* do *SOAP* define regras para encapsulamento de dados transferidos entre computadores. Isto inclui os dados específicos de uma aplicação, tais como, o nome dos métodos à invocar, parâmetros dos métodos, ou os valores de retorno. Pode incluir também informações a respeito de quem deveria processar o conteúdo do envelope e, em caso de falha, como codificar as mensagens de erro.
- **Codificação:** para trocar dados, os computadores devem concordar com regras de codificação de tipos específicos de dados. *SOAP* inclui seu próprio conjunto de convenções para codificar tipos de dados. A maioria destas convenções são baseadas na especificação do esquema *XML* da *W3C*[2].
- **Convenções *RPC*:** *SOAP* pode ser utilizado em uma variedade de sistemas de mensagens, incluindo mensagens direcionais e bidirecionais. No caso de mensagens bidirecionais, *SOAP* define uma simples convenção para representar chamadas e repostas de procedimentos remotos. Isto permite a uma aplicação

2 *W3C - http://www.w3.org/*

cliente especificar um nome de método remoto, incluindo qualquer número de parâmetros, e receber um resposta do servidor.

Um exemplo de funcionamento do protocolo *SOAP* é o uso do serviço de temperatura do *XMethods*[3]. O *XMethods* fornece um simples serviço de temperatura, o qual informa a temperatura atual por *CEP*. O método do serviço, *getTemp*, requer uma *string* com o *CEP* e retorna um simples valor flutuante. Na figura 6 é ilustrado o funcionamento deste serviço, onde o cliente *SOAP* solicita a temperatura atual de um determinado *CEP* e recebe do servidor *SOAP* a resposta a esta solicitação.

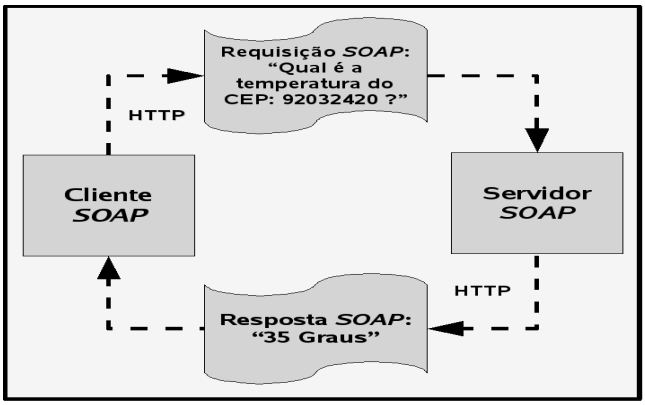

Figura 6: Funcionamento do serviço *SOAP* de temperatura por *CEP* do *XMethods.net*.

O cliente requisitante deve incluir o nome e os parâmetros do método invocado. Na tabela 4 é apresentado o código de requisição e resposta do serviço de temperatura do *XMethods*.

Tabela 4: Requisição e resposta do serviço de temperatura .

Componente	Requisição	Resposta
Cabeçalho HTTP	POST /perl/soaplite.cgi HTTP/1.0 Host: services.xmethods.com	HTTP/1.1 200 OK Date: Fri, 26 Dec 2008 13:13:13 GMT Server: Apache/1.3.14 (Unix) tomcat/1.0 PHP/4.0.1pl2 SOAPServer: SOAP::Lite/Perl/0.50 Cache-Control: s-maxage=60, proxy-revalidate
SOAP/HTTP Binding	Content-Type: text/xml; charset=utf-8 Content-Length: 538 SOAPAction: "urn:xmethodsTemperature#Temperature" <?xml version='1.0' encoding='UTF-8'?>	Content-Length: 539 Content-Type: text/xml <?xml version='1.0' encoding='UTF-8'?>
Envelope SOAP	<SOAP-ENV:Envelope	<SOAP-ENV:Envelope
Cabeçalho SOAP	xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">	xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
Corpo SOAP	<SOAP-ENV:Body> <ns1:getTemp xmlns:ns1="urn:xmethods-Temperature" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"> <zipcode xsi:type="xsd:string">10016</zipcode> </ns1:getTemp> </SOAP-ENV:Body>	<SOAP-ENV:Body> <ns1:getTempResponse xmlns:ns1="urn:xmethods-Temperature" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"> <return xsi:type="xsd:float">71.0</return> </ns1:getTempResponse> </SOAP-ENV:Body>

3 *XMethods - http://www.xmethods.net/*

Componente	Requisição	Resposta
Cabeçalho HTTP	</SOAP-ENV:Envelope>	</SOAP-ENV:Envelope>

Como pode-se perceber na tabela 4, uma mensagem *SOAP,* é sempre uma mensagem *XML*, que se desdobra em uma chamada de serviço no envelope *SOAP*. Este envelope é necessário tanto na requisição do cliente quanto na resposta do servidor *SOAP* e tem por finalidade acionar o *WS*. Enquanto isto, o cabeçalho e o *binding* tem por objetivo encapsular e transportar a mensagem *SOAP* utilizando algum protocolo de transporte, como exemplo, o *HTTP*. O envelope *SOAP* também é subdividido em cabeçalho e corpo, onde o primeiro diz respeito aos *namespaces* da mensagem *SOAP* e o segundo é a invocação da operação do *WS* com seus parâmetros ou a resposta de retorno.

2.3.2.2 *Web Service Description Language* (*WSDL*)

Enquanto *SOAP* é utilizado como um formato de troca de mensagens, *WSDL* é usado para descrever a chamada de um *WS*. *WSDL* é um vocabulário *XML* para descrever *WS*, onde eles estão localizados, e como eles podem ser chamados [Woods 2006]. Usando a descrição *WSDL* de um *WS* pode-se tipicamente codificar um cliente *SOAP* automaticamente, sem a necessidade de nenhuma programação manual.

O documento *WSDL* descreve o que, como, e onde encontram-se os *WS*, como apresentado no documento *WSDL* simplificado da figura 7.

<?xml version="1.0" encoding="utf-8" ?> <definitions>	
<types> ... <element name="qty" type="string" minOccurs="0"/> ... </types> <message name="POMessageIn"> ... <part name="Quantity" type="qty"/> ... </message> <portType name="POPortType"> <operation> ... <input message="POMessageIn"/> ... </operation> </portType>	O que
<binding name="SOAP" portType="POPortType"> ... SOAP/HTTP binding definition ... </binding	Como
<service name="OrderWineService"> <port name="Order" binding="SOAP"> <address location="http://www.unisinos.br/Order/"/> </port> </service>	Onde
</definitions>	

Figura 7: *WSDL* simplificado.

Como pode-se verificar na figura 7, um *portType* descreve uma interface abstrata, tipo de *WS,* do *WS* e as operações de execução. Cada operação de execução pode ter uma entrada (*input*), uma saída (*output*) e mensagens de erro (*fault*). As diferentes mensagens são construídas a partir de tipos de dados embutidos ou customizados. Tipos de dados são definidos utilizando *XML Schema,* e pode ser dados complexos. Na seção *binding* é especificado exatamente um protocolo para a operação do *portType*. Isto diz como deve-se dar a comunicação com o *WS*. Por fim, a seção *service* especifica o endereço de rede do serviço, determinando onde encontra-se o serviço.

2.3.2.3 *Universal, Description, Discovery and Integration* (*UDDI*)

UDDI é uma especificação técnica utilizada para descrever, descobrir, e integrar web services. Portanto, *UDDI* é um componente crítico da pilha de protocolos do WS, possibilitando as companhias publicar e pesquisar *WS* [Cerani 2002]. Ele é composto por duas partes: uma especificação técnica para construir e distribuir *WS*, a qual permite que as informações sejam armazenadas em um formato *XML* específico e o *UDDI Business Registry*, que é uma implementação operacional completa da especificação *UDDI* [Newcomer 2002].

Segundo [Hansen 2003; Cerani 2002], os dados capturados pela *UDDI* podem ser divididos em três categorias principais:

- **Páginas brancas:** inclui informações gerais a respeito de uma companhia específica, por exemplo, nomes e descrições de negócios, informações de contato, endereços e números de telefone. Ele pode incluir identificadores únicos do negócio, tais como, número *D-U-N-S*[4] (Estados Unidos) e *EAN*[5] *(*Brasil*)*.
- **Páginas amarelas:** inclui dados de classificação geral tanto da companhia quanto dos serviços oferecidos por esta. Por exemplo, estes dados poderiam incluir indústria, produto, ou códigos geográficos baseados em padrões de taxonomia.
- **Páginas verdes:** este categoria contém informações técnicas a respeito de um *WS*. Geralmente, inclui um ponteiro para uma especificação externa e um endereço para invocar o *WS*. *UDDI* não é restrito para descrever *WS* baseados em *SOAP*. Ou seja, pode incluir desde uma simples página *web* ou endereço de *email* até outras tecnologias mais sofisticadas, tais como, *CORBA* e serviços *RMI* do *Java*.

4 *D-U-N-S - Data Universal Numbering System*
5 *EAN - European Article Number*

UDDI e *WSDL* são complementares, de modo que *UDDI* tem um papel similar a um banco de dados de *WSDL* [Weerawarana 2005]. Por esta razão, as entidades da *UDDI* possuem correspondência direta com a *WSDL* como relacionado na tabela 5.

Tabela 5: Mapeamento entre *UDDI* e *WSDL* [Weerawarana 2005].

Entidade WSDL	*Entidade Correspondente do UDDI*
portType	**tModel**
binding	**tModel**
service	**businessService**
port	**bindingTemplate**

O relacionamento entre *UDDI* e *WSDL* pode ser melhor visualizado na figura 8.

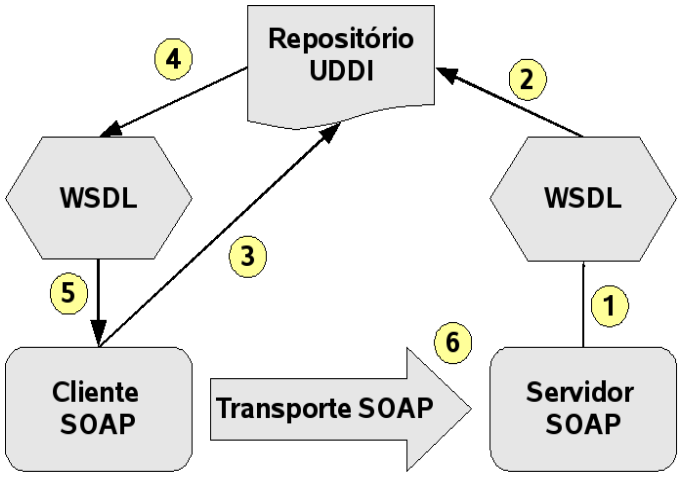

Figura 8: Uso do *UDDI* para descobrir um *WS* [Newcomer 2002].

Na figura 8, o servidor *SOAP* gera um arquivo *WSDL* (1) que é publicado no repositório *UDDI* (2) utilizando a *API UDDI* específica para esta operação. O *WSDL* publicado no *UDDI* pode conter informações sobre identificação e classificação do *WS*. Estas informações são importantes para a efetivação da pesquisa realizada pelo cliente *SOAP*. Após realizada a pesquisa pelo cliente *SOAP* (3), o repositório *UDDI* retorna o *WSDL* (4) que é processado pelo cliente *SOAP* (5). Tendo conhecimento do endereço e da definição *WSDL* do serviço desejado, o cliente *SOAP* inicia a troca de mensagens (6) com o servidor *SOAP*.

2.3.3 *REST*

REST (*Representational State Transfer*) é um estilo de arquitetura de *software* para sistemas distribuídos de hipermídia que representa um modelo de como a *Web* deve funcionar [Fielding 2000]. O modelo *REST* utiliza um conjunto e interfaces genéricas para promover interações sem estado (*stateless*) através da transferência de representações de recursos, em

vez de operar diretamente sobre esses recursos. O conceito de recurso é a principal abstração deste modelo. Esta abstração viabiliza o uso de recursos identificados pelas suas *URIs* (*Uniform Resource Identifiers*) e sua manipulação por troca de mensagens *statless* sobre *HTTP*. Esta faceta da arquitetura permite que *WS* desenvolvidos com conceitos *REST* sejam utilizados como recursos com o protocolo *HTTP,* tornando-se assim uma alternativa ao *SOAP*.

Segundo[Pautasso 2008], o estilo *REST* é baseado em quatro princípios:

- **Identificação do recurso através de *URI*:** um *Web Services* com implementação *REST* expõe um conjunto de recursos que identificam as formas de interação com seus clientes. Estes recursos são identificados por *URIs*, que fornecem um espaço de endereçamento global para descoberta de recursos e serviços;
- **Interface uniforme:** os recursos são manipulados utilizando um conjunto fixo de quatro operações: criar, ler, atualizar e excluir, respectivamente, *PUT*, *GET*, *POST* e *DELETE*. *PUT* cria um novo recurso, que pode ser excluído utilizando o *DELETE*. *GET* retorna o estado corrente do recurso em algumas representações. *POST* transfere um estado novo para dentro do recurso;
- **Mensagens auto-descritivas:** os recursos são dissociados da sua representação, para que seu conteúdo possa ser acessado em uma variedade de formatos (por exemplo, *HTML*, *XML*, formato texto, *PDF*, *JPEG*, etc.). Metadados do recurso esta disponível é utilizado, por exemplo, para controlar *caching*, detectar erros de transmissão, negociar o formato de representação mais apropriado, e executar autenticação ou controle de acesso;
- **Interações *stateful* através de *hiperlinks*:** todas as interações com um recurso são *stateless* (sem estado), por exemplo, mensagens de requisições são auto-contidas. Interações *stateful* (com estado) são baseadas no conceito de transferência explicita de estado. Existem muitas técnicas de troca de estados, por exemplo, reescrita de *URI*, *cookies*, e campos de formulário escondidos. Estados podem ser incorporados nas mensagens das respostas para apontar os estados futuros válidos da interação;

Diferentemente do *SOAP*, *REST* prevê o uso apenas do protocolo *HTTP,* e utiliza todos os comandos e facetas deste protocolo representados pelos comandos *PUT*, *GET*, *POST* e *DELETE*. Além desta característica, outro aspecto arquitetural do *REST,* é a ausência do uso do protocolo *WSDL*. De fato, o *REST* não utiliza nenhuma das especificações *WS* definidas

pela *W3C*[6], também chamadas de especificações *WS-**. Na tabela 6 são comparadas as principais características do serviços *REST* e *SOAP* (*WS-**):

Tabela 6: Comparação dos serviços *REST* e *SOAP* (*WS-). Adaptado de [Shaw 2008].**

Característica	Serviços *REST*	*SOAP* (*WS-**)
Endereçamento	Limitado a topologia de endereçamento via *URI* e comandos *HTTP*	Disponibilizado via *WS-Addressing*
Transporte	*HTTP* e *HTTPS*	Qualquer (por exemplo, *RPC* e *TCP/IP*)
Codificação	Somente texto	Qualquer (por exemplo, binário)
Padrão de troca de mensagem	Apenas pedido de resposta	Qualquer (por exemplo, duplo)
Topologia	Ponto a ponto	Qualquer (por exemplo, *service broker*)
Ordenação	Não suportado	Disponível via *WS-ReliableMessaging*
Transações	Não suportado	Disponível via *WS-AtomicTransactions*
Segurança	Somente segurança de transporte via *HTTPS*	*HTTPS*, *WS-Security*, *WS-SecureConversation*, e *WS-Trust*

Como pode-se perceber na tabela 6, os serviços *REST* estão focados no protocolo *HTTP* e *HTTPS*, enquanto *SOAP (WS-*)* segue um conjunto de protocolos e especificações mais amplos. Estas limitações do *REST* não tiram a suas capacidades de implementações e soluções *WS*. Segundo [Shaw 2008], na maioria das soluções os itens não suportados ou parcialmente suportados pelo *REST* não são necessários. Além disso, *REST* possui uma implementação mais simples do que *SOAP* (*WS-**), porque segue uma arquitetura que exige menos protocolos. Isto faz com que o *REST* seja uma boa alternativa ao *SOAP*.

2.4 *SOA*

SOA não é tecnologia, e *SOA* não acontece por acaso [Rosen 2008]. *SOA* é uma abordagem arquitetural para construir sistemas que necessitam investimentos em arquitetura e *TI*, uma visão de estratégia e negócio, disciplinas de engenharia e governança, e suporte a estrutura organizacional. É uma abordagem de desenvolvimento de software na qual seus serviços são construídos como componentes reutilizáveis que visam fornecer um baixo acoplamento e interoperabilidade [Pijanowski 2007]. Este modelo preconiza que todas as

6 *W3C – http://www.w3c.org*

funcionalidades das aplicações devem ser disponibilizadas em forma de serviços. Sendo assim, o modelo *SOA* utiliza-se fortemente dos conceitos de *WS* e estende estes ao propor a padronização e melhores práticas na construção e uso de serviços.

A arquitetura *SOA* difere da arquitetura tradicional, por ser dinâmica e possibilitar a composição de aplicações em tempo de execução, usando os serviços disponíveis. A arquitetura tradicional é estática, tendo como foco apenas o projeto e a codificação da aplicação [Nakamura 2004]. Temas como gerenciamento do ciclo de vida, registro de serviços, interoperabilidade, orquestração, e composição dinâmica de software, não são completamente atendidas na arquitetura tradicional, o que torna a arquitetura *SOA* um novo paradigma no desenho e implementação de software.

Os benefícios mais visíveis da arquitetura *SOA* estão no reuso, colaboração, interoperação e produtividade no desenvolvimento de *software*, pois esta arquitetura visa justamente a construção de soluções baseadas em serviços existentes com a composição e orquestração destes [Juric 2007]. O poder existente na composição e orquestração de serviços são o grande diferencial da arquitetura *SOA*. Estas características permitem que soluções antes impensáveis pelo custo e tempo de implementação tornassem muito mais fáceis de implementar. É possível inclusive integrar aplicações antigas com novas tecnologias, sem necessidade de reescrever estas, apenas construindo serviços, criando composições e orquestrando as chamadas.

SOA proporciona uma solução para reduzir custos, ganhar tempo e integrar o antigo com o novo utilizando serviços *WS*. Essa nova abordagem arquitetural tem colaborado para substituir aos poucos as abordagens centralizadas.

2.4.1 Abordagem Arquitetural *SOA*

A figura 9 apresenta um modelo de arquitetura do *W3C* que encapsula diferentes conceitos, como: política, mensagens, recursos e ações que traduzem um modelo geral de arquitetura *SOA* [W3C 2004]. Um serviço é controlado por meio de políticas; descrito por meio de recursos, por exemplo: metadados; e envia as respostas às requisições por meio de mensagens. Os recursos são mantidos por organizações ou pessoas e geralmente possuem um registro de endereço. Políticas são estabelecidas por organizações e empresas, referentes a um recurso ou serviço. As mensagens são originadas por um solicitante possuindo em seu conteúdo um corpo e um cabeçalho, e são entregues por algum tipo de protocolo de mensagem.

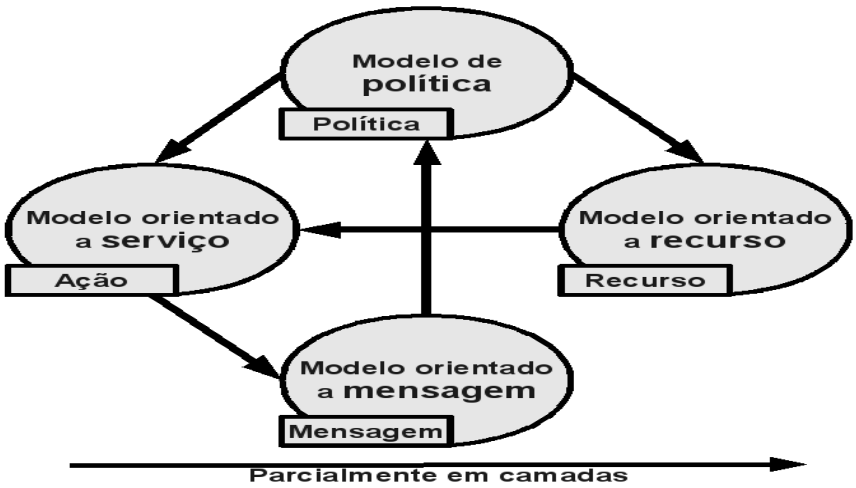

Figura 9: Modelo W3C SOA [W3C 2004].

Conforme apresentado na figura 9, os modelos da arquitetura *SOA são*:

- **Modelo orientado a mensagem:** define a mensagem em termos de conteúdo (cabeçalho e corpo), transporte de entrega, solicitante e provedor do serviço;
- **Modelo orientado a recurso:** define os recursos em termos de endereço (*URI*), representação e dono do recurso;
- **Modelo de políticas:** define a política em termos de recursos, aplicada também para descrições dos serviços;
- **Modelo orientado a serviço:** mais complexo de todos. Um serviço é oferecido e utilizado, sendo mediado por meio de trocas de mensagens.

Do ponto de vista dos serviços os princípios que a arquitetura *SOA* deve seguir são [Erl 2005]:

- **Reusabilidade de serviços:** os serviços devem ser reutilizados. Para alcançar este objetivo os componentes devem ser abstraídos e encapsulados com uma separação clara entre a interface e a lógica da implementação.
- **Contrato de serviços:** este princípio é um reforço as definições *WSDL* e ao conceito de interfaces de aplicação, onde todo o serviço deve fornecer claramente sua forma de acesso e objetivos com um contrato.
- **Baixo acoplamento dos serviços:** o nível de acoplamento é muito importante na arquitetura *SOA*. Ele deve ser o menor possível para garantir a independência e o estímulo a composição de serviços.

- **Abstração dos serviços:** o propósito deste princípio é esconder os detalhes do serviço. O sucesso deste princípio esta diretamente ligado a implementação do contrato de serviço.
- **Composição de serviços:** deve ser apto a suportar os conceitos de associação, tais como, agregação e composição de serviços.
- **Autonomia dos serviços:** o princípio de autonomia esta associado ao princípio do baixo acoplamento dos serviços. Este princípio é importante pois enfatiza a independência e auto-suficiência de um dado serviço.
- **Serviços sem estado:** a atomicidade e o funcionamento dos serviços sem guardar estado é desejável, pois deste modo os serviços passam a ser mais responsivos e independentes de estado.
- **Descoberta de serviços:** este princípio preconiza a necessidade de características nos serviços que permitam e facilitam a sua localização.

Todos os princípios apresentados anteriormente tem equivalentes no paradigma orientado a objetos. Estes princípios, quando implementados adequadamente, elevam a robustez da arquitetura *SOA*.

2.5 *Linguagens de Composição de Serviços Web*

A composição de serviços *web* é caracterizada pela execução de dois ou mais serviços com objetivo de atender um processo de negócio [Crespo 2000]. As composições de serviços podem ser classificadas como orquestração ou coreografia. A orquestração ocorre quando um serviço ou processo central controla e coordena os demais serviços em um fluxo de processos. Em contrapartida, na coreografia não existe a figura do processo central e sim uma colaboração entre os serviços que compõem a composição. Em ambos os casos os serviços não sabem e não precisam saber que estão envolvidos em um processos de composição e também que estão fazendo parte de um processo de nível mais alto. Na figura 10 pode-se perceber claramente a distinção entre o modelo de composição por orquestração e coreografia.

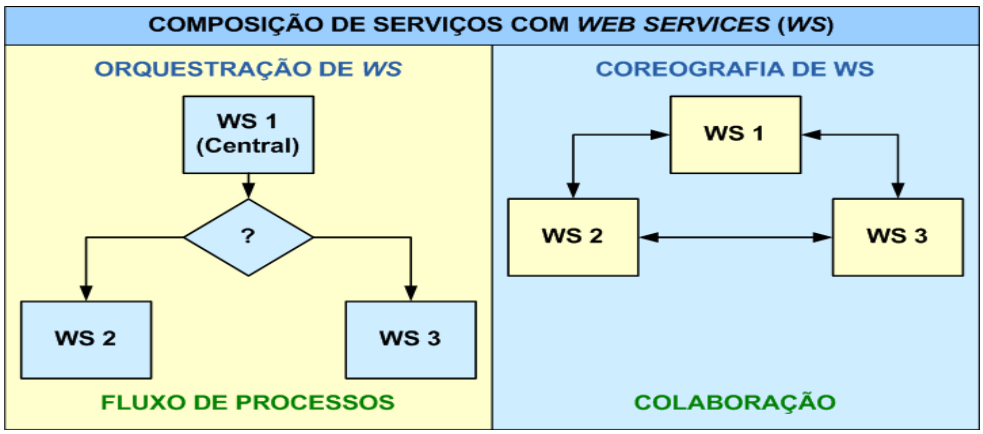

Figura 10: Composição de serviços WS por orquestração e coreografia.

Segundo [Aalst 2003] a ideia de composição de serviços por orquestração é tão antiga quanto as primeiras iniciativas de *workflow* ocorridas nas décadas de 80 e 90, as quais visavam atender a forte demanda por orquestração de processos de negócios. Estas inciativas, como pode-se perceber na linha histórica da automação de processos e sistemas de *workflow* apresentada na figura 11, produziram muitos produtos e exigiram muito esforço nas últimas décadas, gerando com isto muito conhecimento e experiências no tema de composição de serviços. Portanto, para se avaliar e desenvolver novas linguagens de composição e orquestração de serviços é razoável que as experiências e *design patterns* adquiridos até então com orquestração sejam considerados.

Design patterns ou padrões de projetos descrevem um problema que ocorre repetidas vezes em nosso ambiente [Gammar 1994]. Esta descrição apresenta o núcleo da solução do problema, de tal forma que pode-se usar essa solução muitas vezes sem nunca fazê-la igual duas vezes, mas utilizando sempre a mesma ideia. Padrões de projeto possuem diferentes níveis de abstração, podendo ser classificados em diversas categorias, de modo a facilitar a sua recuperação e utilização num determinado problema. Como referencial para uma efetiva comparação é importante sempre termos um conjunto de *patterns* esperados em um domínio de problema que possam ser comparados e consequentemente medidos.

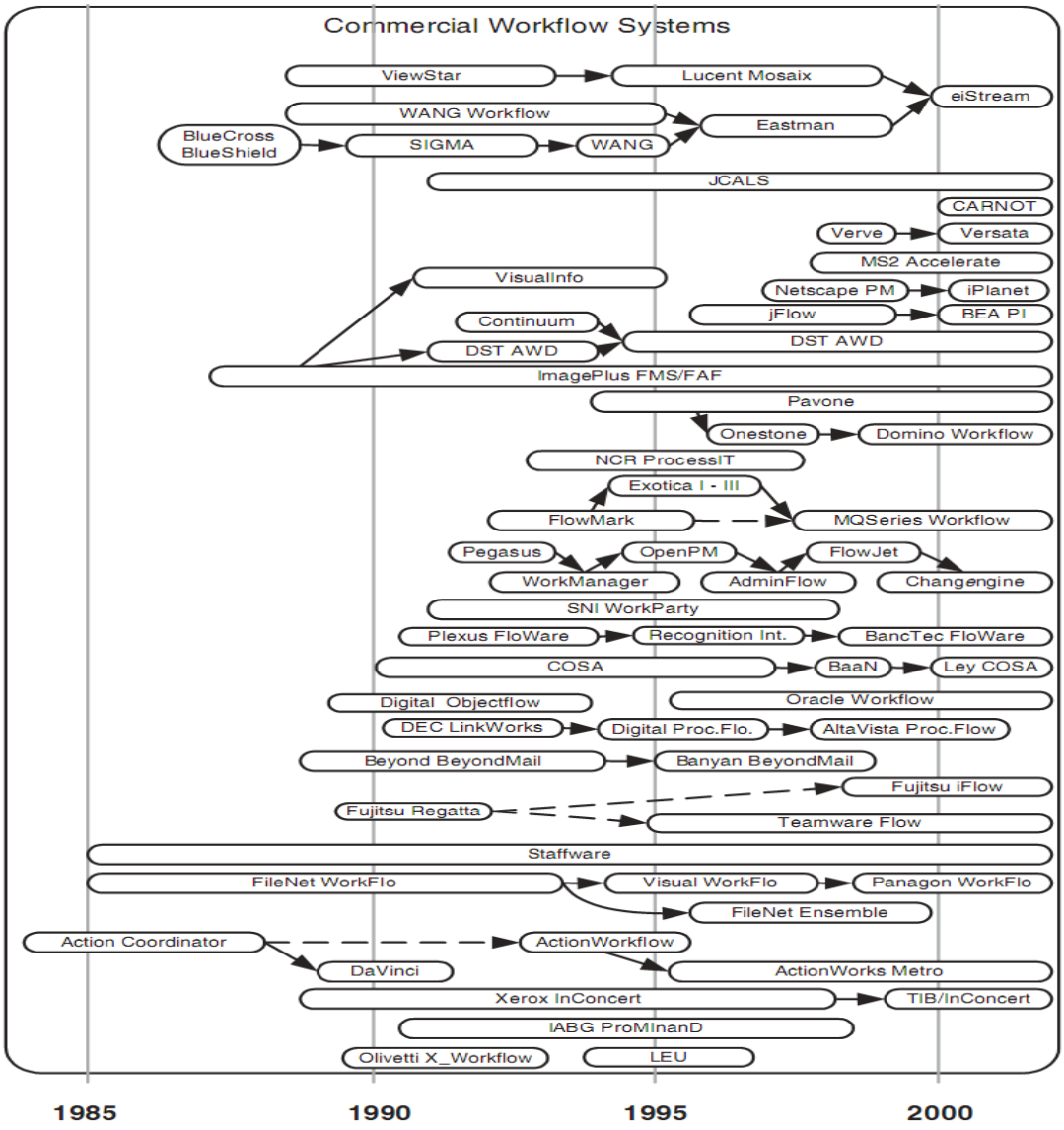

Figura 11: Automação de processos e sistemas de *workflow* [Mühlen 2002].

Composição de serviços web é um paradigma emergente para integração de aplicações dentro e fora das organizações. Conforme [Wohed 2002], este novo paradigma tende a expressar a lógica de composição de serviços web usando uma linguagem de modelagem de processos de negócios adaptada para WS. Segundo [Will 2002], as linguagens de modelagem de processos de negócios e composição de serviços existem há bastante tempo e tem seus patterns já bem definidos, conforme descritos na tabela 7 abaixo:

Tabela 7: *Patterns* para linguagem de modelagem de processos de negócios [Will 2002].

#	*Pattern*	Descrição
1	*Sequence*	Uma atividade em um processo é ativada após a conclusão de outra atividade no mesmo processo.
2	*Parallel split*	Um ponto no processo onde são disparadas múltiplas

#	*Pattern*	Descrição
		atividades que podem ser executadas em paralelo, simultaneamente ou em qualquer ordem.
3	*Synchronization*	Um ponto no processo onde múltiplos subprocessos ou atividades paralelas convergem para um único caminho ou atividade de controle, assim sincronizando múltiplos caminhos. Este padrão pressupõe que cada um dos caminhos é executado apenas uma vez.
4	*Exclusive choice*	Um ponto no processo em que, com base numa decisão ou fluxo de dados, um dos vários caminhos é escolhido.
5	*Simple merge*	Um ponto do processo onde dois ou mais caminhos alternativos se unem sem sincronização. Trata-se da hipótese neste padrão de que nenhum dos caminhos alternativos é executado em paralelo.
6	*Multi-choice*	Um ponto no processo onde, baseado numa decisão ou controle de dados, um número de caminhos são escolhidos.
7	*Synchronizing merge*	Um ponto no processo onde múltiplos caminhos convergem para dentro de um simples caminho. Se mais de um caminho é considerado, a sincronização dos caminhos ativos precisa ser feita. Se apenas um caminho for considerado, a alternativa deveria ser reconvergir sem sincronização. Parte-se do pressuposto que neste padrão o ramo já foi ativado, não podendo ser ativado novamente enquanto ainda está aguardando uma junção dos outros ramos ficar completa.
8	*Multi-merge*	Um ponto no processo onde dois ou mais ramos reconvergem sem sincronização. Se mais de um caminho esta ativo, possivelmente concomitantemente, a atividade seguinte de junção é iniciada para cada ativação de cada ramo recebido.
9	*Discriminator*	O *discriminator* é um ponto processo de que espera por um processo de recebimento de ramos completar antes de ativar a atividade subsequente. Todos os processos posteriores a este processo recebido serão desconsiderados. É utilizado no contexto de um *loop*.
10	*Arbitrary cycles*	Um ponto em um processo em que uma ou mais atividades podem ser feitas repetidamente.
11	*Implicit termination*	Um dado subprocesso deve ser encerrado quando não há mais nada a ser feito.
12	*Multiple instances without synchronization*	No contexto de um caso simples múltiplas instâncias de atividades podem ser criadas sem sincronização.
13	*Multiple instances with a priori design time knowledge*	Uma instância de uma atividade de um processo é habilitada múltiplas vezes. O número de instâncias de uma certa atividade de um certo processo é conhecido em tempo de projeto. Uma vez que todas as instâncias são completadas uma outra atividade pode ser iniciada.
14	*Multiple instances with a priori*	Este é caso em que uma atividade é habilitada várias vezes. O número de instâncias de uma atividade depende

#	*Pattern*	Descrição
	runtime knowledge	de características conhecidas em tempo de execução antes da instância da atividade ser iniciada. Uma vez que todas as instâncias são finalizadas uma outra atividade pode ser iniciada.
15	*Multiple instances without a priori runtime knowledge*	Este é caso em que uma atividade é habilitada várias vezes. O número de instâncias de uma dada atividade não é conhecida nem durante o desenho do projeto e nem durante o tempo de execução. Uma nova instância pode ser criada a qualquer momento, estando finalizado ou não instâncias iniciadas anteriormente.
16	*Deferred choice*	Um ponto do processo onde um dos muitos caminhos é escolhido. Em contraste com o padrão 3 (*Synchronization*), a escolha não é realizada explicitamente, mas muitas alternativas são oferecidas para o ambiente. Entretanto, em contraste com o padrão 2 (*Parallel-Split*), somente uma alternativa é executada. Isto significa dizer que uma vez que um caminho é escolhido os outros são eliminados.
17	*Interleaved parallel routing*	Um conjunto de atividades é executado em uma ordem arbitrária. Cada atividade no conjunto é executada, a ordem é decidida em tempo de execução, e duas atividades não são executadas ao mesmo tempo.
18	*Milestone*	O início de uma atividade depende de um determinado estado. Exemplo: a atividade é inicializada se um certo marco foi alcançado e não expirou ainda.
19	*Cancel activity*	Uma atividade habilitada é desabilitada e removida.
20	*Cancel case*	Um processo completo é desativado e removido.
21	*Exception Handling*	Tratamento de exceções e falhas ocorridas durante a execução de um dado serviço.

Em resposta a este novo paradigma de composição de serviços *web*, uma grande quantidade de linguagens e técnicas de composição de serviços *web* emergiu e está continuamente evoluindo com novas propostas e diferentes soluções. Estas novas soluções possuem sintaxe e modelos distintos, entretanto podem ser classificadas dentro dos *patterns* descritos anteriormente. Abaixo segue uma comparação (tabela 8) entre as principais linguagens (*BPEL4WS*, *XLANG*, *WSFL, XPDL e OWL-S*) de composição de serviços *web* e alguns produtos (*Staffware, MQ Series Workflow, Panagon eProcess* e *FLOWer*) de *workflow* de mercado:

Tabela 8: Comparação das linguagens de composição. Adaptado de [Will 2003].

#	*Pattern*	BPEL4WS	XLANG	WSFL	XPDL	Staffware	MQ Series Workflow	Panagon eProcess	FLOWer	OWL-S
1	*Sequence*	+	+	+	+	+	+	+	+	+
2	*Parallel Split*	+	+	+	+	+	+	+	+	+
3	*Synchronization*	+	+	+	+	+	+	+	+	+
4	*Exclusive Choice*	+	+	+	+	+	+	+	+	+
5	*Simple Merge*	+	+	+	+	+	+	+	+	+
6	*Multi-choice*	+	-	+	+	-	+	+	-	+
7	*Synchronizing Merge*	+	-	+	-	-	+	+	-	+
8	*Multi-merge*	-	-	-	-	-	-	-	+/-	-
9	*Discriminator*	-	-	-	-	-	-	-	+/-	-
10	*Arbitrary Cycles*	-	-	-	+	+	-	+/-	-	-
11	*Implicit Termination*	+	-	+	+	+	+	+	-	+
12	*Multiple Instances Without Synchronization*	+	+	+	-	-	-	+	+	+
13	*Multiple Instances With a Priori Design Time Knowledge*	+	+	+	+	+	+	+	+	+
14	*Multiple Instances With a Priori Runtime Knowledge*	-	-	-	-	-	-	-	+	+
15	*Multiple Instances Without a Priori Runtime Knowledge*	-	-	-	-	-	-	-	+	+
16	*Deferred Choice*	+	+	-	-	-	-	-	+/-	+
17	*Interleaved Parallel*	+/-	-	-	-	-	-	-	+/-	+
18	*Milestone*	-	-	-	-	-	-	-	-	-
19	*Cancel Activity*	+	+	+	-	+	-	-	+/-	+
20	*Cancel Case*	+	+	+	-	-	-	+	+/-	-
21	*Exception Handling*	+	+/-	+/-	-	-	-	+-	-	-

Entre as linguagens de composição para web destaca-se a BPEL4WS [Nakamura 2004; Hackmann 2006] que se tornou um padrão de fato no mercado e a OWL-S [W3C 2004] devido ao seu foco em ontologias. A supremacia da BPEL pode ser explicada por razões históricas: BPEL é baseada na WSFL (Web Services Flow Language) da IBM [Leymann 2001] e na XLANG (Web Services for Business Process Design) da Microsof [Thatte 2001]. BPEL combina as características de estrutura em bloco da XLANG com as características de linguagem gráfica da WSFL. Enquanto isto, a OWL-S deve ser considerada, pois possui um paradigma distinto de todas as outras tecnologias de composição de serviços

2.5.1 BPEL4WS (Business Process Execution Language for Web Services)

BPEL4WS é uma especificação construída a partir das especificações *WSFL* (*Web Services Flow Language*) da *IBM* e *XLANG* da *Microsoft* [Will 2003]. É uma linguagem para definição de composição de serviços *WS* que permite executar e abstrair processos de negócios. Basicamente, a linguagem *BPEL* é divida em quatro partes como descrito abaixo:

- ***partnerLinks*:** é a seção que define os elementos do processo que cooperam entre si de forma a satisfazer os processos de negócio.
- ***variables*:** é a seção que define a estrutura de dados que a aplicação irá utilizar. São definidos os as mensagens *WSDL* e o *XML* com seu esquema *XSD* (*XML Schema Definition*).
- ***faultHandler*:** nesta seção define-se o tratamento de erros e as operações do *WS* que serão chamadas quando ocorrer algum problema.
- ***sequence*:** seção que define a sequência e os procedimentos que serão chamados.

Na figura abaixo segue um exemplo "*Hello World*" utilizando a linguagem *BPEL*:

```
<process name="HelloWorld" targetNamespace = "http://jbpm.org/examples/hello"
    xmlns:tns="http://jbpm.org/examples/hello"
    xmlns:bpel="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
    xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/">
    <partnerLinks>
        <!-- establishes the relationship with the caller agent -->
        <partnerLink name="caller" partnerLinkType="tns:Greeter-Caller" myRole="Greeter" />
    </partnerLinks>
    <variables>
        <!-- holds the incoming message -->
        <variable name="request" messageType="tns:nameMessage" />
        <!-- holds the outgoing message -->
        <variable name="response" messageType="tns:greetingMessage" />
    </variables>
    <sequence name="MainSeq">
        <!-- receive the name of a person -->
        <receive name="ReceiveName" operation="sayHello" partnerLink="caller"
        portType="tns:Greeter" variable="request" createInstance="yes" />

        <!-- compose a greeting phrase -->
        <assign name="ComposeGreeting">
            <copy>
              <from expression="concat('Hello, ', bpel:getVariableData('request', 'name'), '!')" />
              <to variable="response" part="greeting" />
            </copy>
        </assign>

        <!-- send greeting back to caller -->
        <reply name="SendGreeting" operation="sayHello" partnerLink="caller"
          portType="tns:Greeter" variable="response" />
    </sequence>
</process>
```

Figura 12: Exemplo "*Hello World*" com *BPEL*.

O propósito principal do *BPEL4WS* é permitir a orquestração de serviços e processos na *web* utilizando *WS*. Por conta disto, *BPEL* descreve a interação entre *WS* e não os *WS* propriamente dito. Ele utiliza muito a definição *WSDL* para viabilizar a integração dos *WS*. A

linguagem também possui construtores para controle fluxo: *if, while, repeatUntil e forEach*. Pode-se utilizar o elemento <*scope*> para agrupar blocos de subprocessos e <*fault*> para tratar falhas. Existem outros elementos para definição de tempos de espera: quantidade de tempo a esperar, deadline e número de mensagens recebidas. Outro recurso interessante é a possibilidade de definir a execução em paralelo de uma atividade com o comando flow, ao invés de sequencial com o comando *sequence*. Isto em conjunto com o comando link pode definir se o processo é assíncrono ou não, permitindo assim a construção de processos complexos. Os tratamentos de erros e compensação, ou seja, desfazer um processo quando ocorre algum erro no processo também são suportados pela linguagem *BPEL4WS*.

BPEL4WS tornou-se a principal linguagem de composição de serviços ao implementar as principais características das soluções voltadas a *workflow* existentes no mercado. Sua característica procedural apesar de direta é de difícil uso sem uma ferramenta gráfica.

2.5.2 *OWL-S* (*Ontology Web Language for Services*)

OWL-S é uma linguagem de composição e orquestração de serviços com foco em ontologia. Ontologia é um termo que tem origem na filosofia, onde é o nome de um ramo da metafísica que ocupa-se da existência. É uma disciplina que estuda a natureza da existência: os tipos de objetos e seus relacionamentos [Berners-Lee 2001]. Em computação uma ontologia é um modelo de dados que representa um conjunto de conceitos dentro de um domínio e os relacionamentos entre estes. Uma ontologia é utilizada para realizar inferência sobre os objetos do domínio, ou de outra forma uma descrição de conceitos e relacionamentos que podem existir para um agente ou uma comunidade de agentes. Ontologias geralmente descrevem indivíduos, classes, atributos e relacionamentos.

A *OWL* possui a habilidade de expressar informações ontológicas de instâncias de múltiplos documentos fazendo ligação dos dados e dando sentido semântico para eles. Esta habilidade é estendida para serviços com a especificação *OWL-S*. *OWL-S* é o poder da ontologia para serviços na *web*. Segundo [Martin 2004], os usuários e agentes de software deveriam ser capazes de descobrir, invocar, compor e monitorar recursos *web* que oferecem recursos e serviços específicos com propriedades especiais, e deveriam ser capazes de fazê-lo com um elevado grau de automação. Isto é o que permite a linguagem de composição *OWL-S*.

Essencialmente a *OWL-S* cobre três áreas como demonstradas na figura 13:

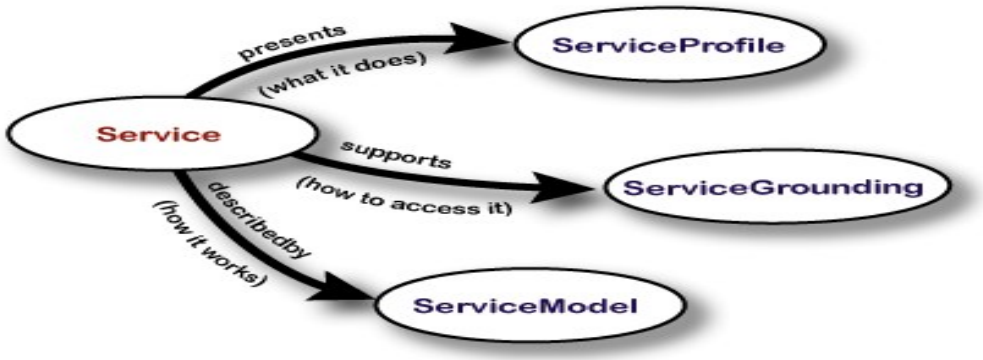

Figura 13: Serviços de ontologia da *OWL-S* [Martin 2004].

- **Descoberta automática de *WS*:** o *service profile* descreve o que o serviço faz em termos de capacidades e é usado para propósitos de descoberta de serviços.
- **Invocação automática de *WS*:** o *process model* descreve formas que os clientes podem interagir com o serviço. Isto inclui as entradas, saídas, pré-condições e resultados da execução do serviço.
- **Composição e interoperação automática de *WS*:** o *service grounding* especifica os detalhes que o cliente precisa seguir para interagir com o serviço, tais como, protocolos de comunicação, formatos de mensagens e números de portas.

Embora as linguagens *OWL-S* e *WSDL* tenham diferentes níveis de especificação e usos, existem algumas interseções entre elas:

- Um processo único e atômico da *OWL-S* corresponde a uma operação *WSDL*.
- As entradas e saídas de um processo atômico *OWL-S* correspondem às mensagens *WSDL*.
- Os tipos de entradas e saídas de um processo atômico OWL-S correspondem aos tipos abstratos da *WSDL*.

Na figura abaixo segue um exemplo *OWL-S* com a definição de um processo de pesquisa de código postal:

```
<?xml version="1.0" encoding="UTF-8"?>
...
<rdf:RDF>
...
<owl:Ontology rdf:about="">
   <owl:imports rdf:resource="http://www.mindswap.org/2004/owl-s/1.1/MindswapProfileHierarchy.owl"/>
</owl:Ontology>

<!-- Service description -->
<service:Service rdf:ID="ZipCodeFinderService">
   <service:presents rdf:resource="#ZipCodeFinderProfile"/>
   <service:describedBy rdf:resource="#ZipCodeFinderProcess"/>
   <service:supports rdf:resource="#ZipCodeFinderGrounding"/>
</service:Service>
```

```
<!-- Profile description -->
<mind:MapService rdf:ID="ZipCodeFinderProfile">
  <service:presentedBy rdf:resource="#ZipCodeFinderService"/>
  <profile:serviceName xml:lang="en">Pesquisa Código Postal</profile:serviceName>
  <profile:textDescription xml:lang="en">Retorna o código postal para uma determinada cidade. Se
    existem mais que um código postal associado com a cidade somente o primeiro é retornado.
    </profile:textDescription>
  <profile:hasInput rdf:resource="#City"/>
</mind:MapService>

<!-- Process description -->
<process:AtomicProcess rdf:ID="ZipCodeFinderProcess">
  <service:describes rdf:resource="#ZipCodeFinderService"/>
  <process:hasInput rdf:resource="#City"/>
</process:AtomicProcess>

<process:Input rdf:ID="City">
  <process:parameterType rdf:datatype="&xsd;#anyURI">http://www.w3.org/2001/XMLSchema#string</
    process:parameterType>
  <rdfs:label>0:City</rdfs:label>
</process:Input>

<!-- Grounding description -->
<grounding:WsdlGrounding rdf:ID="ZipCodeFinderGrounding">
  <service:supportedBy rdf:resource="#ZipCodeFinderService"/>
  <grounding:hasAtomicProcessGrounding rdf:resource="#ZipCodeFinderProcessGrounding"/>
</grounding:WsdlGrounding>

<grounding:WsdlAtomicProcessGrounding rdf:ID="ZipCodeFinderProcessGrounding">
  <grounding:owlsProcess rdf:resource="#ZipCodeFinderProcess"/>
  <grounding:wsdlDocument
    rdf:datatype="&xsd;#anyURI">http://www.tilisoft.com/ws/LocInfo/ZipCode.asmx?
    WSDL</grounding:wsdlDocument>
  <grounding:wsdlOperation>
    <grounding:WsdlOperationRef>
    <grounding:portType
      rdf:datatype="&xsd;#anyURI">http://www.tilisoft.com/ws/LocInfo/ZipCodeSoap</grounding:portTy
      pe>
    <grounding:operation rdf:datatype="&xsd;#anyURI">http://www.tilisoft.com/ws/LocInfo/ListByCity</
      grounding:operation>
    </grounding:WsdlOperationRef>
  </grounding:wsdlOperation>

  <grounding:wsdlInputMessage
    rdf:datatype="&xsd;#anyURI">http://www.tilisoft.com/ws/LocInfo/ListByCitySoapIn</grounding:wsdl
    InputMessage>
  <grounding:wsdlInput>
    <grounding:WsdlInputMessageMap>
      <grounding:owlsParameter rdf:resource="#City"/>
      <grounding:wsdlMessagePart
        rdf:datatype="&xsd;#anyURI">http://www.tilisoft.com/ws/LocInfo/City</grounding:wsdlMessageP
        art>
    </grounding:WsdlInputMessageMap>
  </grounding:wsdlInput>

  <grounding:wsdlOutputMessage
    rdf:datatype="&xsd;#anyURI">http://www.tilisoft.com/ws/LocInfo/ListByCitySoapOut</grounding:ws
    dlOutputMessage>
  <grounding:wsdlOutput>
    <grounding:WsdlOutputMessageMap>
      <grounding:owlsParameter rdf:resource="#ZipCode"/>
      <grounding:wsdlMessagePart
        rdf:datatype="&xsd;#anyURI">http://www.tilisoft.com/ws/LocInfo/ListByCityResult</grounding:w
        sdlMessagePart>
      <grounding:xsltTransformationString>...</grounding:xsltTransformationString>
    </grounding:WsdlOutputMessageMap>
  </grounding:wsdlOutput>
</grounding:WsdlAtomicProcessGrounding>
</rdf:RDF>
```

Figura 14: Exemplo adaptado de pesquisa de código postal em *OWL-S*.

Assim como a *BPEL4WS*, a *OWL-S* tem por objetivo principal descrever a orquestração de serviços *web* utilizando *WS*. Por outro lado, como pode-se perceber no código da figura 14, a *OWL-S* insere em sua especificação na seção *grounding* a definição *WSDL*. Outro ponto interessante é que por definição o código *OWL-S* completo deve ser publicado como recurso num servidor *OWL-S* especializado, de modo, que este possa ser baixado e analisado por seus usuários. A linguagem *OWL-S* também possui controles de fluxo: *if-then-else, iterate, choice, condition e repeat-until*. Possui também vários comandos para controle de construção: *sequence, split, split-join e any-order*. A *OWL-S* também fornece suporte a definição de classes de domínio com o comando *class* e o relacionamento entre classes. Neste relacionamentos é possível definir a cardinalidade e as heranças entre as classes. Talvez este seja o maior diferencial da *OWL-S* em relação a *BPEL4WS*, pois ela permite com este recurso estruturar melhor o conhecimento de domínio e sua semântica.

2.6 Linguagens de Programação

As linguagens de programação são geralmente categorizadas em quatro grupos: imperativo, funcional, lógica, e orientada para objetos [Sebesta 2006]. Estas categorizações são classificações de paradigmas de programação. Estes paradigmas fornecem a visão que o desenvolvedor possui sobre a estruturação e execução de um programa. Por exemplo:

- **imperativo:** o desenvolvedor descreve a computação como ações ou comandos que mudam o estado de um programa;
- **funcional:** os programadores podem abstrair uma aplicação como uma sequência de funções executadas de modo empilhado;
- **lógica:** o programador deste paradigma faz uso da lógica matemática. Diferentemente do paradigma imperativo, onde frequentemente se define "como" deve ser computado, neste paradigma define-se "o que" deve ser executado.
- **orientada para objetos:** desenvolvedores podem abstrair um programa como uma coleção de objetos interagindo entre si;

O agrupamento de características dos diferentes paradigmas geram novas subcategorias de linguagens de programação. Um exemplo é a programação declarativa, que é baseada em programação funcional e programação lógica. Esta subcategoria por sua vez, é bastante difundida em linguagens específicas por domínio, onde o domínio refere-se ao domínio de discurso, ou seja, o assunto ao qual a linguagem está descrevendo. Como exemplo de linguagens declarativas específicas por domínio temos *XSLT* (*eXtensible Stylesheet Language*

Transformations), utilizada para transformar documentos *XML*, e *SQL* (*Structured Query Language*), usada para realizar requisições a banco de dados.

Conforme [Metsker 2002], todas as linguagens de programação são abstrações humanas para a geração de código que é entendido pelo computador. Pessoas trabalham com texto, enquanto computadores trabalham com código binário. A ponte que permite a conexão entre computadores e usuários de linguagens de programação são os compiladores. Os compiladores são programas que a partir de um código escrito em uma linguagem, chamado de código fonte, criam um programa semanticamente equivalente escrito em outra linguagem, chamado de código objeto. O código objeto por sua vez é submetido a um processo de montagem que gera um código que é entendido pela máquina, chamado de código executável.

Figura 15: Fases de um compilador [Aho 2006].

Como pode-se verificar na figura 15 um compilador é formado basicamente por dois grandes grupos: compilador e montador. O compilador é responsável por interpretar um código fonte e transformar este em um código objeto, enquanto o montador tem por objetivo transformar código objeto em código executável. Alguns compiladores do tipo interpretadores não geram código executável, entretanto eles preparam o código objeto para que este seja executado por uma máquina virtual. Uma máquina virtual é uma abstração de *software* de uma máquina real, ou seja, é um computador fictício criado por um programa de simulação.

Um compilador como apresentado na figura 15 é composto de quatro partes [Appel 2004]:

- **Análise léxica:** responsável pela análise dos caracteres do texto do código fonte do programa que gera uma sequência símbolos chamados de *tokens* que podem ser interpretados por um *parser*.
- **Análise sintática:** transforma os símbolos do texto de entrada analisados pela análise léxica em uma árvore sintática que representa a gramática da linguagem. Esta fase é também chamada de *parser* e sua saída é uma árvore sintática.
- **Análise semântica:** extrai os elementos da árvore sintática abstrata. Determina o que cada elemento significa, relaciona o uso das variáveis com suas definições e checa os tipos de expressões, entre outros.
- **Código intermediário:** nesta fase são gerados os códigos intermediários que serão utilizados pelo montador ou pelo interpretador da máquina virtual.

A principal função do montador, por outro lado, é unir os programas compilados de forma independente e unificá-los em um programa executável. Isto inclui criar o programa final em um formato compatível com os requisitos do sistema operacional para carregá-lo na memória e iniciar sua execução.

2.6.1 Gramática de Linguagens de Programação *BNF* e Gerador de Compilador *ANTLR*

Conforme [Aho 2006], a fase de análise de um compilador é organizada em torno da sintaxe da linguagem a ser analisada. A sintaxe de uma linguagem de programação descreve a forma apropriada de escrever o código de um programa. Esta forma de escrever o código segue um padrão formal. Segundo [Appel 2004], as gramáticas livre de contexto, são a forma mais usual de se descrever um padrão formal. As gramáticas livre de contexto têm o seguinte formato: *símbolo*→ *símbolo símbolo ... símbolo*. Onde existem zero ou mais símbolos do lado direito da fórmula após o caractere →. Cada símbolo é um terminal, significando que ele é um *token* do alfabeto da linguagem, ou não terminal, significando que ele aparece do lado esquerdo de alguma produção e representa o início de um símbolo da gramática. Um *token* nunca pode aparecer do lado esquerdo de uma produção. Como pode-se perceber, as gramáticas livres de contexto são um conjunto de regras de derivação.

2.6.1.1 *BNF* (*Backus–Naur Form*)

Tipicamente as linguagens de programação são especificadas utilizando a definição *BNF* (*Backus–Naur Form*) [Friedman 2008]. *BNF* é uma meta-sintaxe utilizada para expressar gramáticas livres de contexto, ou seja, é um método formal de descrever linguagens formais. Ela é amplamente utilizada como notação para linguagens de programação. Por ser uma gramática livre de contexto, a especificação *BNF* também é um conjunto de regras de derivação, escritas da seguinte forma: *<símbolo> ::= <expressão com símbolos>*. Onde o *<símbolo>* é um não-terminal e a *<expressão com símbolos>* consiste de sequências de símbolos. Símbolos que nunca aparecem do lado esquerdo são ditos símbolos terminais. No exemplo da figura 16 temos a definição de um endereço postal completo utilizando a especificação *BNF*:

<endereço> ::= <nome-completo> <rua> <uf-cidade>
<nome-completo> ::= <nome> <sobrenome> | <nome>
<rua> ::= <nome-da-rua> <número> <apto>
<uf-cidade> ::= <CEP> <cidade> "/" <estado>

Figura 16: Especificação *BNF* do endereço postal.

A especificação *BNF* do endereço postal apresentada na figura 16 pode ser lida da seguinte forma: um *endereço* é composto do *nome-completo* seguido da *rua* e da *uf-cidade*. O *nome-completo* por sua vez é composto de um *nome* seguido do *sobrenome* ou somente do *nome* do remetente. A *rua* consiste do *nome-da-rua* seguido do *número* e do *apto*. Por fim, a *uf-cidade* é formada pelo *CEP* seguida da *cidade*, do caractere / e do *estado*. No exemplo descrito, alguns símbolos, como por exemplo, *nome e nome-da-rua*, não foram especificados. Estes símbolos poderiam ser descritos utilizando regras *BNF* adicionais.

Como pode-se perceber no exemplo da figura 16 a *BNF* possui somente um comando mais avançado representado pelo caractere /. Este comando demonstra opcionalidade entre os símbolos de uma regra de produção. Devido a este limitação, outras extensões para *BNF* foram propostas. Um exemplo é a *EBNF* (*Extended Backus-Naur Form*), que estendeu a *BNF* e tornou-se referência nas gramáticas de livre contexto para linguagens de programação [Terry 1996]. As notações acrescentadas a *BNF* pela *EBNF* foram:

- **Não-terminais:** são escritos como simples palavras, tal como em *VarDeclaration*.
- **Terminais:** são escritos com aspas duplas, como em "*BEGIN*".
- **( ):** parenteses são utilizados para denotar agrupamento.
- **[ ]:** colchetes são usados para denotar um ou vários símbolos opcionais.

- **{ }:** chaves são utilizados para denotar uma repetição opcional de um símbolo ou grupo de símbolos.
- **= :** é utilizado no lugar do símbolo ::= ou →.
- **. :** usado para denotar o fim de cada produção.
- ***:** operador de cardinalidade que significa zero ou mais.
- **+:** significa um ou mais.
- **(* *):** permitem comentários.
- **- :** o sinal '-' especifica a faixa entre dois valores. Por exemplo, de '0' a '9' seria declarado como '0-9'.
- **Espaços:** são desconsiderados na avaliação da gramática.

Com as notações trazidas pela extensão *EBNF* a declaração formal das gramáticas das linguagens de programação ganhou em poder e facilidade, permitindo que qualquer gramática pudesse ser definida formalmente utilizando as regras de derivação desta metodologia.

2.6.1.2 *ANTLR* (*Another Tool for Language Recognition*), *StringTemplate e DSL*

A declaração formal da gramática das linguagens de programação trazida pela *EBNF,* logo possibilitou que ferramentas de geração automática de código de compiladores, chamadas de geradores de *parsers*, fossem desenvolvidas e agregadas a variações da *EBNF*. Neste contexto, uma variedade de ferramentas de geração de compiladores foram desenvolvidas [Metsker 2002]. Entre as mais populares atualmente temos: *JavaCC, SableCC* e *ANTLR*.

Entre os geradores de *parsers* o que merece uma atenção especial é o *ANTLR* [Parr 2007], devido a sua flexibilidade na geração de compiladores em diferentes linguagens, tais como, *C#, Java* e *Python*, além de ampla documentação e uma ferramenta de desenvolvimento integrado, chamada de *ANTLRWorks* [ANTLWORKS 2008]. *ANTLRWorks* é uma ferramenta visual escrita em *Java* que facilita o desenvolvimento da gramática *ANTLR*. Ela possui um ambiente com editor de gramática com um interpretador para protótipo rápido e um *debugger*. Além disto, possui visualizador gráfico das dependências das regras da gramática, diagrama de sintaxe dos *tokens*, autômato determinístico finito dos *tokens*, checagem da gramática, entre outros.

Segundo [Parr 2007], os principais benefícios do gerador de *parser ANTLR são:*

- Gera código de leitura humana que é facilmente desdobrado para outras aplicações.

- Gera um poderoso reconhecedor de descendentes recursivos usando *LL*(*), uma extensão para *LL(k)* que utiliza um tomador de decisões arbitrárias de *lookahead*. *Lookahead* é um algoritmo que olha a frente na árvore sintática ou tokens para tomar decisões. A extensão *LL(k)* é um analisador sintático baseado em *LL* que analisa *k tokens* a frente para tomar decisões. O analisador *LL* é essencialmente um analisador sintático descendente que tenta deduzir as produções a partir do nó raiz, ou seja, de cima para baixo.
- Bem integrado com o *StringTemplate*, que é um modelo de motor especificamente desenhado para gerar texto estruturado, tal como, código fonte.
- Tem um ambiente de desenvolvimento de gramática gráfico, chamado de *ANTLRWorks,* como discutido anteriormente.
- É ativamente suportado por um *website* do projeto e por grupos de email com muito tráfego.
- O código é completamente aberto com licença *BSD*.
- Flexível e automatiza muitas tarefas básicas.
- Suporta muitas linguagens, tais como: *Java, C#, Python, Ruby, Objective-C, C, e C++*.

A gramática do *ANTLR* é muito similar em conceito a sintaxe do *EBNF*. Isto deve-se principalmente ao fato do *ANTLR* ser baseado no *EBNF*. Entretanto, existem algumas diferenças na notação que necessitam ser alteradas na gramática escrita em *EBNF* para funcionar no *ANTLR*

- O caractere '–' deve ser substituído pelo caractere '_' no nome das regras.
- Alterar os colchetes ('[…]') da produção por '(...)?'.
- O sinal '=' entre o nome da regra e a definição deve ser substituído por ':'.
- Os operadores de cardinalidade '*' e '+' devem ser colocados ao final do *token* e não no início.
- Comentários especificados com ';' devem ser substituídos por '//'.
- Uma regra de produção em *ANTLR* termina com o caractere ';'.
- A declaração de faixa de valores '-' deve ser substituída por '..'.

Adicionalmente as alterações descritas anteriormente o *ANTLR* suporta ainda, a inserção de código da linguagem destino na definição da gramática. Este código pode ser utilizado para realizar a análise semântica do *parser* ou ainda para instanciar e acionar o *framework StringTemplate.*

StringTemplate é um motor de modelo *Java*, portável para *C#* e *Python*, utilizado para gerar código fonte, páginas *web*, *emails*, ou qualquer outra saída de texto formatada [StringTemplate 2008]. Modelos são *strings* ou documentos com entradas que podem ser preenchidas com expressões que são uma função dos atributos. Os atributos são parâmetros passados para o modelo. Do ponto de vista do *ANTLR*, os parâmetros do modelo são preenchidos durante a execução do *parser*. Sendo preenchidos os parâmetros, o *StringTemplate* esta apto a gerar o arquivo de saída automaticamente com os dados preenchidos pelo *ANTLR*.

O processo de alteração do modelo do *StringTemplate* não exige a regeneração do *parser*. Isto ocorre porque o *StringTemplate* é baseado em um modelo no formato texto e portanto não compilado. A facilidade em alterar o modelo, sem alterar a gramática ou o *parser*, viabiliza gerar saídas do compilador com diferentes visões da árvore sintática do *ANTLR.* Por exemplo, seria possível gerar ao mesmo tempo sem alteração do *parser*, um *XML* como resultado da compilação e um arquivo contendo instruções de montagem para o *Jasmin*[7] gerar *bytecodes Java*.

Outro aspecto importante a ser observado no *ANTLR* é a sua vocação para o desenvolvimento de *DSL* (*Domain-Specific Languages*) [Parr 2007]. Segundo [Kelly 2008], *DSL* é uma especificação ou linguagem de programação dedicada a um domínio de problema em particular, uma representação técnica de um determinado problema ou somente uma solução técnica. Ou seja, *DSL* são linguagens com um objetivo específico que visam atender um domínio de aplicação definido.

O principal benefício da *DSL* é permitir expressar em um alto de nível de abstração a solução de um domínio de problema. Isto facilita o desenvolvimento de soluções pelos especialistas no domínio da linguagem, visto que, estes terão mais facilidade para entender, validar e modificar aplicações escritas com *DSL.* As *DSLs* também tendem a ser mais portáveis, reusáveis e produtivas, sendo também em grande parte, auto-explicáveis e auto-documentadas. Conforme [Kelly 2008], geralmente estas características, as tornam simples, leve e de fácil edição, não necessitando assim de *software* de autoria para desenvolver com elas.

7 *JASMIN - http://jasmin.sourceforge.net/*

2.7 Plataformas de Desenvolvimento para Dispositivos Móveis

Atualmente existe uma variedade de plataformas alvo e ambientes para desenvolvimento de aplicativos para dispositivos móveis. A escolha da plataforma está relacionada ao tipo de dispositivo e recursos que deseja utilizar na aplicação desenvolvida. Toda plataforma possui um sistema operacional e *hardware* específico para seu funcionamento. As principais plataformas disponíveis no mercado são: *.Net Compact Framework*, para dispositivos com sistema operacional *Windows Mobile; Java Micro Edition*, para aparelhos que suportem a máquina virtual da *Sun*; e o *Android*, para dispositivos construídos sob esta arquitetura. Com o objetivo de entender melhor o funcionamento e os recursos existentes nas principais plataformas disponíveis no mercado, em seguida estudaremos em detalhes cada uma destas soluções.

2.7.1 *.Net Compact Framework*

Conforme [MSDN 2008], *.Net Compact Framework* é a plataforma da *Microsoft* de referência para desenvolvimento em dispositivos móveis. Esta plataforma possui suporte a *WS* e implementa os principais recursos da plataforma *.Net Framework* do *desktop.* Ela roda sobre os sistemas operacionais *Windows Mobile* e *Windows Embedded CE.*

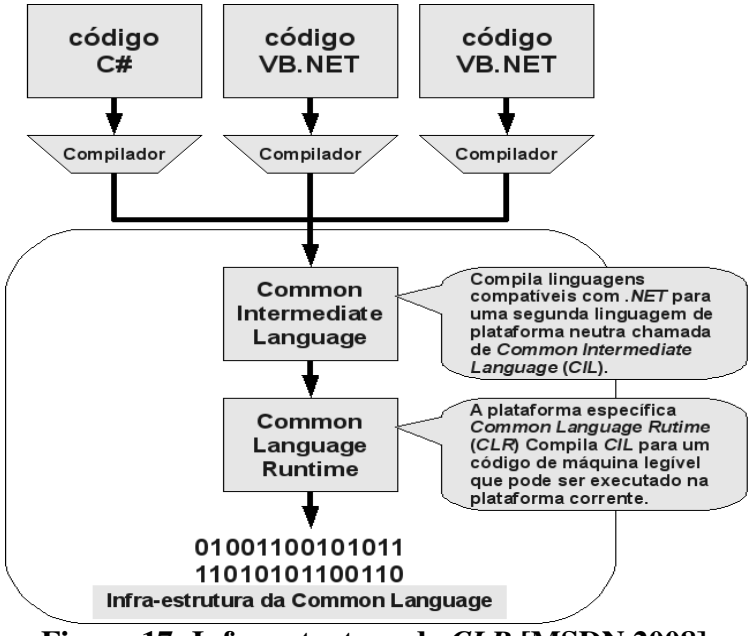

Figura 17: Infra-estrutura da *CLR* [MSDN 2008].

O .NET Compact Framework suporta as linguagens *Visual Basic* e *Visual C#* que roda sobre uma máquina virtual chamada de *Common Language Runtime (CLR)* com um *framework* chamado de *Framework Class Library* (*FCL*). O *CLR* interpreta a linguagem intermediária gerada pelo compilador do *.NET Compact Framework*, chamada de *Common Intermediate Language* (*CIL*), e executa este código com instruções nativas do próprio dispositivo móvel conforme ilustrado na figura 17.

A plataforma *.NET Compact Framework* possui também um ambiente de desenvolvimento integrado chamado de *Visual Studio .Net*. Este ambiente possui um conjunto de ferramentas que suporta todo o ciclo de desenvolvimento de aplicativos para dispositivos móveis. Isto inclui um emulador e *debugador* embutidos no produto que permitem uma simulação de um aplicativo em um ambiente virtual equivalente ao dispositivo móvel real.

2.7.2 Java Micro Edition (JME)

Segundo [SUN 2008], a plataforma *Java Micro Edition* é um arquitetura para desenvolvimento de aplicações para dispositivos móveis. Ela é uma combinação de tecnologias e especificações que combinadas formam um ambiente para construção e execução de aplicativos. As especificações são agrupadas em pacotes baseados em três elementos:

- Configuração: é um conjunto de bibliotecas básicas e uma máquina virtual. A configuração para pequenos dispositivos móveis é chamada de *CLDC (Connected Limited Device Configuration)* e a configuração dispositivos maiores, tais como, *smartphones*, é chamada de *CDC (Connected Device Configuration)*;
- Perfil: é um conjunto de *API*'s que suportam uma quantidade menor de dispositivos chamados de *MIDP* (*Mobile Information Device Profile*);
- Opcional: é um pacote com um conjunto de *API*'s para uma tecnologia específica;

A plataforma *JME* possui uma máquina virtual chamada *JVM* (*Java Virtual Machine*). A *JVM* interpreta uma representação intermediária de código binário, chamada de *bytecodes*, e executa este código com instruções nativas do dispositivo móvel hospedeiro. O *bytecode* é gerado a partir da compilação do fonte da linguagem *Java*. Na figura 18 é apresentada detalhadamente a arquitetura da plataforma *Java* com os componentes explicados anteriormente.

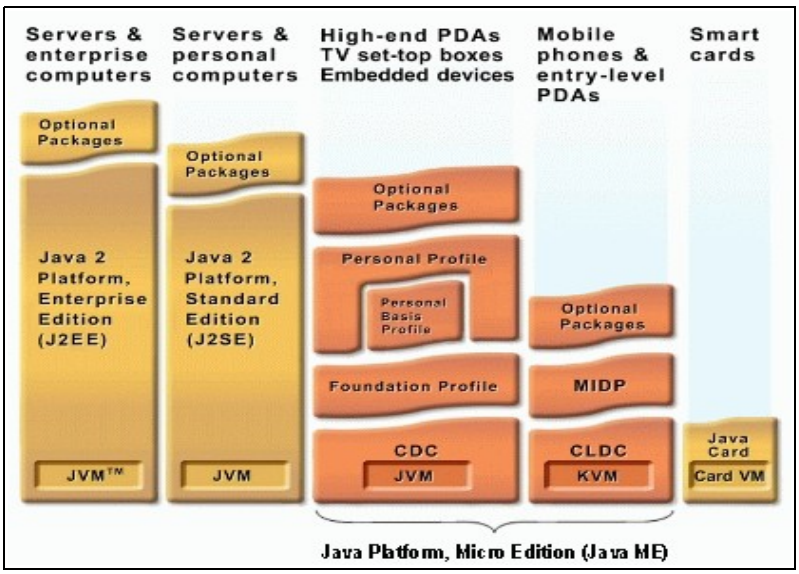

Figura 18: Plataforma *Java* [SUN 2008].

JME também possui suporte a *WS* e um sob-conjunto de bibliotecas para processar *XML*. Existe ainda, a possibilidade utilizar bibliotecas externas ao pacote *JME*, que dão suporte a *WS* otimizado para dispositivos móveis, como exemplos: *kXML* [Haustein 2005] e *kSOAP*[8].

A arquitetura *JME* é suportada por vários de ambientes de desenvolvimento integrados: *Netbeans*, *Eclipse, IntelliJ IDEA* e *Elixir*, dentre outros. Geralmente estes ambientes precisam de *plugins* para funcionar corretamente com o *JME*. Estes *plugins* trazem consigo emuladores e facilidades para depurar, além de uma completa integração com as bibliotecas do *JME*.

2.7.3 Android

Android é um pilha de *softwares* para dispositivos móveis que inclui um sistema operacional, *middleware* e aplicações chaves [Android 2008]. O *Android* é resultado de uma aliança de 34 empresas de tecnologia e telecomunicações, chamada de *Open Handset Alliance*[9], que uniu esforços para desenvolver uma plataforma aberta e pública para suportar desenvolvimentos de aplicativos para dispositivos móveis.

Como pode-se perceber na figura 19 a pilha de *software* do *Android* é composta do sistema operacional *Linux*, bibliotecas, ambiente de execução *Android*, *framework* de aplicações e algumas aplicações exemplo.

8 *KSOAP - http://ksoap.org/*
9 *Open Hand Alliance - http://www.openhandsetalliance.com/*

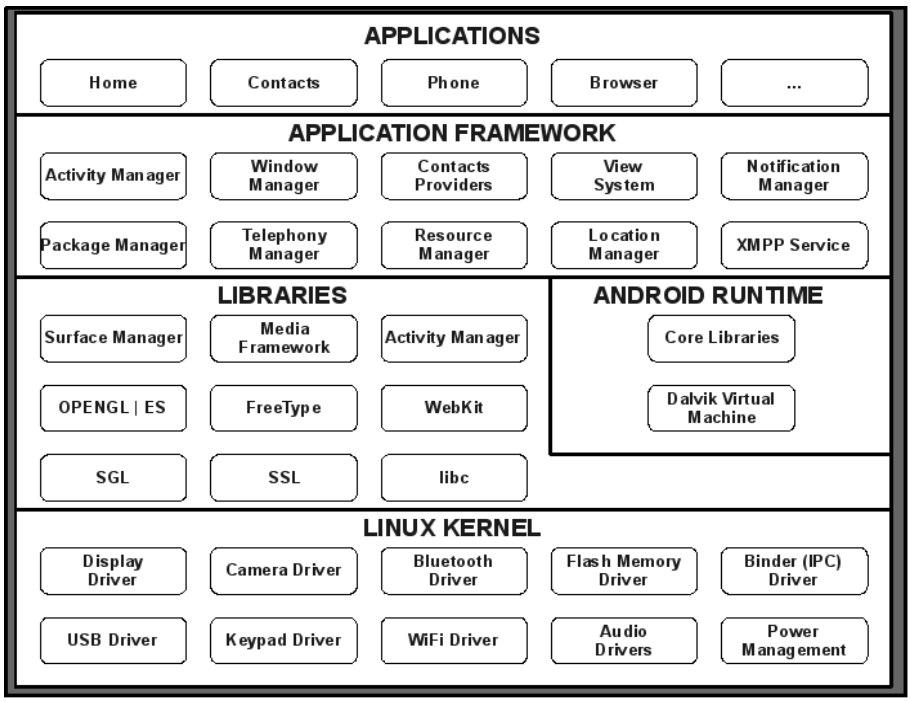

Figura 19: Arquitetura *Android* [Android 2008].

O *Android* possui uma máquina virtual chamada de *Dalvik Virtual Machine*[10] *(DVM)*. A *DVM* interpreta código intermediário, chamado de *bytecode*, e executa este código com instruções nativas do dispositivo móvel hospedeiro. O *bytecode* é gerado a partir da transformação do *bytecode Java* compilado previamente com o compilador da linguagem *Java*. Este processo de compilação *Java* e transformação para *bytecode Dalvik* é facilitado pelo uso de ambientes integrados de desenvolvimento. Atualmente, os ambientes de desenvolvimento *Eclipse*[11]*, Netbeans*[12] *e IntelliJ IDEA*[13]*,* possuem suporte total para a plataforma *Android*, permitindo emular e depurar as aplicações desenvolvidas para esta arquitetura.

Tendo em vista que o *Android* é programado em *Java* e que as principais bibliotecas da especificação *Java* foram migradas para esta plataforma, quase todos os aplicativos escritos em *Java* existentes no mercado podem ser adaptados para esta arquitetura. Como exemplo, temos as bibliotecas *kXML* e *kSOAP*, que podem ser utilizadas no *Android* e que preenchem a falta de suporte a *WS* nativa da plataforma.

10 *DVM - http://www.dalvikvm.com/*
11 *Eclipse - http://www.eclipse.org/*
12 *Netbeans - http://www.netbeans.org/*
13 *IntelliJ IDEA - http://www.jetbrains.com/idea/*

3 TRABALHOS RELACIONADOS

3.1 *CBPEL* – Linguagem para definição de processos de negócios interorganizacionais

[Teófilo 2005] apresenta uma extensão à linguagem *BPEL* chamada de *CBPEL*, que significa *Common Business Process Execution Language*. Esta extensão tem por objetivo modelar processos de negócios inter-organizacionais não previstos nas linguagens de composição ou orquestração de serviços, tais como, *BPEL*, *WSCI*, *XLANG* e *BPML*. Os processos interorganizacionais são aqueles que envolvem mais de uma empresa e tem por principal característica a coreografia de processos. A coreografia de processos, diferente da orquestração de processos, ocorre quando inexiste um participante com o controle absoluto de todo o processo, ou seja, o controle dos processos encontra-se distribuído entre os participantes.

Para o desenvolvimento deste trabalho foram avaliadas duas formas de descrever os processos interorganizacionais. Uma delas consiste no somatório das descrições individuais dos processos de todos os participantes. Essa forma exige que os processos sejam concebidos em paralelo e que haja perfeita correlação nos aspectos de fluxo de dados e controle. A outra forma avaliada leva em conta que as interações sejam complementares entre os parceiros envolvidos e que o fluxo de controle possa ser dividido entre todos os participantes. Nesta forma os processos inter-organizacionais podem ser descritos por um único fluxo global. Neste fluxo global as atividades de interação indicam o participante emissor e o receptor. Estes serão decompostos nos fluxos correspondentes a cada participante. A esta forma deu-se o nome de forma global unificada de processos interorganizacionais.

A forma global unificada é uma forma de descrever processos inter-organizacionais de modo mais sintético. Esta forma apoiá-se nos seguintes pontos:

- As interações envolvem uma emissão e sua respectiva recepção, que podem ser representadas numa única atividade de emissão-recepção.
- Os fluxos de controle existentes nos vários participantes, apesar de poderem ser diferentes, eles devem partilhar de padrões comuns para que o processo global seja isento de erros. Por exemplo, um envio de três mensagens diferentes em sequência,

requer a recepção de três mensagens, que pode ser em paralelo, em sequência, ou por uma ordem qualquer.

Juntando as interações emissão-recepção num fluxo de controle único e global, envolvendo todos os participantes, tem-se a descrição do processo interorganizacional descrito de forma global unificada. Para exemplificar este processo, foi apresentado no trabalho, um exemplo de processo interorganizacional envolvendo três e quatro participantes. Como podemos ver na figura 20 do lado esquerdo estão descritos os participantes de forma discreta, ou seja, cada participante tem um processo, e do lado direito são descritos pela forma global unificada. Neste exemplo utiliza-se a notação gráfica das redes *petri* para descrever o processo. Cada círculo representa um lugar, que é um contentor de unidades, e cada quadrado/ retângulo representa uma transição que ao disparar: retira uma unidade de cada lugar de entrada; deposita uma unidade em cada lugar de saída; e executa uma ação associada (se houver).

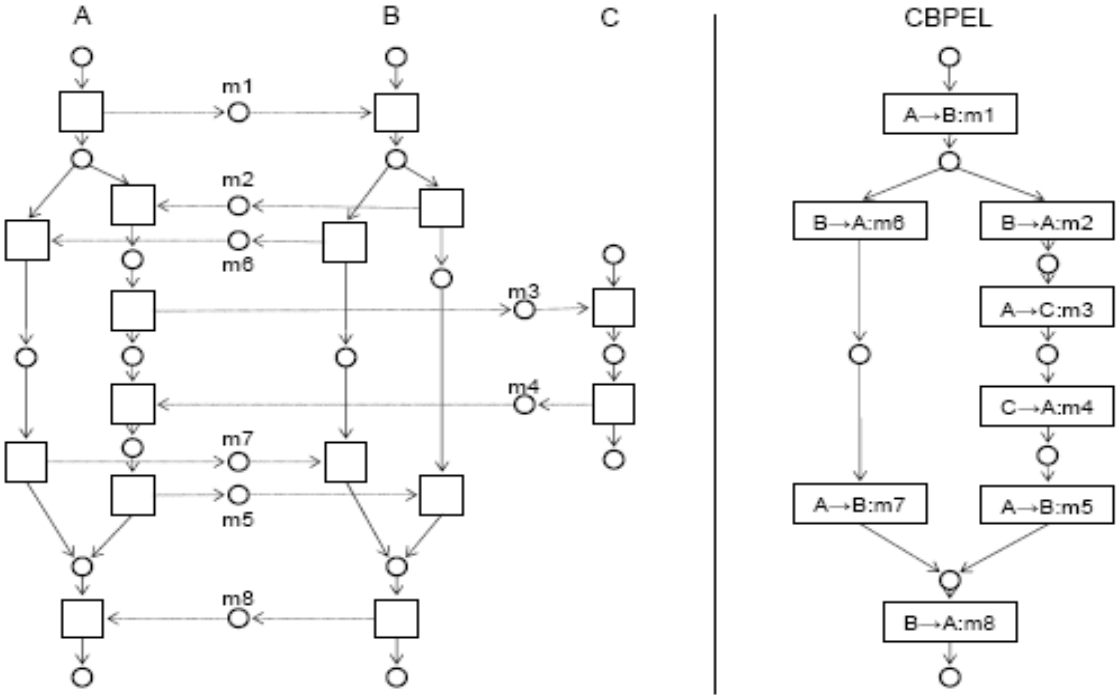

Figura 20: Exemplo de forma global unificada utilizando uma decisão (*switch*) [Teófilo 2005].

A decomposição do processo descrito de forma global unificada em processos de vários participantes tem os seguintes passos: 1) replicar o processo global para cada participante; 2) no processo de cada participante: a) substituir todas as atividades em que o próprio não participa de atividades inertes, ou seja que não têm quaisquer efeitos; b) substituir

todas as atividades de interação por atividades de emissão ou recepção, consoante ao seu papel; c) substituir as primitivas de decisão, no fluxo de controle, em que o próprio não seja o participante ativo na tomada de decisão, por decisões tardias em função das mensagens que são recebidas; d) eliminar as atividades inertes e simplificar os fluxos de controle.

No esforço de suportar o processo global unificado descrito anteriormente, foram implementados elementos na linguagem *CBPEL* que definem regras e comportamentos do processo. Estes elementos foram organizados em dois grupos: dados e parceiros; e interações e controle de fluxo. Abaixo, para melhor entendimento, serão brevemente apresentados cada um dos grupos e seus elementos:

a) **Dados e parceiros.** Neste grupo foram abordados os elementos *partnerLinkType*, *partnerLink, partners*, *variables*, *correlation sets* e *assign*:
   - ***partnerLinkType* e *partnerLinks*:** diferentemente da definição inicial do *BPEL* que prevê um papel para cada *portType,* obedecendo as definições *WSDL*, a *CBPEL* habilita a declaração de dois papéis por *partnerLink*.
   - ***partners:*** o *CBPEL* define os papéis globais e suas participações nos *partnerLinks* com o uso do elemento *partners.*
   - ***variable(s), assign, copy e correlationSets*:** para estes elementos não houve alteração no *CBPEL* em relação a *BPEL*;
   - ***process*:** o nome do elemento foi alterado para *commonProcess* para deixar clara a mudança de domínio.

b) **Interações e controle de fluxo.** Neste grupo foram abordados os elementos de interações (*invoke, receive, reply)* e de controle de fluxo (*wait, empty, sequence, switch, while, pick,* e *flow*)
   - **interações:** a *CBPEL* diferente da *BPEL* apresenta uma visão global comum do processo, por isto a iteração passou a ser modelada por um único elemento chamado de *send*, que representa um *invoke* num parceiro e um *receive* noutro.
   - **controle fluxo:** os elementos de controle de fluxo permaneceram iguais a *BPEL* na linguagem *CBPEL*.

A linguagem *CBPEL* desenvolvida neste trabalho teve um foco maior na descrição dos dados e parceiros, do que nas interações e controle de fluxo do *BPEL*. O maior desafio encontrou-se em relacionar, distribuir e controlar os processos inter-organizacionais. As interações e controle de fluxo, como percebeu-se, já encontram-se bem resolvidos na linguagem *BPEL*, não exigindo assim, maiores cuidados.

3.2 *PICCOLA - a Small Composition Language*

A linguagem *PICCOLA* [Archemann 2001] foi desenvolvida em *Java* com o objetivo de viabilizar a composição de componentes de *software* entre plataformas e linguagens distintas. Isto foi realizado com a separação da implementação e da composição dos componentes. O paradigma chave deste projeto preconiza que toda a aplicação é composta de componentes que são agrupados e organizados por *scripts*. Os componentes são independentes e focados em um domínio específico, enquanto os *scripts* possibilitam a organização e a comunicação entre os componentes.

O principal argumento deste trabalho é que a flexibilidade e a adaptabilidade necessária para aplicações baseadas em componentes para cobrir mudanças de requisitos e reaproveitamento de serviços, comuns em projetos de desenvolvimento de *software*, podem ser substancialmente elevadas, se pensarmos em arquiteturas, *scripts*, coordenação e colas, além dos componentes. Entretanto, para que isto seja factível, é necessário deixar claro a separação entre os elementos computacionais e seus relacionamentos.

Procurando criar um modelo de composição de componentes que priorizasse os relacionamentos entre os elementos de *software*, foram exploradas duas abordagens de desenvolvimento. A primeira baseada em um estilo de programação imperativa e a segunda em um estilo funcional e declarativo. Como resultado deste trabalho investigativo, a linguagem *PICCOLA*, aproveitou do paradigma funcional o conceito de expressões formais com funções de alta ordem e do imperativo manteve o controle de estados e tratamento de *I/O (Input/Output)*. Ou seja, obteve-se uma linguagem híbrida, bastante influenciada por *Python* e *Haskell*, apesar de ter sido desenvolvida na linguagem *Java*.

A linguagem *PICCOLA* foi idealizada para dar suporte a alguns conceitos chaves das linguagens de composição de componentes, como descritos abaixo e apresentados na figura 21:

- **Componentes:** um componente é uma caixa-preta que fornece e utiliza serviços. Estes serviços podem ser conectáveis. O valor adicionado dos componentes encontra-se no fato destes serem padronizados. Um componente que não possui compatibilidade para ser conectado com outro componente não tem muita utilidade.
- **Arquiteturas:** componentes são por definição elementos de um *framekwork* de componentes. Eles aderem a uma arquitetura particular de componentes ou a um

estilo de arquitetura que define os conectores, e as regras de composição correspondentes. Um conector é o mecanismo utilizado para colocar componentes juntos.

- ***Scripts*:** um *script* define como os componentes são conectados juntos. Pense que um *script* diz aos atores como eles devem executar os vários papéis em uma peça de teatro. A essência das linguagens de *script* é a configuração dos componentes, possivelmente definidas fora da linguagem. Uma linguagem de *script* costumeiramente irá dizer como tratar o *script* de um componente.
- **Coordenação:** se os componentes são agentes em um ambiente distribuído, então fala-se da coordenação particular de um *script*. A linguagem de coordenação é focada no gerenciamento das dependências entre componentes concorrentes e distribuídos.
- **Cola (*glue*):** Embora seja reivindicado que os componentes devam ser construídos para serem compostos, o que frequentemente ocorre é que evita-se utilizar componentes que não tenham conectadores compatíveis com o que se esta desenvolvendo. Nestas situações somente os códigos cola (*glue)* podem adaptar conectadores de componentes para fazer o que realmente desejamos. Adaptadores cola não são simplesmente interfaces, mas podem ser adaptadores de contratos cliente/servidor ou plataformas ponto de dependências.

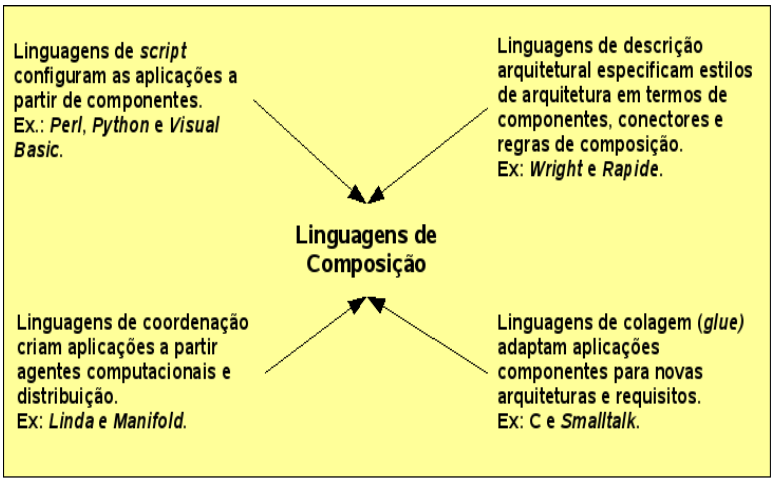

Figura 21: Conceitos chaves das linguagens de composição de componentes. Adaptado de [Archemann 2001].

Os conceitos discutidos anteriormente foram essenciais para a delimitação do trabalho de desenvolvimento da linguagem de composição de componentes *PICCOLA*. Elas são os pilares da linguagem.

Do ponto de vista de implementação, a linguagem *PICCOLA* consiste de alguns componentes fundamentais:

- ***ident = e*:** amarra o formulário de expressão *e* ao nome *ident*.
- ***export ident = e*:** estende o contexto corrente com a amarração *ident = e*. A extensão é feita por uma atualização funcional. Então, uma expressão *export ident = e, global*, onde *global* denota ao contexto corrente, é equivalente a *global = (global, ident = e).*
- ***e.ident*:** poduz o valor do rótulo *ident* amarrado a expressão da fórmula *e.*
- ***def ident(ident1) … (identN) = e*:** define uma abstração parametrizada sobre a fórmula da expressão *e*. Mais precisamente, esta construção é utilizada para definir um função *ident* com os parâmetros formais *ident1...identN*. Funções são valores de classes primários.
- ***return e*:** retorna a expressão *e*. No geral, este termo é usado para especificar um retorno antecipado.
- ***ident(e1)...(eN) [in eM]*:** chama uma função *ident* com o argumento *e1, ..., en* sincronizados. Se o *eM* é especificado, a função é invocada usando *eM* como o contexto atual, de outra forma o contexto atual é usado.
- ***operador infix*:** operadores, como +, -, | e > são atalhos sintáticos que denotam funções; eles são codificados com rótulos de __+, __-, __| e __> que mapeiam as operações correspondentes. Por exemplo, a expressão *e1 | e2*, lê-se *e1 pipe e2*, denota a chamada da função *pipe* de dentro do contexto *e1* usando *e2* como argumento.

Aos componentes descritos anteriormente, foi inserido também, um *gateway Java* com uma interface padrão e alguns serviços básicos de *I/O*. Os serviços *I/O* foram desenvolvidos para permitir a carga e o transporte dos *scripts PICCOLA* entre os componentes distribuídos.

3.3 *CLAM - Composition Language for Autonomous Megamodules*

[Sample 2000] apresenta uma linguagem de composição de serviços chamada de *CLAM (Composition Language for Autonomous Megamodules)*. *CLAM* é uma linguagem focada em composição de serviços assíncrona de larga escala com módulos autônomos. Ela é uma linguagem compilada procedural que gera código executável baseada em uma linguagem chamada de *CHAIMS*. Entre outras características esta linguagem foi desenvolvida para suportar otimizações tanto em tempo de compilação quanto em tempo de execução.

A linguagem *CLAM* foi desenhada para compor módulos ou serviços em larga escala, chamados de mega-módulos. Um programa cliente composto de vários mega-módulos é chamado de mega-programa. Frequentemente mega-módulos são escritos em diferentes linguagens de programação, residem em máquinas distintas, e são integrados para dentro de diferentes sistemas distribuídos. Alguns sistemas utilizando *CLAM* necessitam de pontes para diferente arquiteturas, linguagens e sistemas distribuídos. Esta linguagem esta fortemente focada em coordenação e composição para atingir seus objetivos. No item de coordenação foi implementada a sincronização, ordenação e *timing*. Do item de composição a principal preocupação deu-se em combinar as partes do serviços em um todo.

As principais características implementadas na linguagem *CLAM* foram:

a) **paralelismo:** a linguagem de composição *CLAM* pode ser utilizada para agendar e organizar mega-módulos. Os mega-módulos podem ser utilizados para operar em paralelo. A linguagem de mega-programação coordena estes mega-módulos de foram assíncrona. Para tornar isto possível a linguagem *CLAM* é subdividida em várias primitivas para melhor controlar o tempo e a execução dos módulos:
   - ***SETPARAM*:** é utilizado para atribuir parâmetros para um método de um mega-módulo em particular que tenha sido estabelecido em um *SETUP*.
   - ***GETPARAM*:** pode retornar o valor, tipo, e nome da descrição de um parâmetro de qualquer método de um mega-módulo em particular.
   - ***INVOKE*:** inicia a execução de um método específico com o parâmetros definidos em *SETPARAM*.
   - ***EXTRACT*:** coleta os resultados de uma invocação.
   - ***TERMINATE*:** finaliza uma invocação em execução ou uma conexão de um mega-programa de um mega-módulo em específico.

b) **otimização:** existem quatro otimizações possíveis com *CLAM* em tempo de compilação e execução.
   - ***ESTIMATE*:** esta diretiva calcula o tempo estimado de um serviço baseado na taxa , no tempo e o volume de dados da invocação. Esta informação é utilizada pelo mega-programa agendar a invocação da melhor forma possível.
   - ***EXAMINE*:** é utilizado para determinar o estado da invocação. Ele retorna o estado enumerado do mega-programa. Ele pode ser *DONE, PARTIAL, NOT_DONE* e *ERROR*. Esta informação pode ser utilizada pelo módulo de transporte para sincronizar o mega-programa depois da seção de um código paralelo.

- **Extrações progressivas:** o *CLAM* pode mudar a estratégia de execução durante o processamento de um serviço utilizando técnicas de agendamento especulativo.
- **Ajuste de parâmetros:** o mega-programa também pode decidir dinamicamente checar a performance de vários mega-módulos para otimizar os parâmetros.

c) **controle de fluxo:** *CLAM* implementou alguns controles de fluxo para viabilizar o fluxo das composições.

- *IF e WHILE:* estas definições são utilizadas para executar serviços de forma condicional.
- ***INVEX*:** executa igual ao *INVOKE* com a exceção de que a aplicação não retorna até que os resultados sejam coletados da invocação.
- *EXWDONE:* espera até que todos os resultados desejados de uma invocação em particular esteja disponível e os resultado sejam coletados. *INVEX* e *EXWDONE* geralmente trabalham em conjunto.

d) **repositório:** a linguagem *CLAM* implementou um repositório similar ao *CORBA* que entre outras informações armazena os mapeamentos de nomes entre os mega-programas e os dados de localização, protocolo e métodos dos mega-módulos.

e) **tipos de dados:** foram implementados seis tipos de dados: *opaque* (tipo complexo), *integer*, *string*, *boolean*, *real* e *datetime*.

Este trabalho mostrou uma linguagem de composição de serviços com recursos de otimização de execução e com amplas características de controle de fluxo com paralelismo.

4 *inSOA* (*invoke Service Oriented Architecture*)

Este capítulo descreve os passos da implementação do compilador da linguagem *inSOA*. Serão discutidas as características, a gramática, a estrutura, a análise e o projeto da linguagem. Os elementos e a forma de uso da linguagem também serão abordados. Será apresentada ainda a questão da integração com o motor de composição *SmallSOA* e as possíveis utilidades da *inSOA*, assim como, sua inserção no projeto *U-SOA*[14].

4.1 Projeto *U-SOA*

A linguagem *inSOA (invoke Service Oriented Architecture),* chamada anteriormente de *SQLSOA (Structured reQuest Language for Service Oriented Architecture)*, faz parte de um projeto chamado de *U-SOA* [Zanuz, Barcelos et al. 2008A]. *U-SOA* é um projeto de *framework* para construção de sistemas colaborativos ubíquos baseados em arquiteturas orientadas a serviço. Este *framework* prevê também estruturas capazes de fornecer mapas das composições de serviços e dos respectivos serviços utilizados nessas composições, possibilitando assim, que diversas análises de impacto e de *QoS (Quality of Service)* possam ser construídas a partir destes mapas.

U-SOA (*Ubiquitous Service-Oriented Architecture*), conforme ilustra a figura 22, pode ser visualizado como uma pilha de tecnologias separadas em camadas. As camadas envolvidas na arquitetura incluem desde níveis mais baixos, onde se encontra a arquitetura *Android*, até o topo da pilha, onde se encontram as aplicações colaborativas ubíquas suportadas pela arquitetura.

14 *U-SOA - http://sourceforge.net/projects/usoa/*

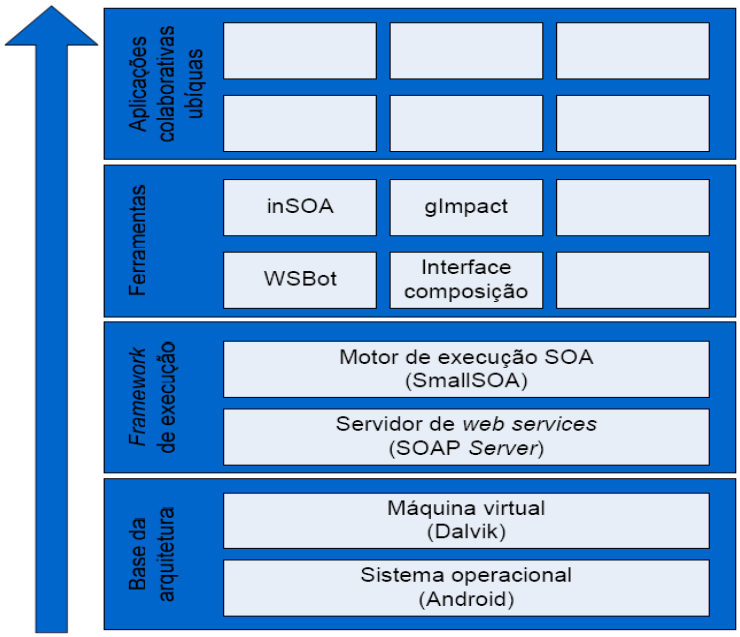

Figura 22: Arquitetura *U-SOA*.

Atualmente, os seguintes trabalhos já foram ou estão sendo desenvolvidos.

- Servidor de *WS*, baseado no *SOAPServer*[15].
- Motor de composição de serviços *SmallSOA* [Zanuz, Barcelos et al. 2008B].
- Linguagem de composição de serviços *inSOA,* anteriormente chamada de *SLQSOA*, objeto deste trabalho.
- Ferramenta para análise de impacto na composição de serviços, chamada de *gImpact*.
- Robô para localização e catalogação de serviços *WSBot*.
- Interface gráfica para composição de serviços em dispositivos móveis baseada na linguagem *inSOA*.

No mais alto nível da arquitetura U-SOA se encontram as aplicações colaborativas ubíquas que poderão ser criadas a partir dos componentes desenvolvidos nas camadas inferiores.

4.1.1 *SmallSOA*

SmallSOA é um motor para execução de composições de serviços (*web services*) em ambientes ubíquos. O motor se caracteriza por oferecer condições para a execução de composições de serviços em ambientes móveis e distribuídos, conforme ilustra a figura 23.

15 *SOAPServer - http://sourceforge.net/projects/jmesoapserver/*

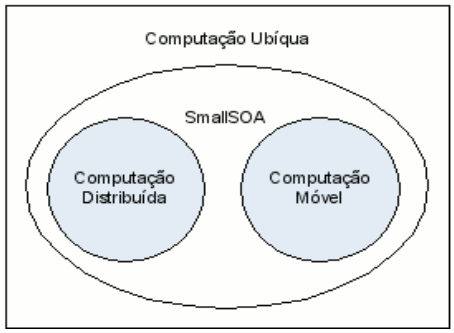

Figura 23: Contexto atual de SmallSOA.

Um motor para execução de composições de serviços é um componente de software que, dada uma linguagem de composição de serviços, interpreta uma descrição de uma composição escrita nessa linguagem e processa-a, normalmente retornando algo para um usuário cliente, conforme ilustra a figura 24. A descrição da composição processada pelo motor é escrita através da linguagem *inSOA*.

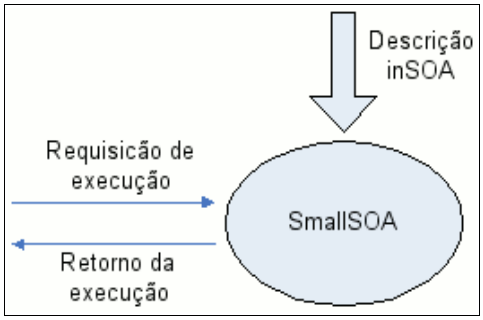

Figura 24: Interpretação de linguagem de descrição de serviços.

O motor *SmallSOA* deve, portanto, ser capaz de interpretar uma descrição de composição de serviços escritos através da linguagem *inSOA*. Os principais aspectos descritos pela linguagem e que são interpretados pelo motor são:

- Chamadas a serviços e a outras composições de serviços;
- Manipulação dos retornos dos serviços e dos retornos ao usuário cliente;
- Manipulação de variáveis;
- Tratamento de exceções.

Além da sua função de interpretador ou executor da linguagem de composição de serviços *inSOA*, *SmallSOA* possui outras duas funções adicionais relacionadas e que dão apoio à principal:

- Publicação das composições de serviços;
- Disponibilização das composições de serviços.

4.1.2 *gImpact*

gImpact é uma ferramenta de análise de impacto para arquitetura *SOA* em dispositivos móveis. Esta ferramenta tem por objetivo reduzir o custo de execução dos serviços, fornecendo informações sobre o impacto na utilização destes, e também prover recursos para analisar a qualidade do serviços, com vistas para aumentar o nível de disponibilidade dos serviços.

Uma das principais tarefas do *gImpact* é a análise de impacto sobre os serviços. Para realizar esta tarefa o *gImpact* constrói uma árvore de composição como ilustrado na figura 25. Esta árvore tem todos os seus nós e relações visitados pelo *gImpact*, onde são identificados os tempos, a disponibilidade e a composição dos serviços.

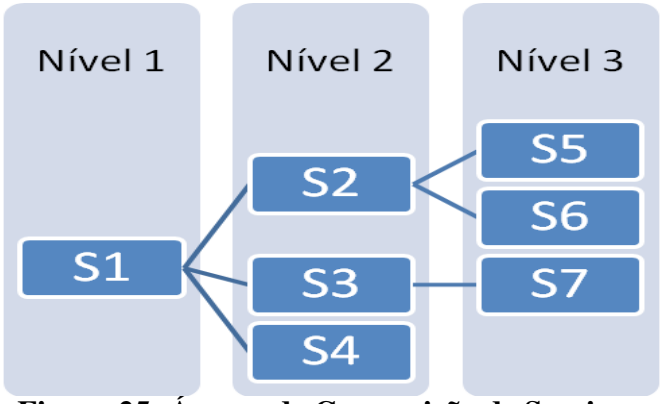

Figura 25: Árvore de Composição de Serviços.

Para quantificar a análise de impacto, o *gImpact* possui uma fórmula de cálculo como apresentado na figura 26. A fórmula visa medir a complexidade da composição, onde quanto mais nós forem visitados e mais distante forem estes nós, mais complexa é a composição.

$$I(S) = \sum_{S_i \in S} (D(S_i) + L(S_i) + ((1 - Q(S_i)) * P))$$

Figura 26: Fórmula do *gImpact*.

Abaixo a descrição de cada variável:

- ***I*:** corresponde ao impacto;
- ***S*:** aos serviços;
- ***D*:** distância que o serviço se encontra na árvore do serviço;
- ***L*:** localização do serviço, deve ser 0 caso se encontre no mesmo servidor;
- ***Q*:** indicando o percentual da qualidade da conexão com o serviço;

- ***P*:** indica o percentual de importância que possui a qualidade para o usuário, este parâmetro é configurado na ferramenta.

O nível de impacto calculo pelo *gImpact* é de vital importância para a determinação da complexidade e do impacto na alteração de composições. Ele também é importante no desenvolvimento de uma nova composição e na determinação do nível de qualidade da composição.

4.2 Especificação e Definição da Linguagem

inSOA é uma linguagem declarativa de composição de serviços com foco em dispositivos móveis. Ela faz composição de serviços chamando e compondo serviços da arquitetura *SOA* utilizando o protocolo *SOAP* (*WS-**). O protótipo da linguagem roda inteiramente em dispositivo móvel e possui integração com o motor de composição *SmallSOA*. Assim como o *SmallSOA*, a linguagem *inSOA*, está inserida em um projeto maior que tem como objetivo disponibilizar uma meta-arquitetura para dar suporte a ambientes colaborativos em dispositivos móveis.

4.3 Características

A linguagem *inSOA* possui um série de caraterísticas que a tornam uma linguagem de composição de serviços bastante flexível. Estas características são:

a) **Insensível a capitalização:** todas as declarações da linguagem são insensíveis a capitalização, evitando assim, erros decorrentes de interpretação da digitação dos comandos pelo usuário;
b) **Declaração flexível:** com exceção do comando principal *invoke,* o restante dos comandos podem ser declarados em qualquer posição;
c) **Validação de padrões:** os endereços de *URI* e *XPath* requisitados em alguns comandos da *inSOA* são validados pelas suas especificações e gramáticas padrões, respectivamente, *URI RFC* 3986[16] e *XPath* 2.0 [W3C 2003]. Isto ocorre, porque a linguagem trata estes elementos como comandos e não como *strings*.
d) **Tratamento *XML*:** o tratamento do *XML* de retorno do *WS* requisitado é via *XPath* e encontra-se embutido internamente na linguagem.
e) **Opcionalidade:** com exceção das chamadas dos *WS* o restante dos comandos da linguagem são opcionais. Isto faz com que um *script inSOA* seja do tamanho da

16 *URI RFC 3986* - http://www.ietf.org/rfc/rfc3986.txt

necessidade do usuário, não sendo necessário inserir comandos que não são necessários para a solução programada.

f) **Tratamento de erros:** todos os erros de sintaxe e semântica são tratados em tempo de compilação e execução, sendo mostrados de forma clara e precisa, identificando o local exato do erro e com sugestões de solução quando possível.

g) **Orquestração:** esta característica diz respeito a habilidade da *inSOA* em orquestrar as chamadas dos serviços. Estas chamadas podem ocorrer de forma paralela dependendo do nível de dependência entre os serviços. Os retornos dos *WS* também podem ser manipulados e restringidos, em acordo com o código do *script* desenvolvido.

h) **Saída customizável:** a saída da linguagem pode ser customizada simplesmente alterando um *script StringTemplate*. O compilador gera em tempo de execução uma árvore de objetos de todos os comandos interpretados. Esta árvore pode ser utilizada para gerar uma saída, tal como, a interface *XML* de integração com o motor *SmallSOA*, o qual foi implementado no protótipo do compilador *inSOA*. Outras possibilidades com esta caraterística, seriam a geração automática de *bytecodes* ou impressão em qualquer formato e padrão da árvore sintática abstrata da interpretação.

i) **Invocação otimizada:** a ordem em que os *WS* devem ser chamados no processo de orquestração é otimizada pelo compilador. O usuário pode declarar as chamadas dos *WS* em qualquer ordem, entretanto o compilador *inSOA* irá reordenar estas chamadas da forma mais otimizada possível, considerando as dependências entre os *WS* e a quantidade de parâmetros requisitados.

j) **Embutível:** é possível embutir a linguagem *inSOA* em outras linguagens de propósito geral, da mesma forma que ocorre hoje com a linguagem *SQL*.

k) **Leve:** não é necessário *software* de autoria para desenvolver um *script inSOA*. A não existência de códigos extensos e rótulos de marcação na *inSOA*, tipo o *XML*, a torna leve, viabilizando a criação de composições manualmente utilizando qualquer editor de texto de propósito geral.

l) **Abordagem:** linguagem com uma abordagem declarativa de domínio específico para declaração de composição de serviços.

m) **Baseada:** a edição de um *script inSOA* dá-se de forma textual, portanto ela é baseada em texto.

Como pode-se verificar na listagem das características acima a linguagem *inSOA* tenta ser bastante flexível, segura e customizável, sem contudo perder seu foco principal em compor serviços.

Tabela 9: Comparação das características das linguagens de composição.

#	Característica	*BPEL*	*OWL-S*	*inSOA*
a	**Insensível a capitalização**	NÃO	NÃO	SIM
b	**Declaração Flexível**	NÃO	NÃO	SIM
c	**Validação de Padrões**	NÃO	NÃO	SIM
d	**Tratamento *XML***	Externo	Externo	Interno
e	**Opcionalidade**	NÃO	NÃO	SIM
f	**Tratamento de Erros**	SIM	SIM	SIM
g	**Orquestração**	SIM	SIM	SIM
h	**Saída Customizável**	NÃO	NÃO	SIM
i	**Invocação Otimizada**	NÃO	NÃO	SIM
j	**Embutida**	NÃO	NÃO	SIM
k	**Leve**	NÃO	NÃO	SIM
l	**Abordagem**	Funcional	Orientada para Objetos	DSL
m	**Baseada**	XML	XML	Texto

A tabela 9 apresenta um comparativo das características dos dois principais paradigmas de linguagens de composição e a linguagem *inSOA*. Como pode-se perceber a linguagem *inSOA* possui vantagens em praticamente todos os itens comparados. Um item a ser destacado é a sua abordagem *DSL* e a sua estruturada baseada em texto, que a tornam mais leve e simples de desenvolver, podendo inclusive ser embutidas em linguagens de propósito geral devido a estas características. Isto traz benefícios na criação da composição e também na distribuição da composição entre diferentes plataformas de *software*.

4.4 Estrutura e gramática

A gramática da linguagem *inSOA* foi desenvolvida utilizando a ferramenta *ANTLRWorks* [ANTLWORKS 2008]. *AntlrWorks* é uma ferramenta visual escrita em *Java* que facilita o desenvolvimento da gramática *ANTLR* [Parr 2007]. Ele possui um ambiente editor de gramática com um interpretador para protótipo rápido e um *debugger* para isolar erros de gramática. O *ANTLR,* como discutido anteriormente, é uma ferramenta que fornece um *framework* para construir reconhecedores, interpretadores, compiladores, e tradutores de descrição de gramáticas. Ele segue o conceito da gramática *BNF,* mas possui sintaxe e formato diferente deste.

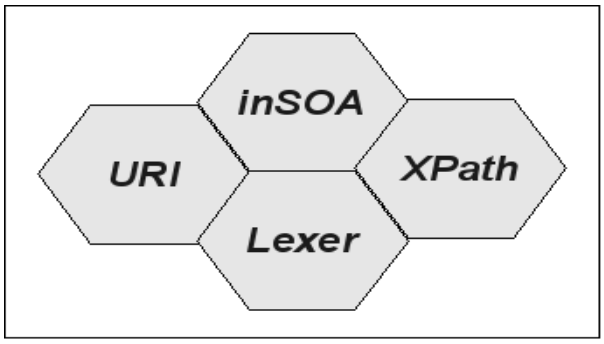

Figura 27: Pacotes de gramática da linguagem *inSOA*.

Com o objetivo de melhor estruturar a gramática *inSOA* e também evitar o limite de 64k por classe, imposto pela linguagem *Java*, o projeto foi dividido em quatro pacotes como apresentado na figura 27. O pacote principal contendo a gramática da linguagem *inSOA* utiliza os outros três pacotes: *URI*, *XPath* e *Lexer*. Cada um desses, representa, respectivamente, o endereço *URI* e *Namespace*, o caminho *XPath* e os *tokens* léxicos.

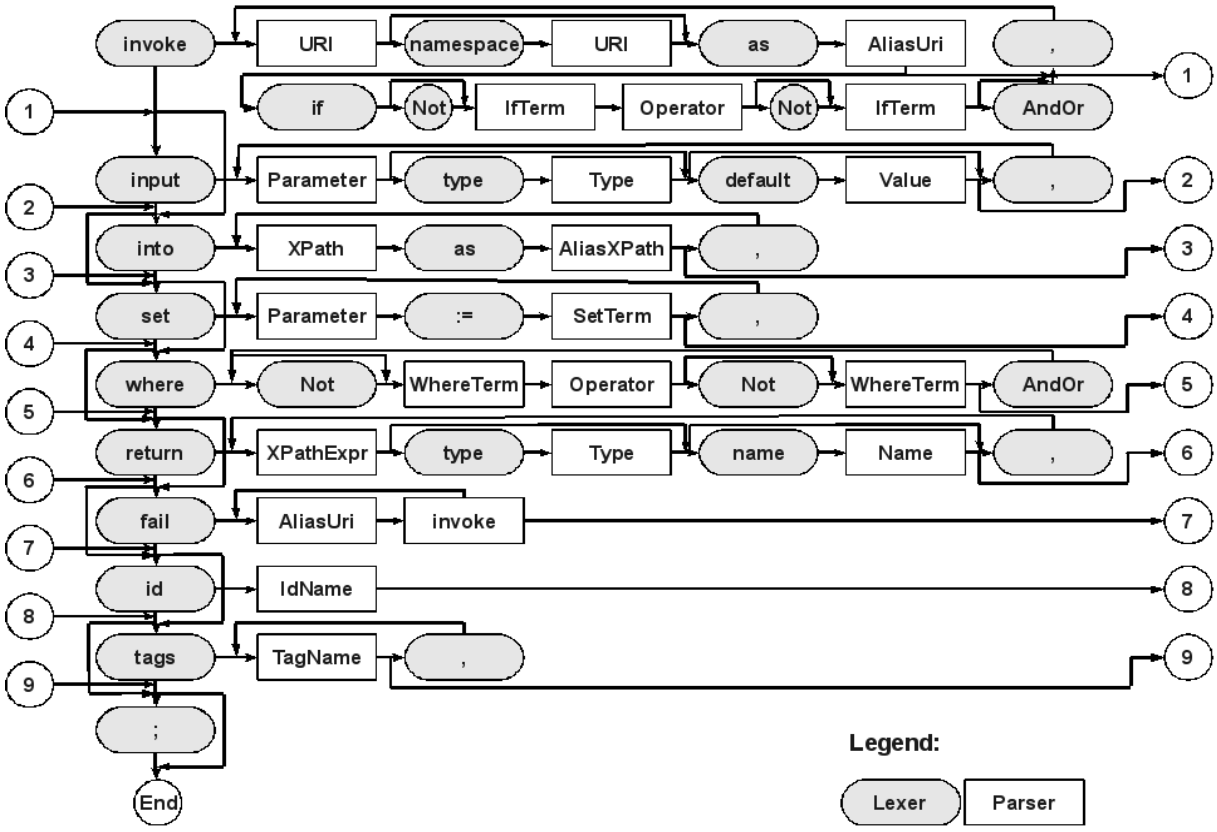

Figura 28: Diagrama da sintaxe da linguagem *inSOA*.

A linguagem *inSOA* é representada por um código em formato de *script*. Este *script* está dividido essencialmente em um comando obrigatório e outros oito opcionais. O comando

obrigatório é o *invoke,* enquanto os outros oito são: *input, into, set, where, return, fail, id e tags*.

Na figura 28 é apresentado o diagrama da sintaxe da linguagem *inSOA*. Nas elipses encontram-se os elementos léxicos ou comandos da linguagem, enquanto nos retângulos são demonstrados os elementos de *parser* da *inSOA*. As setas pontilhadas do diagrama representam a opcionalidade entre os elementos e comandos da linguagem. Como pode-se perceber alguns comandos, tais como, o *invoke*, após o caractere vírgula, podem ser opcionalmente recursivos.

Como apresentando anteriormente, um código *inSOA* é chamado de *script* e representa uma composição única. Este *script* possui oito comandos, sendo a maioria deles opcionais, e com exceção do *invoke,* o restante dos comandos pode ser declarado em qualquer ordem. Cada comando possui um conjunto de elementos ou subcomandos que definem o funcionamento deste. Todo comando ou subcomando possui associado a ele um valor. Este valor pode ser um literal ou uma expressão. As expressões, tais como, *XPath* e *URI*, são validadas pelas suas gramáticas respectivas, enquanto os literais são apenas classificados entre numéricos e *string.* Quando os literais são classificados como numéricos, a precisão e os aspectos decimais são validados. Os *scripts inSOA* também permitem inserir comentários. Os comentários em uma linha são precedidos dos caracteres --, enquanto os comentários de múltiplas linhas devem começar dos caracteres /** e finalizados por */.

A gramática simplificada da linguagem *inSOA,* utilizando a especificação *ANTLR,* é apresentada no anexo I. No entanto, em seguida serão discutidos cada um dos comandos da linguagem em detalhe.

4.5 Comandos, subcomandos e elementos

A linguagem *inSOA* possui oito comandos. Cada comando possui uma série de subcomandos ou elementos com seus valores associados. Existem algumas dependências entre alguns comandos, assim como, instanciação de variáveis que podem ser utilizadas entre os diferentes elementos da linguagem. Estas e outras características da *inSOA* serão apresentadas abaixo:

a) ***Invoke*:** é o comando principal da *inSOA*. Ele é o único comando mandatório da linguagem. Neste comando, são declarados os *WS* que devem ser invocados e o

controle de fluxo separados por vírgula. Os elementos deste comando estão detalhados na tabela 10:

Tabela 10: Subcomandos e elementos do comando *Invoke*.

Item	Obri-gatório	Descrição
Declaração	Sim	*Invoke URI.Operation [namespace NameSpaceURI] as AliasURI [if [Not] IfTerm Operator [Not] IfTerm [AndOr [Not] IfTerm Operator [Not] IfTerm]*] [, URI.Operation [namespace NameSpaceURI] as AliasURI [if [Not] IfTerm Operator [Not] IfTerm [AndOr [Not] IfTerm Operator [Not] IfTerm]*]]*.* Um *invoke* contém uma ou mais *URIs com suas operações* separadas por vírgula, onde cada *URI* pode ter uma especificação de *namespace* e obrigatoriamente um alias.
URI. Operation	Sim	É o endereço do *WS* (*endpoint*) e a declaração da operação do *WS* separada por um ponto. A *URI s*egue o padrão *URI RFC 3986*.
namespace	Não	É o subcomando que define o namespace do *WS*. Utilizado quando o servidor *SOAP* onde esta hospedado o *WS* exigir *namespace*.
NameSpace URI	Não	É o valor do subcomando *namespace*. Ele esta associado ao subcomando *namespace* e também segue o padrão *RFC 3986*.
as	Sim	Define o alias da *URI* declarada.
AliasUri	Sim	É o valor do subcomando *as*. O *alias* é o nome que identifica o endereço *URI* e sua operação, além do *NameSpaceURI*. Ele deve começar com caracter e aceita números e letras na sua composição.
if	Não	O subcomando *if* é utilizado para inserir condições a execução do *WS* declarado. Estas condições de execução são utilizadas para implementar controles de fluxo, tais como, os desvios por valores de contexto.
Not	Não	Subcomando de negação, ou seja, tudo que vier após este subcomando será negado.

Item	Obri-gatório	Descrição
IfTerm	Sim	Aqui é declarado o termo de validação da execução do *WS*. O valor pode ser um literal ou uma variável. O valor literal aceita números naturais, ponto flutuante ou exponenciais, além de *strings* entre aspas simples ou duplas. Um valor variável pode ser um *AliasXPath* com caminho ou um dos parâmetros de entrada. O valor de *AliasXPath* com caminho segue o formato: *AliasXPath XPath,* onde o *AliasXPath* é aquele declarado no comando *Into* e o *XPath* é o caminho onde encontra-se o valor desejado do *Into*. O valor do parâmetro de entrada é aquele declarado no comando *Input*. Este elemento suporta expressões também. As expressões podem utilizar os operadores +, -, *, / e ||, respectivamente, adição, substração, multiplicação, divisão e concatenação. Quando utilizados, os operadores devem ser separados por espaços.
Operator	Sim	Neste elemento são declarados os operadores relacionais e de restrição deste comando. Estas operadores são: ==, *!=*, <, <=, > *e* >=, que significam respectivamente, igual, diferente, menor, menor ou igual, maior e maior ou igual.
AndOr	Não	Este é um subcomando *booleano* utilizado na avaliação dos termos. Ele aceita os subcomandos *and* e *or,* onde *and* é conjunção entre as expressões e *or* é disjunção.

b) ***Input*:** este comando define os parâmetros de entrada da composição. Este parâmetros podem ser utilizados nas expressões dos outros comandos a serem estudados. Além do nome do parâmetro são declarados o tipo e o valor padrão do mesmo, como descritos na tabela 11.

Tabela 11: Subcomandos e elementos do comando *Input*.

Item	Obri-gatório	Descrição
Declaração	Não	*Input Parameter [type Type] [default Value] [, Input Parameter [type Type] [default Value]]**. Um *input* contém um ou mais parâmetros separados por vírgula, onde em cada parâmetro pode-se definir o tipo e valor padrão do parâmetro de entrada.
Parameter	Sim	É o nome do parâmetro de entrada.
type	Não	É o subcomando que define o tipo do parâmetro.
Type	Não	É o valor do subcomando *type*. O tipo de parâmetro pode ser: *string, number* ou *base64*. Os tipos complexos devem ser classificados como *base64*.
default	Não	É o subcomando que define o valor padrão do parâmetro da

		composição.
Value	Não	É o valor do subcomando *default*. Este valor deve ser compatível com o tipo definido no subcomando *type*.

c) ***Into*****:** é um comando que deve ser utilizado quando faz-se necessário a transformação dos retornos das operações dos *WS,* ou ainda, quando deseja-se utilizar os retornos dos *WS* em outros comandos da *inSOA*. A transformação ocorre com as declarações *XPath* do comando, enquanto a utilização em outros comandos dá-se pelo alias definido na declaração do subcomando *as* do comando *Into*. A quantidade de alias neste comando deve ser igual ao número de *WS* declarados no comando *invoke*. Na tabela 12 os elementos deste comando são apresentados em detalhe.

Tabela 12: Subcomandos e elementos do comando *Into*.

Item	Obri-gatório	Descrição
Declaração	Não	*Into XPath as AliasXPath [, XPath as AliasXPath]**. Um *into* contém uma ou mais expressões separadas por vírgula, onde cada expressão deve ter obrigatoriamente o *XPath* correspondente e o alias deste.
XPath	Sim	É uma expressão *XPath* versão 2.0 que seleciona e trata os dados que irão retornar do *WS* chamado no comando *invoke*.
as	Sim	É o subcomando que define o alias do *XPath*. Este *alias* declara o nome da variável que deverá armazenar o resultado da interpretação do *XPath*.
AliasXPath	Sim	É o valor do subcomando *as*. Ele representa o alias que poderá ser utilizado no restante dos comandos.

d) ***Set*****:** este comando declara a atribuição de valores ou expressões para os parâmetros dos *WS* declarados no comando *invoke*. Entre os outras possibilidades, os valores ou expressões podem se utilizar dos alias declarados no comando *into* e dos parâmetros definidos no *input*, conforme descrição detalhada apresentada na tabela 13.

Tabela 13: Subcomandos e elementos do comando *Set*.

Item	Obri-gatório	Descrição
Declaração	Não	*Set Parameter := SetTerm [, Parameter := SetTerm]**. Um *Set* declara as expressões ou valores que devem ser atribuídos aos parâmetros dos *WS*.
Parameter	Sim	Este elemento representa um parâmetro. Um parâmetro tem o seguinte formato: *AliasUri.FieldName*, onde o *AliasUri* é o alias declarado no *invoke* e o *FieldName* é o nome do parâmetro da operação do *WS*.

Item	Obri-gatório	Descrição
:=	Sim	É o subcomando de atribuição de valores ou expressões para o *Parameter*.
SetTerm	Sim	Aqui é declarado o valor do subcomando de atribuição *:=*. Da mesma forma que o subcomando *ifTerm* discutido anteriormente no comando *Invoke*, o subcomando *SetTerm* suporta literais ou variáveis. Aplica-se aqui também as expressões e seus respectivos operadores.

e) ***Where*****:** o comando *Where* declara as projeções relacionais dos conjuntos de dados e restrições que devem ser aplicadas sobre os resultados retornados e tratados no comando *Into*. As projeções tem por objetivo relacionar os resultados do *Into* e as restrições são utilizadas para limitar os dados retornados, criando um novo subconjunto destes, conforme detalhados na tabela 14.

Tabela 14: Subcomandos e elementos do comando *Where*.

Item	Obri-gatório	Descrição
Declaração	Não	*Where [Not] WhereTerm Operator [Not] WhereTerm [AndOr Where [Not] WhereTerm Expression [Not] WhereTerm]**. O *Where* declara termos do comando, restringidos por operadores e, relacionados pelas projeções *AndOr.* Em seguida cada um dos elementos deste comando serão explicados.
Not	Não	Subcomando de negação, ou seja, tudo que vier após este subcomando será negado.
WhereTerm	Sim	Da mesma forma que os subcomandos *IfTerm* e *SetTerm* discutidos anteriormente nos comandos *Invoke e Set*, o subcomando *WhereTerm* suporta literais ou variáveis. As expressões com seus respectivos operadores, descritos nos subcomandos *IfTerm* e *SetTerm,* são aplicáveis aqui também.
Operator	Sim	Este elemento funciona da mesma forma que os subcomandos *IfTerm* e *SetTerm*.
AndOr	Não	Este é um subcomando *booleano* que dá suporte a projeção dos conjuntos. Ele aceita os subcomandos *and* e *or,* onde *and* é conjunção entre as expressões e *or* é disjunção.

f) ***Return*****:** este comando define os elementos que devem ser retornados após a execução da composição *inSOA*. Se este comando não for declarado, será retornado apenas um indicativo de que a composição foi executada com ou sem sucesso. Na tabela 15 os subcomandos deste comando são detalhados.

Tabela 15: Subcomandos e elementos do comando *Return*.

Item	Obri-gatório	Descrição
Declaração	Não	*Return XpathExpr [type Type] [name Name] [, XpathExpr [type Type] [name Name]]**. Um *Return* contém uma ou mais expressões *XPath* separadas por vírgula, onde em cada expressão pode-se definir o tipo e o nome do parâmetro de retorno.
Value	Sim	Aqui é declarado o valor de retorno da composição. Da mesma forma que o subcomando *IfTerm*, *WhereTerm* e *SetTerm* dos comandos *Invoke*, *Where e Set,* o valor pode ser ou literal ou uma variável, além de conter expressões com operadores matemáticos e de concatenção.
name	Não	Este subcomando define um nome para o valor que será retornado da composição. Este nome é útil para o cliente da composição saber o nome do valor que será devolvido.
Type	Não	É o valor do sub-comando *type*. O tipo de parâmetro pode ser: *string, number* ou *base64*. Os tipos complexos devem ser classificados como *base64*.
name	Não	Este subcomando define um nome para o valor que será retornado da composição. Este nome é útil para o cliente da composicao saber o nome do valor que será devolvido.
Name	Não	Este elemento contém o valor do subcomando *name*.

g) ***Fail*:** este comando declara o tratamento para as falhas ocorridas durante a execução dos *WS* do comando *invoke*. Este tratamento de falhas intercepta o *WS* gerador do erro e chama um novo *invoke* para compensar esta falha. Este novo *invoke* aceita os mesmos comandos que o *invoke* principal suporta, além de poder se utilizar dos alias e parâmetros de entrada declarados no *invoke* principal. Na tabela 16 são detalhados os elementos do comando *Fail*.

Tabela 16: Subcomandos e elementos do comando *Fail*.

Item	Obri-gatório	Descrição
Declaração	Não	*Fail [AliasUri invoke]+*. Um *Fail* contém uma ou mais expressões de tratamento de falhas. Em cada uma das expressões define-se os alias das *URIs e* os *invokes* que irão interceptar as falhas.
AliasUri	Sim	O elemento *AliasUri* é declarado como: alias *:*. No elemento *alias* são declarados os alias que deseja-se interceptar separados por vírgula. Estes *alias* são os mesmos definidos no comando *invoke.* O caracter *:* é necessário para separar a declaração dos *alias* interceptados e o comando *invoke* que

Item	Obri-gatório	Descrição
		irá compensar a falha. O mesmo *alias* não pode ser declarado em mais de um comando de tratamento de falha. Além disso, pode-se utilizar o alias reservado *other*. Este *alias* trata qualquer falha, inpendente do *WS* que tenha gerado o erro.
invoke	Sim	Neste elemento deve-se declarar o *invoke* que irá tratar a falha interceptada no subcomando *AliasUri*. Este *invoke* suporta todos os comandos do *invoke* principal, podendo inclusive utilizar os alias e parâmetros de entrada declarados neste.

h) ***Id*:** o comando *Id* declara a identificação da composição. É aceito para este comando qualquer nome que inicie por um caractere e seja seguido por qualquer quantidade de caracteres ou números. Abaixo, na tabela 17, é descrito os elementos deste comando.

Tabela 17: Subcomandos e elementos do comando *Id*.

Item	Obri-gatório	Descrição
Declaração	Não	*Id idName*. Um *Id* contém um e somente uma descrição. Esta descrição representa o nome de identificação da composição *inSOA*.
id	Sim	Comando que declara a identificação da composição.
idName	Sim	Este elemento contém o valor do comando *id*.

i) ***Tags*:** neste comando são declaradas as categorias e rótulos da composição *inSOA*. Uma categoria ou rótulo serve para classificar as composições e facilitar a pesquisa destas pelos clientes das composições.

Tabela 18: Subcomandos e elementos do comando *Tags*.

Item	Obri-gatório	Descrição
Declaração	Não	*Tags TagName [, TagName]**. O comando *Tags* contém uma ou muitas categorias separadas por vírgula. Estas categorias devem iniciar por carateres.
tags	Sim	Comando que declara as classificações e rótulos da composição *inSOA*.
TagName	Sim	Este elemento contém os valores do comando *tags*.

O conjunto de comandos, subcomandos e elementos *inSOA* visam atender as funcionalidades de composição de serviços requisitadas pelos *patterns* de composição. Estes *patterns* de composição serão discutidos na próxima seção.

4.6 *Patterns* de composição de serviços

Existem alguns *patterns* de composição de serviços, como discutidos no capítulo 2.5, que possibilitam a classificação e determinação das capacidades de uma ferramenta ou linguagem de composição. Tendo isto em vista, foram desenvolvidos *scripts inSOA* correspondentes aos vinte e um principais *patterns* de composição com o objetivo de averiguar a aderência da linguagem a estes padrões. Estes *scripts* e sua breve descrição são apresentados a seguir:

1) ***Sequence*:** os *WS* são chamados na sequência em que são declarados. No exemplo abaixo o *WS1 e WS2* são chamados na sequência em que foram definidos.

```
Ex. 1:  invoke http://usoa.insoa/WS1.Operation as a,
                http://usoa.insoa/WS2.Operation as b;
```

2) ***Parallel Split*:** os *WS* são divididos em processos paralelos concorrentes. No exemplo abaixo o *WS1 e WS3* serão divididos e executados em paralelo, visto que, não possuem nenhuma dependência entre si, além de seus parâmetros estarem resolvidos pelos literais atribuídos a eles. O *WS2* será executado após o término do *WS1*, pois possui dependência em relação a este.

```
Ex. 2:  invoke http://usoa.insoa/WS1.Operation as a,
                http://usoa.insoa/WS2.Operation as b,
                http://usoa.insoa/WS3.Operation as c
          into / as ResA, / as ResB, / as ResC
           set a.field := 'Test',
               b.field := ResA/result/text(),
               c.field := 'Test1';
```

3) ***Synchronization*:** os *WS* são divididos em processos paralelos concorrentes e podem convergir para um processo dependente destes. No exemplo abaixo, os *WS1* e *WS3* irão convergir para o *WS2* após finalizarem.

```
Ex. 3:  invoke http://usoa.insoa/WS1.Operation as a,
                http://usoa.insoa/WS2.Operation as b,
                http://usoa.insoa/WS3.Operation as c
          into / as ResA, / as ResB, / as ResC
           set a.field := 'Test',
               b.fielda := ResA/result/text(),
               b.fieldb := ResC/result/text(),
               c.field := 'Test1';
```

4) ***Exclusive Choice*:** uma composição decide o caminho a seguir dependendo de uma condição. No exemplo abaixo, os *WS1* e *WS2* precisam atender suas condições *if* para

serem executados. No caso apresentado a escolha é exclusiva, ou seja, ou é executado o *WS2* ou o *WS3*.

```
Ex. 4:  invoke http://usoa.insoa/WS1.Operation as a,
                http://usoa.insoa/WS2.Operation as b if resA/tag/text() == 'Execute',
                http://usoa.insoa/WS3.Operation as c if resA/tag/text() != 'Execute'
        into / as ResA, / as ResB, / as ResC
          set a.field := 'Test',
              b.field := ResA/result/text(),
              c.field := ResA/result/text();
```

5) ***Simple Merge*:** ele é similar ao *pattern Synchronization*. Entretanto, diferente do *Synchronization*, o *Simple Merge* exige que os *WS* sejam agrupados ou façam parte de um outro *WS.* No exemplo abaixo ocorre um agrupamento implícito das composições no *WS3*, que agrupa *o WS2* e o *WS4*.

```
Ex. 5:  invoke http://usoa.insoa/WS1.Operation as a,
                http://usoa.insoa/WS2.Operation as b,
                http://usoa.insoa/WS3.Operation as c,
                http://usoa.insoa/WS4.Operation as d
        into / as ResA, / as ResB, / as ResC, / as ResD
          set a.field := 'Test',
              b.fielda := ResA/result/text(),
              b.fieldb := ResC/result/text(),
              c.field := 'Test1',
              d.field := ResC/result/text();
```

6) ***Multi-Choice*:** o comando *if* do *inSOA* é uma forma conveniente de habilitar que uma chamada de *WS* seja realizada atendendo múltiplas restrições e consequentemente escolhas. O exemplo abaixo apresenta um caso típico de *multi-choice.*

```
Ex. 6:  invoke http://usoa.insoa/WS1.Operation as a,
                http://usoa.insoa/WS2.Operation as b
                     if resA/text() == 'Execute' or resA/result/text() == 'Run',
                http://usoa.insoa/WS3.Operation as c if resA/result/text() != 'Execute'
        into / as ResA, / as ResB, / as ResC
          set a.field := 'Test',
              b.field := ResA/result/text(),
              c.field := ResA/result/text();
```

7) ***Synchronizing Merge*:** o *Synchronizing Merge* é um *pattern* que é implementado no *inSOA* a partir do encadeamento de *WS* utilizando o comando *if,* como demonstrado no próximo exemplo.

```
Ex. 7:  invoke http://usoa.insoa/WS1.Operation as a,
                http://usoa.insoa/WS2.Operation as b if resA/tag/text() != 'Run',
```

```
        http://usoa.insoa/WS3.Operation as c if resA/tag/text() != 'Execute',
        http://usoa.insoa/WS4.Operation as d
into / as ResA, / as ResB, / as ResC, / as ResD
 set a.field := 'Test',
     b.field := ResA/result/text(),
     c.field := ResA/result/text(),
     d.fieldA := ResB/result/text(),
     d.fieldB := ResC/result/text();
```

8) ***Multi-merge*:** este *pattern* ocorre quando dois ou mais *WS* convergem para um único *WS*, como demonstrado no exemplo abaixo.

Ex. 8:
```
invoke http://usoa.insoa/WS1.Operation as a,
       http://usoa.insoa/WS2.Operation as b,
       http://usoa.insoa/WS3.Operation as c
into / as ResA, / as ResB, / as ResC
 set a.field := 'Test',
     b.field:= 'Test',
     c.fieldA := resA/result/text(),
     c.fieldB := resB/result/text();
```

9) ***Discriminator*:** o *pattern discriminator* é implementado no *inSOA* quando um retorno *XML* do *WS* é atribuído a uma variável no comando *into* e cada linha do *XML* é avaliada. Este procedimento de avaliação do *XML* ocorre de modo automático no *inSOA*, ou seja, quando o *script inSOA* é executado e encontra um retorno *XML* que possui mais de uma linha, cada linha é avaliada tanto no comando *Set* quanto no *Where*. No exemplo abaixo, se retornar mais de uma linha no *XML* da variável *ResA* que representa o resultado do *WS1*, estas linhas serão avaliadas e servirão de entrada para o campo *field* do *WS2* (b.*field*). Isto fará com que todos os retornos sejam avaliados.

Ex. 9:
```
invoke http://usoa.insoa/WS1.Operation as a,
       http://usoa.insoa/WS2.Operation as b
into / as ResA, / as ResB
 set b.field := ResA/result/text();
```

10) ***Arbitrary Cycles*:** da mesma forma que o *pattern discriminator*, o *pattern arbitrary cycles* é avaliado em tempo de execução, onde os ciclos de entrada e saída são especificados no elemento de atribuição do comando *Set* e as decisões de continuidade pelo subcomando *If* , como demonstrado no exemplo abaixo:

Ex. 10:
```
invoke http://usoa.insoa/WS1.Operation as a,
       http://usoa.insoa/WS2.Operation as b,
       http://usoa.insoa/WS3.Operation as c if ResB/result/text() != 'Stop'
```

```
into / as ResA, / as ResB, / as ResC
  set b.field := ResA/result/text(),
      c.field := ResB/result/text();
```

11) ***Implicit Termination*:** este *pattern* ocorre naturalmente ao término da execução de chamada de *WS*. Ou seja, o término da composição está implícito no término da execução dos *WS* declarados no *script inSOA*. No exemplo abaixo, após a execução da *Operation* do *WS1* a composição *inSOA* irá terminar de forma implícita.

Ex. 11:
```
invoke http://usoa.insoa/WS1.Operation as a;
```

12) ***Multiple Instances Without Synchronization*:** não havendo dependências entre os *WS* declarados na *inSOA,* estes são executados em paralelo e sem sincronismo, desde que seus resultados não sejam utilizados em *WS* subsequentes. Isto significa dizer, que a *inSOA* prevê a execução de múltiplas instâncias sem haver sincronização entre elas da forma requisitada por este *pattern*. No exemplo abaixo o *WS2* e *WS2* irão executar em paralelo sem nenhuma sincronização entre eles, mesmo estes sendo dependentes do *WS1*.

Ex. 12:
```
invoke http://usoa.insoa/WS1.Operation as a,
       http://usoa.insoa/WS2.Operation as b,
       http://usoa.insoa/WS3.Operation as c
 into / as ResA, / as ResB, / as ResC
  set b.field := ResA/result/text(),
      c.field := ResA/result/text();
```

13) ***Multiple Instances With a Priori Design Time Knowledge:***o comando *Set* da *inSOA* pode declarar as dependências entre os *WS*. Sendo assim, sabe-se durante o desenvolvimento da composição as instâncias do processo. Esta característica poderia ser declarada na linguagem utilizando os retornos dos *WS* no *into* ou o comando *if*. No exemplo abaixo o *WS2* somente é executado se *numExec* for maior que zero.

Ex. 13:
```
invoke http://usoa.insoa/WS1.Operation as a,
       http://usoa.insoa/WS2.Operation as b if numExec > 0
 input numExec type number default 3
 into / as ResA, / as ResB
  set b.field := ResA/result/text(),
      numExec := numExec – 1;
```

14) ***Multiple Instances With a Priori Runtime Knowledge:*** devido ao conceito de manipulação de *XML* com *XPath* e a avaliação do retorno de *XML* com múltiplas linhas, este *pattern* é atendido pela linguagem. No exemplo abaixo, em tempo de desenvolvimento, é definido no retorno *ResA* que devem ser considerados apenas as

primeiras 3 (três) linhas do retorno do *WS1*.

```
Ex. 14:  invoke http://usoa.insoa/WS1.Operation as a,
                http://usoa.insoa/WS2.Operation as b
           into //*[position()<4] as ResA, / as ResB
            set b.field := ResA/result/text();
```

15) ***Multiple Instances Without a Priori Runtime Knowledge:*** este *pattern* é atendido na sua totalidade, pelo motivo explicado no item anterior sobre a avaliação *XML* e a manipulação *XPath*. Devido a característica de tratamento interno de *XML* com o *XPath* na linguagem *inSOA*, o desconhecimento do número de instâncias é resolvido de forma implicita durante a execuçao, como demonstrado no exemplo abaixo.

```
Ex. 15:  invoke http://usoa.insoa/WS1.Operation as a,
                http://usoa.insoa/WS2.Operation as b
           into //*[position()<4] as ResA, / as ResB
            set b.field := ResA/result/text();
```

16) ***Deferred Choice:*** o subcomando *if* do comando *invoke* da *inSOA* permite tomar decisões em tempo de execução utilizando as variáveis de retorno. Isto pode ser utilizado para proteger ou adiar uma escolha. No exemplo abaixo a *Operation* do *WS1* será executada se o parâmetro de entrada *choice* for maior que zero, se ele for menor que zero será executada a *Operation* do *WS2*.

```
Ex. 16:  invoke http://usoa.insoa/WS1.Operation as a if choice > 0,
                http://usoa.insoa/WS2.Operation as b if choice < 0
           input choice type number default 1;
```

17) ***Interleaved Parallel:*** a execução de *WS* intercalada, onde um mesmo *WS* utiliza informações de dois ou mais caminhos distintos sendo executada somente uma vez, é possível na *inSOA* utilizando as atribuições do comando *Set,* como demonstrado no exemplo abaixo.

```
Ex. 17:  invoke http://usoa.insoa/WS1.Operation as a,
                http://usoa.insoa/WS11.Operation as b,
                http://usoa.insoa/WS2.Operation as c,
                http://usoa.insoa/WS21.Operation as d,
                http://usoa.insoa/WS3.Operation as e
           into / as Res1, / as Res11, / as Res2, / as Res21, / as Res3
            set a.field := 'Test',
                c.field := 'Test',
                e.fieldA := res1/result/text(),
                e.fieldB := res2/result/text(),
                b.field := res1/result/text(),
                d.field := res2/result/text();
```

18) ***Milestone:*** não existe mecanismo na linguagem *inSOA* para controle de marcos de execução, visto que, .isto exigiria armazenamento de estado de execução. O armazenamento de estado de execução permite que um processo de composição fique inativo até que um novo comando inicie a sua execução. Este *pattern* exige que o cliente de uma composição guarde a sessão do servidor para que seja possível executar partes do processo posteriormente.

19) ***Cancel Activity:*** este *pattern* é implementado diretamente no motor de composição *SmallSOA*, sendo assim, é possível requisitar o cancelamento de uma atividade da composição diretamente para o motor, do mesmo modo que ocorre a execução. No exemplo abaixo, caso a execução da composição esteja em execução ainda, ela pode ser cancelada de forma explicita ao enviar um comando de cancelamento para o motor *SmallSOA.*

Ex. 19: invoke http://usoa.insoa/WS1.Operation as a;

20) ***Cancel Case*****:** mesmo conceito do *pattern* 19. Entretanto, este *pattern* se preocupa com toda a composição, ou seja, no exemplo abaixo quando for solicitado o cancelamento da composição, tanto o *WS1* quanto o *WS2* serão cancelados, porque estão sendo executados em paralelo.

```
Ex. 20:  invoke http://usoa.insoa/WS1.Operation as a,
                http://usoa.insoa/WS2.Operation as b;
```

21) ***Exception Handling*****:** o tratamento e compensação de falhas na *inSOA* é tratado pelo comando *fail.* Este comando, como discutido anteriormente, permite interceptar e tratar um erro em tempo de execução de um dado *WS*. O tratamento pode incluir a chamada de um *WS* para estornar transações. Abaixo é apresentado um exemplo que ilustra a intercepção e a chamada de um *WS* para tratar o erro.

```
Ex. 21:  invoke http://usoa.insoa/WS1.Operation as a,
                http://usoa.insoa/WS2.Operation as b
           into / as ResA, / as ResB
            set a.field := 'Test',
                b.field := 'Test'
         fault .a: invoke http://usoa.insoa/HandlingErrorA.Operation as c
                      set c.field := ResA/result/text()
                   other:
                      invoke http://usoa.insoa/HandlingOthers.Operation as a
                          set c.field := 'SendEmail',
                              c.fielda := ResA/result/text();
```

A linguagem de composição *inSOA* foi concebida para possuir de aderência com os *patterns* de composição estudados. Por esta razão, como podemos verificar na tabela 19, a *inSOA* implementa praticamente todos os *patterns* de composição discutidos anteriormente, com exceção do *pattern* 18.

Na tabela 19 é realizada também uma comparação das capacidades da linguagem *inSOA* com a *BPEL* e a *OWL-S*. Nesta tabela o sinal + significa que a linguagem suporta o *pattern*, o sinal - que não suporta o *pattern* descrito, enquanto o sinal +/- que a linguagem não implementa de forma nativa o *pattern*, mas é possível representá-lo de forma indireta.

Tabela 19: Comparação dos patterns implementados na arquitetura *BPEL4WS, OWL-S e inSOA*.

#	*Pattern*	*BPEL4WS*	*OWL-S*	*inSOA*
1	*Sequence*	+	+	+
2	*Parallel Split*	+	+	+
3	*Synchronization*	+	+	+
4	*Exclusive Choice*	+	+	+
5	*Simple Merge*	+	+	+
6	*Multi-choice*	+	+	+
7	*Synchronizing Merge*	+	+	+
8	*Multi-merge*	-	-	+
9	*Discriminator*	-	-	+
10	*Arbitrary Cycles*	-	-	+
11	*Implicit Termination*	+	+	+
12	*Multiple Instances Without Synchronization*	+	+	+
13	*Multiple Instances With a Priori Design Time Knowledge*	+	+	+
14	*Multiple Instances With a Priori Runtime Knowledge*	-	-	+
15	*Multiple Instances Without a Priori Runtime Knowledge*	-	-	+
16	*Deferred Choice*	+	+	+
17	*Interleaved Parallel*	+/-	+/-	+
18	*Milestone*	-	-	-
19	*Cancel Activity*	+	+	+
20	*Cancel Case*	+	+	+
21	*Exception Handling*	+	-	+

A quantidade de *patterns* de composição catalogados é longa e exaustiva. Por questões de foco optou-se por estudar os principais *patterns*. Por esta razão, algumas características da linguagem *inSOA,* implementadas no comando *where* e no *return,* tais como, a projeção e o

retorno múltiplo de dados, não foram discutidas. Porém, estas e outras características são muito importantes nas linguagens de composição, mesmo não tendo sido abordadas.

4.7 Análise, projeto e desenvolvimento

Nesta seção são apresentados os requisitos, a análise, o projeto e os aspectos de desenvolvimento do compilador *inSOA*. Os requisitos são brevemente descritos, a análise e projeto são apresentados através de modelos *UML* [Hamilton 2006] e no desenvolvimento o protótipo e compilador *inSOA* serão discutidos.

Para o desenvolvimento da ferramenta optou-se por uma arquitetura voltada a dispositivos móveis, pelo fato deste trabalho estar inserido em um projeto cujo objetivo é disponibilizar uma arquitetura ubíqua colaborativa para dispositivos móveis.

4.8 Análise e Requisitos

A ferramenta de protótipo da linguagem *inSOA* possui duas funcionalidades principais onde o usuário pode compilar e gerenciar composições *SOA*. Cada uma destas funcionalidades é subdividida em outros requisitos. O requisito de compilação é dividido em análise léxica, parser e semântica. O requisito de análise semântica é dividido ainda em um requisito de otimização e outro de geração da interface *XML*. Esta interface é utilizada para comunicar com o motor de composição *SmallSOA*. O gerenciamento de composições *SOA* se divide em três outros requisitos. Estes requisitos atendem a necessidade chave de comunicação com o motor de composição *SmallSOA*. São eles: publicar, executar e capturar composição *inSOA*. O requisito de publicação sucede a geração da interface *XML* da qual é dependente e sem o qual não funcionará.

Tendo sido identificado os requisitos e as funcionalidades necessárias para a *inSOA* foi realizada a modelagem utilizando-se o diagrama de casos de uso. A figura 29 apresenta o diagrama de casos de uso do protótipo da solução *inSOA*.

Durante o processo de modelagem foram identificados dois atores:

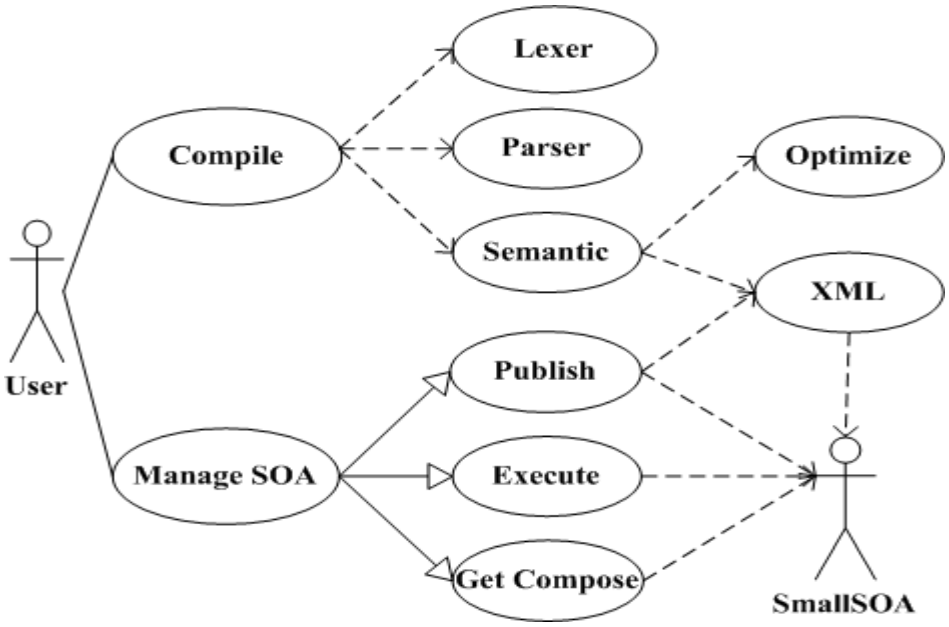

Figura 29: Diagrama de caso de uso da *inSOA*.

- ***User*:** é o usuário da ferramenta *inSOA*.
- ***SmallSOA*:** é o motor de execução da composição *inSOA*, o qual deve estar em todos os dispositivos onde se encontram as composições.

Estes atores requerem alguns requisitos da aplicação para que o objetivo do protótipo *inSOA* seja alcançado. Abaixo são detalhados os requisitos da aplicação seguindo a descrição apresentada na figura 29:

- ***Compile*:** este requisito visa atender a necessidade que um usuário tem de compilar um *script inSOA* e gerar uma interface *XML* otimizada para executar no motor de composição *SmallSOA*. Entretanto, para este requisito ser alcançado são necessários que outros sub-requisitos sejam atendidos. Estes sub-requisitos são inseridos dentro do *compile* e serão detalhados em seguida.
- ***Lexer*:** compreende a análise léxica do *script inSOA*, ou seja, o responsável pela análise dos caracteres do texto do *script* que gera uma sequência de símbolos chamados de *tokens* que podem ser interpretados por um *parser*.
- ***Parser*:** este requisito visa realizar a análise sintática do compilador. Ele transforma os símbolos do texto de entrada analisados pelo *lexer* em uma árvore sintática que representa a gramática da linguagem.
- ***Semantic*:** o requisito *semantic* visa garantir que as características semânticas da linguagem sejam atendidas. Isto inclui, por exemplo, a necessária declaração de variáveis com o comando *Into* quando for necessário manipular os retornos *XML*

dos *WS* e a verificação da igualdade entre o número de *WS* declarados e a quantidade de *XPaths* do *Into*.

- ***Optimize*:** o objetivo deste requisito é dar suporte a necessidade de otimização das chamadas dos *WS,* onde é identificada a ordem das chamadas e o paralelismo destas invocações.
- ***XML Interface*:** o motor de composição *SmallSOA* necessita de uma interface *XML* otimizada para executar uma composição.
- ***Manage SOA*:** este requisito visa atender a necessidade de gerenciar as comunicações com o motor *SmallSOA*. Este gerenciamento inclui a necessidade de publicar, buscar e executar uma composição *inSOA*.
- ***Publish*:** este requisito atende a necessidade de publicar uma composição no motor *SmallSOA*. Esta publicação ocorre com o envio de uma interface *XML* para o motor. Esta interface, entre outras informações, inclui o *script inSOA* que representa a composição.
- ***Execute*:** a execução de uma composição é atendida por requisito e compreende a chamada e a apresentação dos resultados obtidos na execução da composição.
- ***Get Compose*:** é o requisito responsável por localizar e capturar uma composição *inSOA* no motor de execução.

Os requisitos descritos da *inSOA* dão suporte a todo o ciclo de desenvolvimento de *scripts* da linguagem. Este ciclo inclui o processo de criação, compilação, publicação e execução de um *script inSOA*. Nas próximas seções serão detalhadas no projeto da aplicação para melhor entendimento da estrutura da aplicação.

4.9 Projeto

Do ponto de vista de projeto a aplicação *inSOA* foi dividida em três camadas distintas. Estas camadas tem por objetivo separar a interface do usuário, o compilador e a comunicação com o motor de composição. Na figura 30 é exemplificada a divisão em camadas da aplicação *inSOA*.

A camada de Interface é responsável pela interação e apresentação com o usuário. Esta camada pode ser substituída ou adaptada por outras aplicações sem com isto alterar as funcionalidades do restante das camadas. Por exemplo, a camada de Interface poderia ser substituída por camada de uma linguagem de propósito geral que viria a utilizar o compilador e a comunicação *inSOA* para suportar composições de serviços em suas funcionalidades.

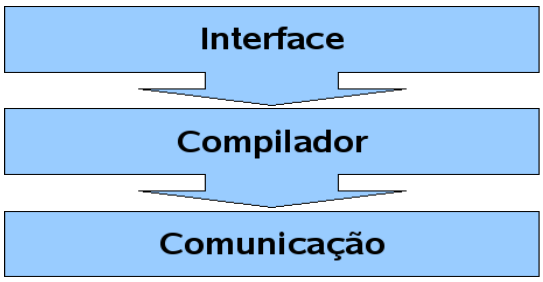

Figura 30: Camadas *inSOA*.

O compilador compreende todos os requisitos de compilação levantados anteriormente nos casos de uso, enquanto a camada de comunicação visa prover um meio de comunicação onde são gerenciadas as integrações *SOA* com o motor de execução.

Para cada camada descrita na figura 30, foram desenvolvidas um conjunto de classes. Estas classes foram separadas em pacotes de acordo com o seu propósito e domínio de aplicação. A divisão em pacotes é apresentada no diagrama de pacotes da figura 31.

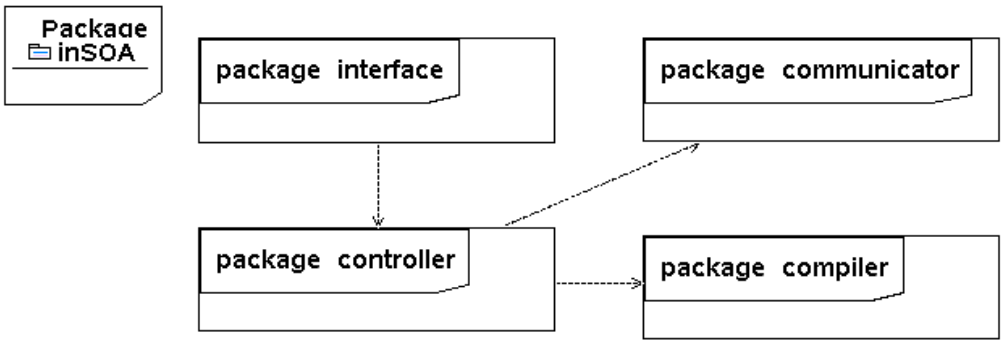

Figura 31: Diagrama de Pacotes.

Os pacotes apresentados na figura 31 agrupam classes que possuem funcionalidades similares. Abaixo são descritas as funcionalidades que cada pacote implementa:

- ***interface*:** neste pacote são armazenadas as classes responsáveis pela interface do usuário. As classes deste pacote não contém regras de negócio. Elas apenas interagem com o usuário e repassam as informações e chamadas para as classes de controle (*controller)*.
- ***controller*:** o pacote de controle redireciona as chamadas da interface para as classes de negócio. Ele também inclui classes utilitárias que preparam os dados para a compilação e a comunicação. Este pacote tem o papel de repositório das classes gerenciadoras.

- ***compiler*:** o pacote *compiler* armazena todas as classes responsáveis pelas análises léxica, *parser* e semântica, bem como, pela otimização e geração do *XML* de integração com o motor de composição.
- ***communicator*:** neste pacote encontram-se as classes responsáveis pela comunicação com o motor de composição *SmallSOA*. Isto inclui as classes responsáveis pela publicação, localização e execução das composições *inSOA*.

A organização e divisão dos pacotes da aplicação baseou-se no padrão de projetos *MVC (Model-View-Controller)* [Buschmann 1996]. O *MVC* separa a camada de apresentação das camadas de controle e negócios, da mesma forma que foi feito no diagrama de pacotes da figura 31.

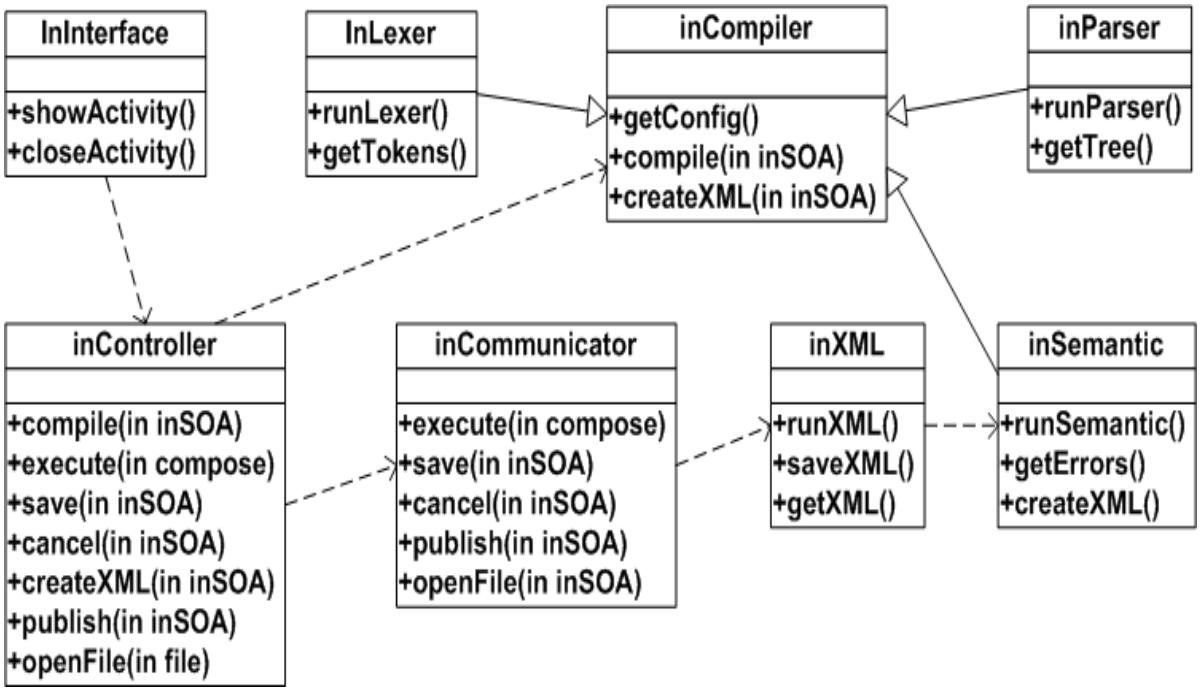

Figura 32: Diagrama de classes das principais classes.

Finalizada a especificação dos pacotes, definiu-se a modelagem das classes. As classes foram definidas e agrupadas nos pacotes apresentados anteriormente. O agrupamento das classes deu-se por similaridade de domínio e especialidade do pacote. O diagrama de classes apresentado na figura 32 representa uma abstração dos requisitos ilustrados nos casos de uso da figura 29. Este diagrama apresenta as principais classes e operações de todas as camadas discutidas até aqui. Por razões de simplicidade, as classes utilitárias, acessórias e os atributos das classes principais não são apresentados neste diagrama.

A classe *InInterface* e todas aquelas relacionadas a interface com o usuário são agrupadas no pacote *interface*. A classe *InController* e suas utilitárias são armazenadas no pacote *controller*. Por conseguinte, as classes *InCompiler*, *InLexer*, *InParser*, *inSemantic* e suas relacionadas são armazenadas no pacote *compiler*. Por fim, a classe *InComunicator*, *InXML* e todas as classes relacionadas a comunicação com o motor de composição são inseridas no pacote *controller*. A divisão da aplicação em pacotes priorizou o agrupamento das classes de acordo com seu domínio e especialização.

4.10 Compilador e Protótipo

inSOA é totalmente escrita em *Java* e preparada para rodar na plataforma *Android.* Por esta razão o *framework ANTLR* foi adaptado no projeto *inSOA* para rodar no *Android*. Pelo mesmo motivo do *ANTLR,* o *framework StringTemplate* [StringTemplate 2008] foi ajustado para rodar no *Android.* O *StringTemplate* é utilizado pela *inSOA* para gerar saídas customizáveis. Ele é utilizado para gerar o *XML* de integração com o motor *SmallSOA.* Entretanto, como será visto adiante, ele pode ser utilizado, por exemplo, para gerar *bytecodes Java*.

Por se tratar de uma plataforma aberta e por possuir como base uma linguagem de programação amplamente difundida, o *Java*, a arquitetura selecionada para o desenvolvimento do projeto foi o *Android* [Android 2008; Haseman 2008]. Além dos itens mencionados, *Android* possui um extenso conjunto de bibliotecas desenvolvidas para a plataforma, além das bibliotecas *Java* para a plataforma *Java Micro Edition*. A plataforma *Android* é composta por um conjunto de ferramentas que abrange, desde sua execução, desenvolvimento e softwares específicos, assim dando autonomia para todo o desenvolvimento do trabalho proposto.

O protótipo da linguagem *inSOA,* desenvolvido para a plataforma *Android,* possui uma interface única, como ilustrado na figura 33.

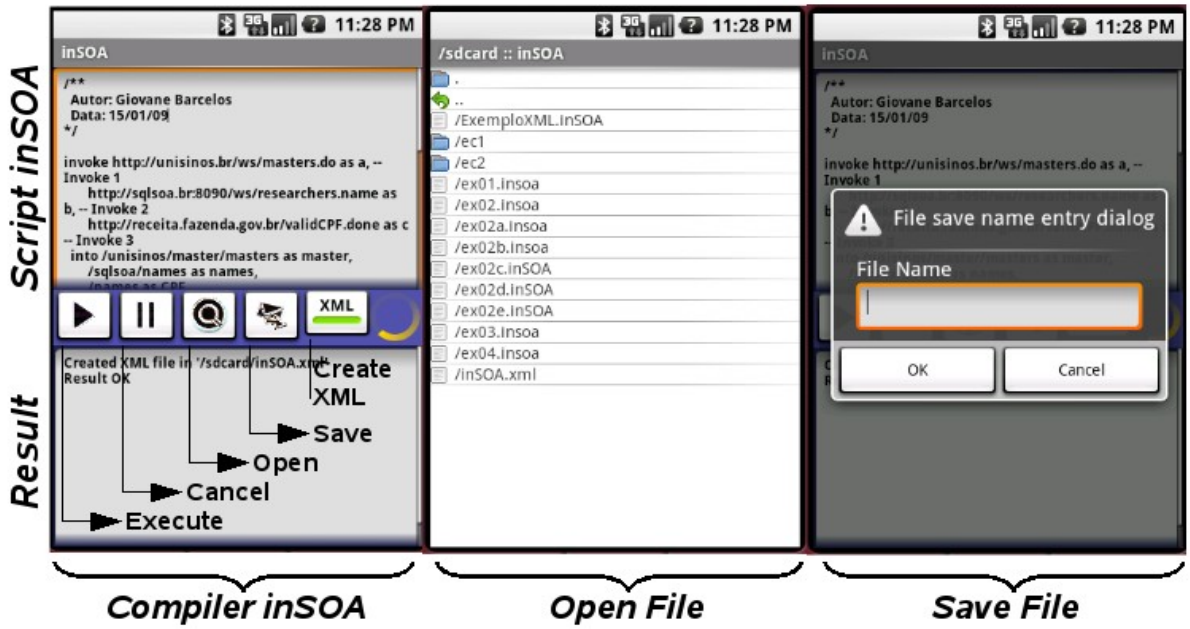

Figura 33: Interface do protótipo *inSOA*.

Na primeira imagem da figura 33 encontra-se a tela principal do compilador que é dividida em três áreas: edição do *script inSOA*, barra de ferramentas e resultado. A área de edição possibilita a edição de um *script inSOA* com tamanho de até 64k. O espaço inferior é reservado para mostrar as mensagens da compilação e o resultado da execução da composição. A barra de ferramentas possui cinco botões com funcionalidades distintas como descrito abaixo:

- ***execute*:** este botão compila, gera o *XML* otimizado, publica e executa a composição. O *XML* otimizado só será gerado neste processo se o botão para criar o XML estiver pressionado.
- ***cancel*:** o *cancel* emite uma sinalização de cancelamento da execução de uma composição em curso.
- ***open*:** este botão tem por objetivo abrir uma composição salva no dispositivo móvel. Na segunda imagem da figura 33 é apresentado o navegador que permite escolher e abrir a composição salva no *Android*.
- ***save*:** possibilita salvar uma composição editada no editor de composições da ferramenta. é necessário informar o nome do arquivo da composição conforme ilustrado na terceira imagem da figura 33.
- ***create XML*:** quando esse botão encontra-se habilitado significa que a operação de execução da composição deve gerar o arquivo *XML* de integração com o motor de composição *SmallSOA*. Se o botão estiver desabilitado, o arquivo de *XML* não será

executado e a ferramenta apenas irá compilar o *script*, não havendo portanto a criação do *XML*, publicação e execução da composição.

As principais funcionalidades dessa ferramenta são: a edição e a execução dos *scripts inSOA*. A execução dá-se via motor de composição após gerado o *XML* de integração. O mecanismo de geração do *XML* é customizável, visto que, o compilador gera uma árvore de objetos em memória que pode ser traduzido pelo *framework StringTemplate* para qualquer saída. Por exemplo, essa saída poderia ser um arquivo de comandos gerador de *bytecodes Java* usando o *Jasmin*[17], que transforma comandos *assembler* em *bytecodes Java*.

4.11 Arquivo *XML* de Integração com *SmallSOA* e *gImpact*

O protótipo *inSOA* é o ambiente de desenvolvimento e execução dos *scripts inSOA*. Ele foi implementado com o objetivo de avaliar o funcionamento da linguagem. Entre outras características deste ambiente encontra-se a integração com o motor de execução *SmallInsoa* e o analisador de impacto *gImpact*. Essa integração dá-se pelo *XML* otimizado descrito na seção anterior e apresentado na figura 34. Este *XML* é otimizado porque declara o paralelismo e a ordem em que os *WS* devem ser executados, bem como, detalha de forma explícita os elementos do *inSOA* e seus valores. Ou seja, o *XML* de integração *inSOA* faz o papel de guia de execução da composição, indicando o melhor caminho e fornecendo as informações de forma estruturada para facilitar a composição dos serviços.

```
<inSOA id="Identification" tags="Example" hash="ba22f53deba9d3d81d0dc94afbdc9b3110e7cd5a">
    <inSOAScript>
        -- inSOA Example
        invoke http://usoa.insoa/WS.Operation as a if way != 1
          input way type Number default 0
            into / as wayPipe
              set a.field := way
          where wayPipe/result/tag/text() == 'Valid'
          return wayPipe/result/field/text() type String name Result
             fail a: invoke http://usoa.insoa/WS1.Operation as b
                        set b.field := 'Rollback',
                            b.way   := way
              id Identification
            tags Example;
    </inSOAScript>
    <Inputs>
        <Input Name="way" Type="Number" Default="0"/>
    </Inputs>
    <Invokes>
        <Invoke Method="SOAP" EndPoint="http://usoa.insoa/WS" NameSpace="" Operation="Operation" Alias="a" Pipe="/"
        Into="wayPipe" ParallelismLevel="1">
            <Iifs>
                <IIF Condition="" NotL="" ExpressionLeft="way" Operator="NE" NotR="" ExpressionRight="1"/>
            </Iifs>
            <Parameters>
                <Parameter Name="field" Value="way" Type=""/>
            </Parameters>
```

17 *JASMIN - http://jasmin.sourceforge.net/*

```
        </Invoke>
    </Invokes>

    <Wheres>
        <Where Condition="" NotL="" ExpressionLeft="wayPipe/result/tag/text()" Operator="EQ" NotR=""
        ExpressionRight="'Valid'"/>
    </Wheres>

    <Returns>
        <Return ReturnExpression="wayPipe/result/field/text()" TypeReturnExpression="String"
        NameReturnExpression="Result"/>
    </Returns>

    <Fails>
        <Fail Alias="a">
            <Invokes>
                <Invoke Method="SOAP" EndPoint="http://usoa.insoa/WS1" NameSpace="" Operation="Operation" Alias="b"
                Pipe="" Into="" ParallelismLevel="0">
                    <Parameters>
                        <Parameter Name="field" Value="Rollback" Type=""/>
                        <Parameter Name="way" Value="way" Type=""/>
                    </Parameters>
                </Invoke>
            </Invokes>
        </Fail>
    </Fails>

</inSOA>
```

Figura 34: Exemplo do *XML* de integração do *inSOA*.

Como se pode perceber na figura 34 o *XML* de integração do *inSOA* é composto de sete seções principais:

- **inSOA:** é o documento XML propriamente dito. Possui os seguintes atributos:

Tabela 20: Atributos da seção *inSOA* do *XML* de integração do *INSOA*.

Atributo	Descrição
Id	Identificação da composição.
Tags	São as classificações da composição. Essas são utilizadas como chaves de pesquisa quando deseja-se localizar composições *inSOA* publicadas no motor *SmallSOA*.
Hash	Identificação única do *script inSOA*. Utiliza o algoritmo de dispersão *SHA-1 (Secure Hash Algorithm)*.

- **inSOAScript:** nesta seção encontra-se o script inSOA original.
- **Inputs:** são descritos os parâmetros de entrada da composição, ou seja, é a interface da composição. Ela contém uma outra seção chamada de Input que contém os atributos descritos na tabela abaixo:

Tabela 21: Atributos da seção *Input* do *XML* de integração do *INSOA*.

Atributo	Descrição
Name	Nome do parâmetro de entrada.
Type	Tipo de dado do parâmetro. Os tipos suportados são: *string*, *number* ou *base64*. O tipo *base64* é utilizado quando deseja-se enviar para a composição um tipo de dado complexo.

Atributo	Descrição
Default	É o valor padrão do parâmetro, ou seja, quando não é fornecido nenhum dado para a aplicação, este é o valor utilizado no parâmetro.

- **Invokes:** nesta seção são declarados todos os invokes da composição, ou seja, todos os WS com seus parâmetros e decisões de fluxo. Abaixo são descritos cada um dos atributos desta seção e suas subseções:

Tabela 22: Atributos da seção *Invokes* do *XML* de integração do *INSOA*.

Atributo	Descrição
Invoke	Subseção *Invoke*. Esta subseção declara todos os *WS* que devem ser executados.
Method	É o método de chamada do *WS*. Pode ser *SOAP* ou *REST*.
EndPoint	Endereço *web* do *WS*.
NameSpace	É o *namespace* que será utilizado internamente no *XML* de comunicação do corpo do *SOAP* ou *REST*.
Operation	Corresponde ao nome da operação do *WS*.
Alias	É o nome da variável identificadora do *WS* e seus atributos. Este valor pode ser utilizado em expressões das outras seções.
Pipe	Este atributo descreve o *XPath* que irá tratar o retorno do *WS*.
Into	Define o nome da variável que deve armazenar o resultado da execução do *Pipe* sobre o retorno do *WS*.
ParallelismLevel	Declara o nível de paralelismo do *WS*. É um número sequêncial que diz a ordem em que o *WS* deve ser executado. Se o número de sequência se repetir em mais de um *Invoke*, significa que estes *WS* podem ser executados em paralelo. A declaração destes *Invokes* é sempre em ordem sequencial.
Iifs	Esta subseção contém as decisões de fluxo que devem ser satisfeitas para que o *WS* seja executado.
Condition	Condição de projeção que deve ser atendida. Os valores suportados são *OR* ou *AND*. Onde, *OR* é disjunção e *AND* é a conjunção.
NotL	É a negação da expressão. O valor esperado é *NOT*.
ExpressionLeft	Expressão de comparação do lado esquerdo declarada no *script inSOA*.
Operator	Declaração de um operador relacional ou de restrição. Estes operadores podem ser: *EQ*, *NE*, *LT*, *LE*, *GT* ou *GE*, que correspondem respectivamente aos operadores ==, *!*=, <, <=, > *e* >=.
NotR	Negação da expressão. O valor esperado é *NOT*.

Atributo	Descrição
ExpressionRigth	Expressão de comparação do lado direito declarada no *script inSOA*.
Parameters	Subseção que define os parâmetros do *WS*.
Name	Nome do parâmetro do *WS*.
Value	Valor que deve ser atribuído ao parâmetro do *WS*. Este valor pode ser uma expressão *inSOA*.
Type	Tipo do parâmetro. Este valor é opcional.

- **Wheres:** esta seção descreve as projeções relacionais e restrições da composição, conforme descrição dos atributos da sua subseção Where apresentada na tabela abaixo:

Tabela 23: Atributos da seção *Where* do *XML* de integração do *INSOA*.

Atributo	Descrição
Condition	Da mesma forma que o atributo *condition* subseção *Iifs*, este atributo representa a condição de projeção da composição. Os valores suportados são *OR* ou *AND*. Onde, *OR* é disjunção e *AND* é a conjunção.
NotL	Negação da expressão. O valor esperado é *NOT*.
ExpressionLeft	Expressão de comparação do lado esquerdo.
Operator	Operador relacional ou de restrição. Segue o padrão já descrito anteriormente: *EQ*, *NE*, *LT*, *LE*, *GT* ou *GE*.
NotR	Negação da expressão. O valor esperado é *NOT*.
ExpressionRight	Expressão de comparação do lado direito.

- **Returns:** descreve os retornos da composição, conforme resumo dos atributos da subseção Return apresentados na tabela abaixo:

Tabela 24: Atributos da seção *Return* do *XML* de integração do *INSOA*.

Atributo	Descrição
Expression	Contém a expressão a ser retornada da composição.
TypeReturn	Tipo do retorno. Os tipos suportados são os mesmos do retorno descrito na subseção *Input*: *string, number* ou *base64*.
NameReturn	Nome do campo de retorno.

- **Fails:** Esta seção descreve a interceptação das falhas na execução dos WS e o tratamento dos mesmos. Ela contém uma subseção chamada de FailAlias, a qual possui um atributo chamado de Alias, o qual contém o alias do WS que gerou o erro. Dentro da subseção FailAlias é descrito uma seção Invoke que descreve uma

composição inSOA para tratar a falha ocorrida no WS interceptado. Esta seção Invoke possui a mesma estrutura de composição descrita até aqui.

O *XML* de integração é utilizado tanto pelo motor de composição *SmallInsoa* e quanto pelo analisador de impacto *gImpact*. O *SmallInsoa* utiliza o *XML* para publicar, localizar e executar uma composição de serviços, enquanto o *gImpact* usa o *XML* para navegar em uma composição e analisar os impactos de alteração da mesma. Como discutido anteriormente, o *XML* de integração é gerado utilizando o *framework StringTemplate.*

```
group inSOA;

invokeMain(script, isoa ) ::= <<
\<?xml version="1.0" encoding="UTF-8"?\>
\<inSOA id="<isoa.id>" tags="<isoa.tags>" hash="<isoa.hash>"\>
  \<inSOAScript\>
    <script>
  \</inSOAScript\>
  <isoa:invoke()>
\</inSOA\>
>>

invoke(isoa) ::= <<
  <if(isoa.boInputs)>
  \<Inputs\>
    <isoa.inputs:inputAux()>
  \</Inputs\>
  <endif>
  \<Invokes\>
   <isoa.invokes:invokex()>
  \</Invokes\>
  <if(isoa.boWheres)>
  \<Wheres\>
    <isoa.wheres: { wheres | <wheres:whereAux()> }>
  \</Wheres\>
  <endif>
  <if(isoa.boReturns)>
  \<Returns\>
    <isoa.returns:returnAux()>
  \</Returns\>
  <endif>
  <if(isoa.boFails)>
  \<Fails\>
    <isoa.fails:failAux()>
  \</Fails\>
  <endif>
>>

inputAux(inputs) ::= <<
  <inputs: { input |
  \<Input Name="<input.name>" Type="<input.type>" Default="<input.defaultt>"/\>
  }>
>>
```

Figura 35: Trecho de código do *inSOA* utilizando *StringTemplate*.

Como pode-se perceber no trecho de código da figura 35, o *StringTemplate* é um documento texto que funciona como uma linguagem interpretada onde são declaradas funções

e comandos, tais como, *invokeMain* e *if*, de forma estrutura, de modo que pode-se gerar qualquer saída texto desejada. No caso do *inSOA* a saída é um *XML*, mas poderia ser, por exemplo, instruções para geradores de *bytecodes Java* ou um relatório com a árvore sintática da linguagem.

A entrada para um documento *StringTemplate* são sempre passados via parâmetros das funções declaradas neste. No *inSOA* este parâmetro é o *script inSOA* e um objeto chamado de *iSOA*. Este objeto contém uma estrutura de objetos, como demonstrado no diagrama de classes da figura 36 que representa todos os comandos da linguagem. O objeto iSOA é uma representação da árvore sintática da linguagem e portanto pode ser manipulado para qualquer propósito relacionado ao domínio da aplicação.

A entrada para um documento StringTemplate é sempre passada pelos parâmetros das funções declaradas neste. No inSOA estes parâmetros são o script inSOA e um objeto chamado de iSOA. Este objeto contém uma estrutura de objetos, como diagramado no diagrama de classes da figura 36 que representa todos os comandos da linguagem. O objeto iSOA é uma representação da árvore sintática da linguagem e portanto pode ser manipulado para qualquer propósito relacionado ao domínio da aplicação.

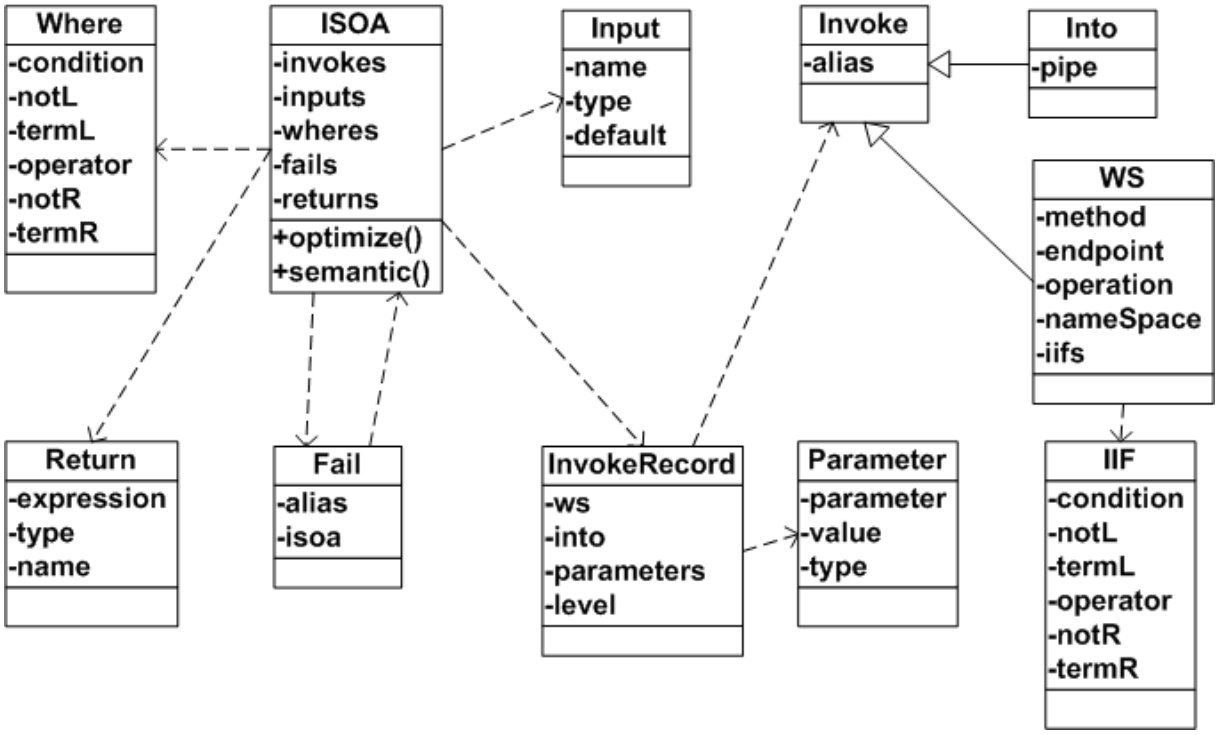

Figura 36: Diagrama de classes do objeto *iSOA*.

Como pode-se verificar na figura 36, a classe do objeto iSOA se relaciona com outras classes que representam os comandos da linguagem *inSOA*. Existe uma classe para cada

comando da linguagem. Estas classes são instanciadas durante o processo de compilação, de acordo com suas características e relacionamentos, ficando todas atreladas ao objeto *iSOA*. Este objeto é transferido para o *StrinTemplate,* juntamente com o *script,* após o término da compilação, gerando o *XML* de saída.

4.12 Tratamento de erros de compilação

Durante o processo de compilação do *script inSOA* todos os erros de sintaxe e semântica são analisados e apresentados para o usuário na área de resultados do compilador *inSOA*, conforme exemplo da figura 37. O compilador informa todos os erros encontrados durante o processo de compilação do script e não somente a primeira ocorrência de erro.

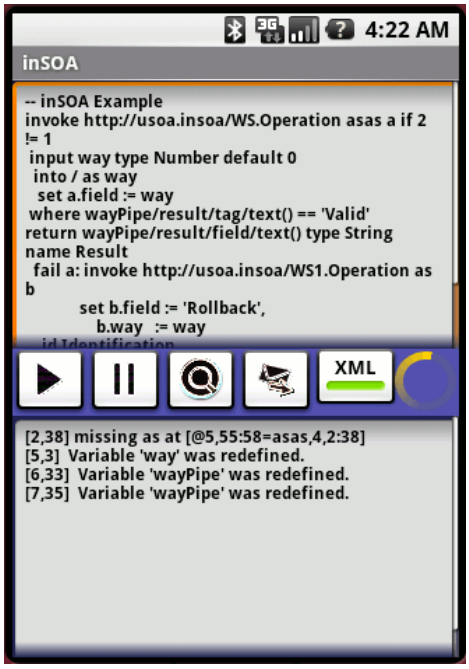

Figura 37: Tela com os erros de compilação do inSOA.

Todos os erros apresentados para usuário sempre indicam a linha e coluna da ocorrência deste, além de informar a regra de produção não atendida da gramática e o *token* de entrada que ocasionou o problema. Na tabela 25 são descritos os principais erros de compilação do compilador *inSOA* informados ao usuário.

Tabela 25: Principais erros do compilador *inSOA*.

Erro	Descrição
Extraneous input <input> expecting <token>	A sequência de caracteres analisada não combina com a regra do <*token*>.

Erro	Descrição
Missing <token> at <coordinates>.	O compilador sentiu falta de um *token* descrito na regra em análise.
Mismatched tree node: <node> expecting <token>	Um nó na análise da árvore não foi correspondido com o <*token*> anaisado.
No viable alternative at input <token>	Nenhuma das alternativas de uma regra foram satisfeitas.
Required (...)+ loop did not match anything at input <token>	O compilador não encontrou um *token* compatível com a entrada na análise de uma regra de repetição.
Rule <rule> failed predicate: <predicate>	Erro na análise de uma regra de predicado.
Variable <variable> was redefined.	Varíavel <*variable*> já definida.
It's necessary to define the 'invoke' for the variable <variable>	Existem mais tratamento *XPath* no comando *Into* separados por vírgula do que definições de *WS* no *Invoke*.
The quantity of 'into' elements is below than 'invoke' statement	A quantidade de elementos *into* é menor que a quantidade de definições do *invoke*.
The parameter <parameter> is not defined	Um parâmetro da entrada não foi definido. Um parâmetro pode ser um parâmetro da composição ou uma variável definido no comando *into* ou *invoke*.
The into value <variable> is not defined	O valor definido com sendo do *into* não foi definido.
The clause 'other' has already been used	A cláusula other já foi usada.
It is necessary a web services operation separeted by the char '.' in the URI definition	É preciso declarar a operação do *WS* separada pelo caracter '.'.

Um mesmo *token* do *script* de entrada pode ser objeto de um ou mais erros. Estes erros sempre indicam problemas que precisam ser corrigidos no *script*, pois a gramática não foi respeitada e a composição está incorreta. Sendo assim, o arquivo *XML* de integração não é gerado mesmo que o botão para gerá-lo esteja pressionado, visto que aqui há erro na composição.

5 ESTUDO DE CASO E RESULTADOS

Segundo [Denzin 2000], o estudo de caso não é um método, mas a escolha de um objeto a ser estudado. Ele permite o estudo de fenômenos em profundidade dentro de seu contexto e possibilita a exploração de fenômenos com base em vários ângulos. Esta metodologia é uma estratégia de pesquisa adequada para avaliar implementações e suas implicações em casos reais de uso.

Com o objetivo de avaliar as funcionalidades da linguagem *inSOA* foram criados alguns cenários utilizando a metodologia de estudo de caso. Estes cenários possibilitaram medir e avaliar alguns indicadores estatísticos. Os indicadores obtidos no estudo de caso viabilizaram a análise e conclusões que serão discutidas na seção 5.2.

5.1 Estudo de Caso

Foram desenvolvidos dois cenários para o estudo de caso com o objetivo de avaliar a composição de serviços utilizando a linguagem *inSOA*. Estes cenários incluíram um processo de trabalho como apresentado na figura 38. Este processo de trabalho, como será apresentado em seguida, inclui o compilador *inSOA*, o motor de execução *SmallSOA* e o analisador de impacto *gImpact*.

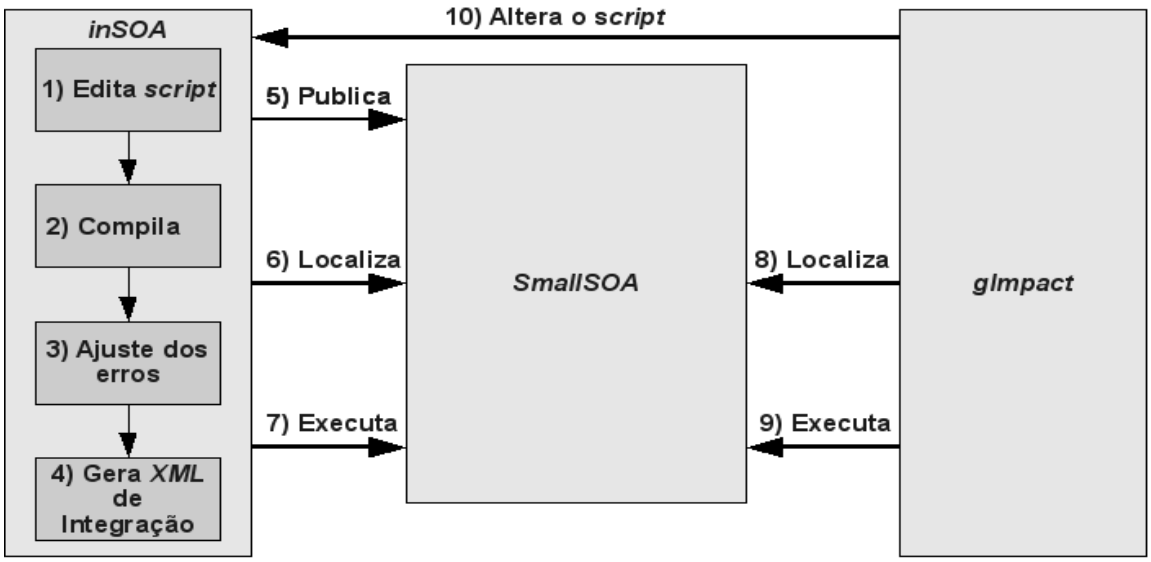

Figura 38: Processo do Estudo de Caso.

O processo do estudo de caso apresentado na figura 38 demonstra os passos de integração entre as aplicações durante os testes. Este processo incluiu dez passos, como descritos abaixo:

1) **Edita script:** dentro do ambiente de desenvolvimento do compilador *inSOA* é editado cada *script*.
2) **Compila:** nesta etapa é realizada a compilação do *script*, o que ocorre quando é pressionado o botão de execução do *script*.
3) **Ajuste dos erros:** caso ocorra algum erro durante a compilação, este deve ser ajustado pelo usuário antes de continuar o processo.
4) **Gera *XML*:** se o botão de geração do *XML* de integração estiver pressionado, o arquivo será automaticamente gerado durante o processo de execução do *script*.
5) **Publica:** esta etapa corresponde ao processo de publicação do *script inSOA* no motor de composição *SmallSOA*.
6) **Localiza:** etapa de localização do *script inSOA* no *SmallSOA*. Este processo pode ocorrer via nome de identificação da composição, código *hash* ou procura por categoria.
7) **Executa:** a execução é solicitada pelo compilador *inSOA* para o *Small*SOA. Após a execução o resultado é apresentado na área de resultados do compilador.
8) **Localiza:** o *gImpact* solicita as composições *inSOA* para o motor de composição com o objetivo de calcular o impacto de uma alteração em uma composição publicada.
9) **Executa:** com o objetivo de calcular impacto dos tempos de execução, o *gImpact* também executa composições *inSOA* no motor *SmallSOA*.
10) **Altera o *script*:** tendo calculado o impacto de alteração em uma composição com o *gImpact*, é possível realizar uma alteração no respectivo *script* e iniciar novamente o processo no compilador *inSOA*.

Todos os testes rodaram sobre a plataforma e emulador *Android.* Cada uma das composições, as quais serão apresentadas nas próximas seções, foram desenvolvidas em um ambiente de desenvolvimento *inSOA* no emulador *Android*. Foram utilizados *WS* finais reais e públicos, tais como, os *WS* da Amazon Books[18]. As composições, por sua vez, foram

18 *Amazon Books - http://www.amazon.com/*

publicadas e executadas em servidores *SmallSOA* sobre emuladores *Android* distintos. Sendo assim, cada servidor *SmallSOA* armazenou apenas uma composição por vez, exigindo desta forma a comunicação entre os diferentes servidores. Por fim, o analisador *gImpact*, do mesmo modo que o *inSOA*, foi executado em um emulador *Android*.

5.1.1 Cenário 1: Consulta de Viagem

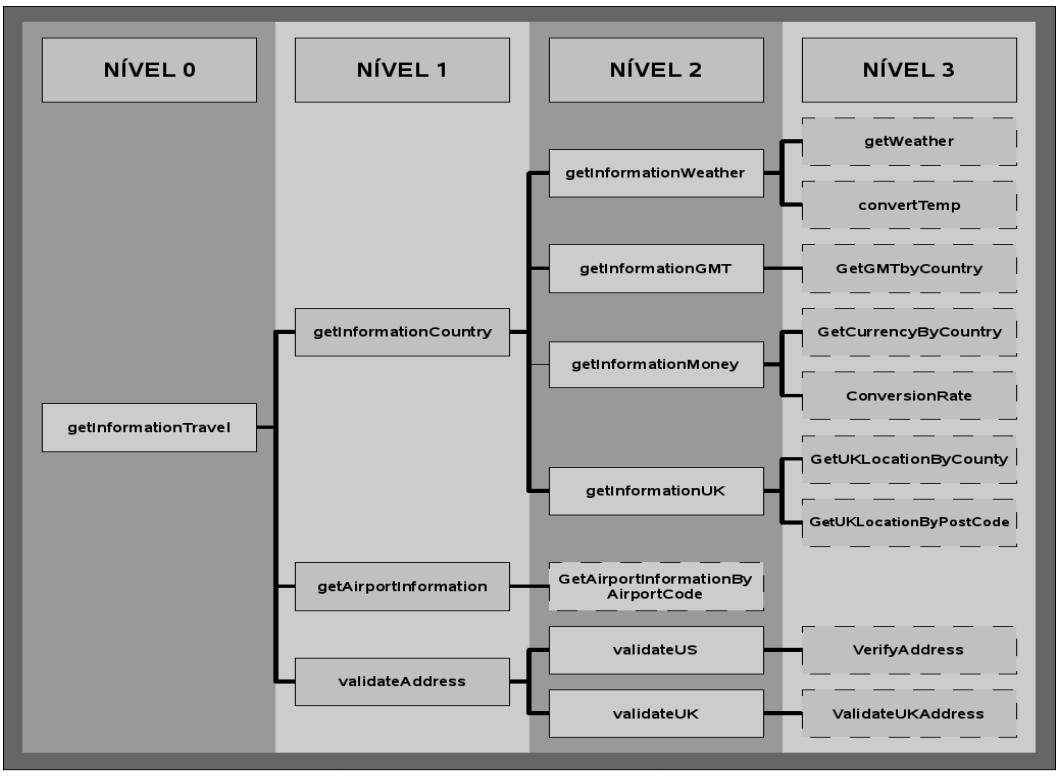

Figura 39: Consulta de Viagem.

O cenário 1, apresentado na figura 39, representa uma consulta de viagens com quatro níveis de composição. Cada nível é composto de uma ou mais composições. Cada composição do nível analisado pode utilizar um ou mais *WS*. Os retângulos contínuos da figura são composições *inSOA*, enquanto os pontilhados são *WS*. Apesar de uma composição *inSOA* declarar os *invokes* como *WS*, estes *WS* podem ser na verdade outra composição. Do ponto de vista de execução, o processo de invocar uma composição como um *WS* é transparente e não exige maiores declarações. Entretanto, para uma efetiva análise de impacto, a ciência do que se trata de composição é essencial, porque o analisador precisa navegar na árvore de dependência das composições até chegar no nível de *WS*. Por esta razão o motor *SmallSOA* comunica o requisitante do serviço do que se trata o *WS*, se é um *WS* propriamente dito ou uma composição *inSOA*.

```
-- inSOA 1: getInformationTravel
invoke http://usoa.smallSOAA.com/smallSOAService.executeComposition as a, -- 1.1
        http://usoa.smallSOAB.com/smallSOAService.executeComposition as b, -- 1.2
        http://usoa.smallSOAC.com/smallSOAService.executeComposition as c  -- 1.3
  input cityTown type String default 'San Diego', stateCounty type String default 'CA',
        zipPostCode type String default '92111',   CountryName type String default 'United States',
        ToCurrency type String default 'BRL',     AirportCode type String default 'SAN'
    into / as country, / as airport, / as validate
     set a.composition := 'getInformationCountry',
        a.parameters := ''CityName='' || cityTown || ''';CountryName='' || CountryName || ''';ToCurrency='' || ToCurrency ||
                        ''';County='' || stateCounty || ''';PostCode='' || zipPostCode || """,
        b.composition := 'getAirportInformation',
        b.parameters := ''airportCode='' || AirportCode || """,
        c.composition := 'validateService',
        c.parameters := ''cityTown='' || cityTown || ''';stateCounty='' || stateCounty || ''';zipPostCode='' || zipPostCode || """"
return country/result/info/text() || airport/result/cityAirport/text() || validate/result/Valid/text() type String name Information
      id getInformationTravel;
```

Figura 40: *Script inSOA* da composição *getInformationTravel*.

A consulta deste cenário visa fornecer informações de um destino de viagem pré-determinado. Esta consulta é acionada pela operação *getInformationTravel*. O *getInformationTravel* é uma composição *inSOA,* como apresentada na figura 40, que possui os seguintes parâmetros de entrada:

- ***countryName*:** nome do país de destino.
- ***stateCounty*:** código do estado ou região de destino.
- ***cityTown*:** nome da cidade ou distrito de destino.
- ***zipPostCode*:** código de endereçamento postal do destino. Este código é opcional.
- ***airportCode:***código do aeroporto que se deseja aterrizar no destino.
- ***toCurrency*:** código da moeda do local de origem, ou seja, a moeda em que se deseja receber os valores de retorno da composição.

O resultado final desta consulta será:

- **Informações do país de destino:** quantidade de graus Célsius.
- **Fuso horário *GMT*:** *GMT* (*Greenwich Mean Time*) do destino, que significa a hora média de *Greenwich*.
- **Moeda local:** moeda local já com a taxa de conversão para a moeda local.
- **Nome da região:** nome da região do país de destino.
- **Cidade do aeroporto:** nome do aeroporto e da cidade do aeroporto.
- **Validade do Endereço:** indicador de validade do endereço de destino.

Foram desenvolvidas outras nove composições além da composição apresentada na figura 40. Para suportar estas composições, foram utilizados dez *WS* distintos, conforme apresentado na figura 39 nos retângulos pontilhados. Estes *WS* trocam informações entre si coordenados pela composição *inSOA,* A composição *inSOA* aciona os *WS* chamando suas

operações com seus devidos parâmetros. Abaixo segue uma breve descrição da função de cada um dos *WS* e composições desenvolvidas:

1) **Composições *inSOA*:**

- ***getInformationTravel:*** composição principal. Retorna as informações consolidadas da viagem de destino.
- ***getInformationCountry*:** responsável por fornecer informações dos graus célsius, do horário *GMT*, da taxa de conversão para o Real e nome da região de destino.
- ***getAirportInformation*:** fornece o nome da cidade e do aeroporto de destino.
- ***validateAddress*:** responsável por validar o endereço de destino fornecido pelo usuário
- ***getInformationWeather*:** fornece os graus célsius de uma determina localização.
- ***getInformationGMT*:** retorna o fuso horário *GMT* da localização.
- ***getInformationMoney*:** fornece a taxa de conversão da moeda do país de destino em relação ao real.
- ***getInformationUK*:** retorna informações do pais de destino, tais como, o nome da região do local pesquisado.
- ***validateUS*:** valida localização nos Estados Unidos.
- ***validateUK*:** valida localização na Inglaterra.

2) ***Web Services*:**

- ***getWeather:*** captura informações da temperatura de um local.
- ***convertTemp:*** converte uma temperatura para diferentes padrões de medida, como por exemplo graus *célsius*.
- ***getGMTbyCountry:*** retorna o fuso horário no padrão *GMT*.
- ***getCurrencyByCountry:*** captura o código da moeda de um país.
- ***conversionRate:*** retorna a taxa de conversão de uma moeda origem para uma de destino.
- ***getUKLocationByCounty:*** retorna informações de uma localização filtrando por país.
- ***getUKLocationByPostCode:*** retorna informações de uma localização filtrando por código postal.
- ***getAirportInformationByAirportCode:*** captura informações de um aeroporto pelo código internacional do aeroporto.
- ***verifyAddress:*** valida um endereço nos Estados Unidos.

- ***validateUKAddress:*** valida um endereço na Inglaterra.

Para cada composição *inSOA* foi gerado o arquivo *XML* de integração correspondente. Estes arquivos, conforme discutido anteriormente, foram executados no motor de composição *inSOA,* o qual retornou o resultado final para o compilador *inSOA* que o apresentou para o usuário conforme a figura 41.

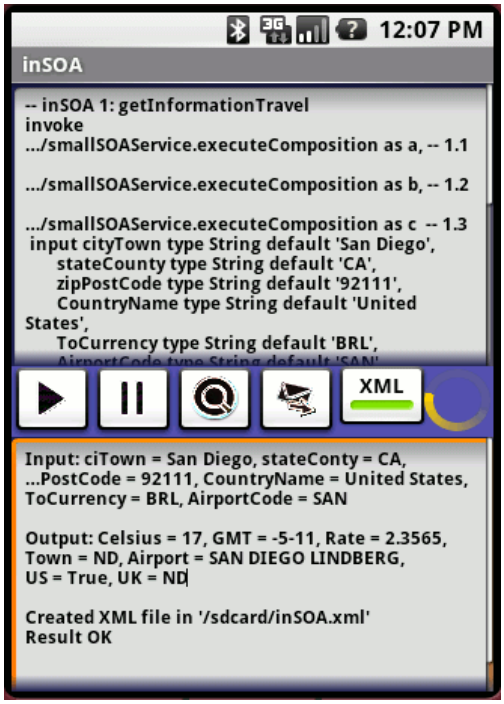

Figura 41: Resultado da composição *getInformationTravel*.

5.1.2 Cenário 2: Consulta de Livros

Este cenário representa uma composição *inSOA* que tem por objetivo retornar e tratar informações de livros consultados na loja virtual *Amazon Books*[19]. Da mesma forma que o cenário 1, este cenário invoca uma série de *WS* dependentes. Alguns destes *WS* são composições enquanto outros são *WS* propriamente dito. Na figura 42 é apresentado um diagrama com a estrutura de dependência dos *WS* dividida em cinco níveis. Neste cenário foram desenvolvidos doze composições *inSOA* que utilizaram-se de oito *WS,* os quais estão destacados na figura 42 nos retângulos pontilhados.

19 *Amazon Books - http://www.amazon.com/*

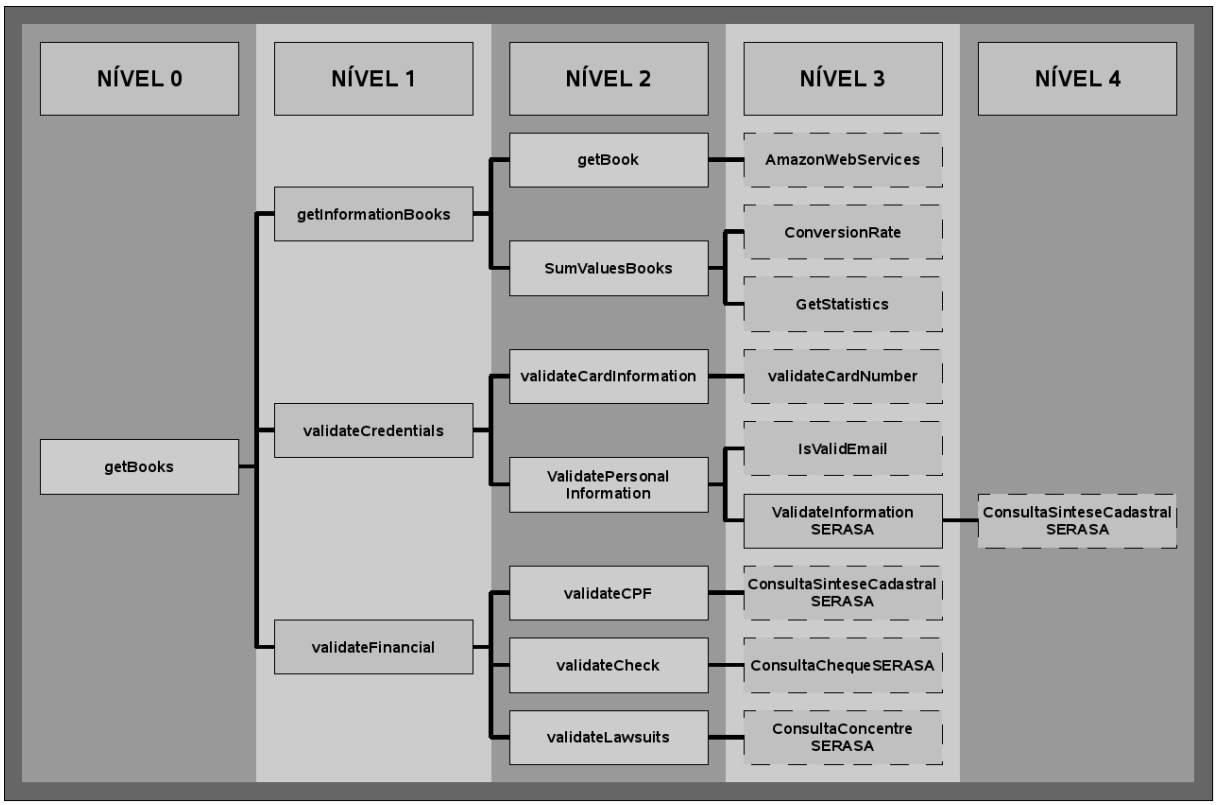

Figura 42: Cenário I - Consulta de Livros.

```
-- inSOA 1: getBooks
invoke http://usoa.smallSOAA.com/smallSOAService.executeComposition as a, -- 1.1
       http://usoa.smallSOAB.com/smallSOAService.executeComposition as b, -- 1.2
       http://usoa.smallSOAC.com/smallSOAService.executeComposition as c  -- 1.3
  input title type String, author type String, isbn type String default '9780978739256',
       email type String default 'test@test.com', Senha type String default 'test', Documento type String default '12345678910',
       Banco type String default '001', Agencia type String default '04790', ContaCorrente type String default '99999',
       ChequeInicial type Number default 1234, ChequeInicialDigito type Number default 1
    into / as books, / as credentials, / as financial
     set a.composition := 'getInformationBooks', a.parameters := 'Keywords=' || isbn || ';Title=' || title || ';Author=' || author,
       b.composition := 'validateCredentials', b.parameters := 'email=' || email || ';Senha=' || Senha || ';Documento=' || Documento,
       c.composition := 'validateFinancial',
       b.parameters := 'email=' || email || ';Senha=' || Senha || ';Documento=' || Documento ||
                       ';Banco=' || Banco || ';Agencia=' || Agencia || ';ContaCorrente=' || ContaCorrente ||
                       ';ChequeInicial=' || ChequeInicial || ';ChequeInicialDigito=' || ChequeInicialDigito
return books/title/text() type String name title, books/author/text() type String name author,
       books/listPrice/text() type Number name listPrice, books/lowestPrice/text() type Number name lowestPrice,
       books/average/text() type Number name average,  credentials/cardValidation/text() type String name cardValidation,
       credentials/validEmail/text() type String name validEmail, credentials/validCPF/text() type String name validCPF,
       financial/validateSERASA/text() type String name validateSERASA,
       financial/validCheck/text() type String name validCheck,
       financial/lawsuitText/text() type String name lawsuitText
     id getBooks;
```

Figura 43: *Script inSOA* da composição *getInformationTravel*.

Como pode-se verificar no *script inSOA* da figura 43, a composição principal chamada de *getBooks* é composta de onze parâmetros de entrada e onze retornos como descritos abaixo:

1) **Entradas:**

- ***title:*** título do livro. Parâmetro opcional.
- **author:** nome do autor. Parâmetro é opcional.

- ***isbn*:** código de identificação do livro. *ISBN* (*International Standard Book Number*) é um número de identificação internacional único do livro. Este parâmetro é opcional.
- ***email* e *senha*:** alguns dos *WS* de validação financeira e de crédito exigem um cadastro prévio com *email* e senha, por isto estas informações são necessárias para que estes *WS* sejam invocados.
- **documento:** cartão de crédito do comprador do livro.
- **banco, agência, contaCorrente, chequeInicial e chequeInicialDigito**: informações bancárias e sequência dos cheques que serão utilizados para efetuar o pagamento dos livros pesquisados. Estas informações são utilizadas pelos WS que executam a validação financeira dos cheques.

2) **Retornos:**

- ***title*:** título do livro.
- ***author*:** autor do livro.
- ***listPrice*:** preço da lista de preço com os descontos.
- ***lowestPrice*:** menor preço dos usados.
- ***average*:** média dos preços.
- ***cardValidation*:** validação do cartão de crédito, ou seja, retorna se o cartão de crédito existe e se este se encontra em período válido.
- ***validEmail*:** validação do email. Verifica se o email realmente existe.
- ***validCPF*:** verifica se o documento de identificação *CPF* é válido.
- ***validateSERASA*:** situação do cliente na empresa de análise de crédito *SERASA*.
- ***validcheck*:** verifica validade do cheque.
- ***lawsuitText*:** identificação de processo judicial contra o cliente.

Foram desenvolvidas outras onze composições além da composição apresentada na figura 43. Para suportar estas composições, foram utilizados dez *WS* distintos, conforme apresentado na figura 42 nos retângulos pontilhados. Estes *WS* trocam informações entre si coordenados pela composição *inSOA,* A composição *inSOA* aciona os *WS* chamando suas operações com seus devidos parâmetros. Abaixo segue uma breve descrição da função de cada um dos *WS* e composições desenvolvidas:

1) **Composições *inSOA*:**

- ***getBooks:*** composição principal. Retorna as informações consolidadas da consulta de livros.

- ***getInformationBooks:*** responsável por capturar informações do livro.
- ***validateCredentials:*** esta composição tem por finalidade validar os documentos e identificações do cliente.
- ***validateFinancial:*** valida o crédito e informações financeiras.
- ***getBook:*** busca informações de livros na loja virtual *Amazon Books*.
- ***sumValuesBooks:*** converte valores para a moeda de origem e calcula estatísticas de preços.
- ***validateCardInformation*****:** valida um código de cartão de crédito e captura sua data de validade.
- ***validatePersonalInformation*****:** responsável por verificar a existência de um email e validar um determinado *CPF*.
- ***validateInformationSERASA:*** valida se o cliente encontra-se com problemas de crédito no *SERASA*.
- ***validateCPF*****:** valida situação do *CPF* na Receita Federal do Brasil.
- ***validateCheck*****:** verifica situação bancária do cliente e se os cheques emitidos não encontram-se bloqueados.
- ***validateLawsuits*****:** valida se o cliente possui processo judicial.

2) ***Web Services*****:**

- ***amazonWebServices*****:** este *WS* permite pesquisas de livros na loja virtal da *Amazon Books*.
- ***conversionRate:*** retorna a taxa de conversão entre duas moedas na data atual.
- ***getStatistics:*** calcula informações estatísticas de um conjunto de dados.
- ***validateCardNumber:*** retorna informações a respeito de um número de cartão de crédito.
- ***isValidEmail:*** verifica a validade e existência de um endereço de *email*.
- ***consultaSinteseCadastralSERASA:*** retorna resumo das informações cadastrais de um cliente registrado no SERASA.
- ***consultaChequeSERASA:*** verifica informações bancárias de uma sequência de cheques do cliente.
- ***consultaConcentreSERASA:*** entre outras coisas, retorna os processos judiciais que o cliente possui relacionados a questões financeiras.

5.2 Resultados do Estudo de Caso

Durante o desenvolvimento do estudo de caso alguns indicadores foram coletados, medidos e avaliados. As avaliações permitiram algumas correções e conclusões sobre a linguagem *inSOA* que serão apresentadas nas próximas seções.

5.2.1 Tempos e Performance

Os tempos de execução do compilador *inSOA* foram coletados durante o processo de compilação dos *scripts inSOA* desenvolvidos no estudo de caso. A coleta foi efetuada utilizando as ferramentas de *profiler* do *Android*: *dmtracedump* e *traceview* [Android 2008]. Estas ferramentas fornecem o tempo exato de execução e a quantidade de ciclos de processamento de cada método de uma aplicação *Android*. Os ciclos representam o custo de processamento de um método dentro da aplicação, enquanto o tempo, é a soma em segundos de um trecho determinado. Este tempo é exato, pois é descontado o tempo dos processos em execução na máquina e o tempo dos trechos de código da coleta do *Android,* ficando desta forma, somente o tempo do método em si. A avaliação da quantidade de ciclos de uma aplicação é útil por permitir um foco maior nos pontos de maior custo para a aplicação.

Na tabela 26 são apresentados a quantidade de ciclos médios de execução dos métodos mais utilizados dentro da aplicação. Como podemos verificar, todos os métodos desta tabela, fazem parte do *framework ANTLR* ou da biblioteca *Java*. Esta avaliação permitiu otimizar alguns pontos da aplicação. Inicialmente, alguns métodos do pacote *compiler* estavam aparecendo entre os primeiros desta tabela. Entretanto, estes métodos foram reescritos, permitindo uma melhor performance na aplicação. A quantidade de ciclos médios foi calculada a partir da média de dez replicações de cada composição do estudo de caso com diferentes parâmetros.

Tabela 26: Ciclos de execução dos métodos do compilador *inSOA*.

#	Método	Ciclos Médios	%	% Total
1	antlr/InputBuffer.LA	201	18,96%	18,96%
2	antlr/CharBuffer.fill	116	10,94%	29,91%
3	antlr/CharScanner.LA	96	9,06%	38,96%
4	antlr/InputBuffer.syncConsume	76	7,17%	46,13%
5	antlr/CharQueue.elementAt	50	4,72%	50,85%
6	org/antlr/stringtemplate/language/GroupLexer.mBIGSTRING	39	3,68%	54,53%
7	antlr/CharScanner.consume	33	3,11%	57,64%
8	java/io/StringReader.read	26	2,45%	60,09%

#	Método	Ciclos Médios	%	% Total
9	antlr/ANTLRHashString.charAt	19	1,79%	61,89%
10	antlr/ANTLRHashString.hashCode	17	1,60%	63,49%
11	antlr/collections/impl/BitSet.member	16	1,51%	65,00%
12	antlr/CharScanner.append	14	1,32%	66,32%
13	java/lang/String.charAt	14	1,32%	67,64%
14	antlr/CharScanner.matchNot	12	1,13%	68,77%
15	org/antlr/stringtemplate/language/ActionLexer.mANONYMOUS_TEMPLATE	11	1,04%	69,81%
16	antlr/TokenBuffer.LA	11	1,04%	70,85%
17	org/antlr/stringtemplate/language/AngleBracketTemplateLexer.mLITERAL	10	0,94%	71,79%
18	antlr/CharQueue.append	9	0,85%	72,64%
19	antlr/ANTLRStringBuffer.append	8	0,75%	73,40%
20	antlr/TokenBuffer.fill	8	0,75%	74,15%
21	java/io/StringReader.isClosed	7	0,66%	74,81%
22	antlr/CharQueue.removeFirst	7	0,66%	75,47%
23	antlr/InputBuffer.consume	6	0,57%	76,04%
24	org/antlr/stringtemplate/language/AngleBracketTemplateLexer.mEXPR	6	0,57%	76,60%
25	antlr/collections/impl/BitSet.bitMask	6	0,57%	77,17%
26	antlr/LLkParser.LA	5	0,47%	77,64%
27	antlr/Token.getType	5	0,47%	78,11%
28	antlr/TokenBuffer.syncConsume	5	0,47%	78,58%
29	org/antlr/stringtemplate/language/AngleBracketTemplateLexer.mSUBTEMPLATE	5	0,47%	79,06%
30	antlr/collections/impl/BitSet.wordNumber	4	0,38%	79,43%
31	antlr/CharScanner.matchRange	4	0,38%	79,81%
32	antlr/TokenQueue.elementAt	4	0,38%	80,19%

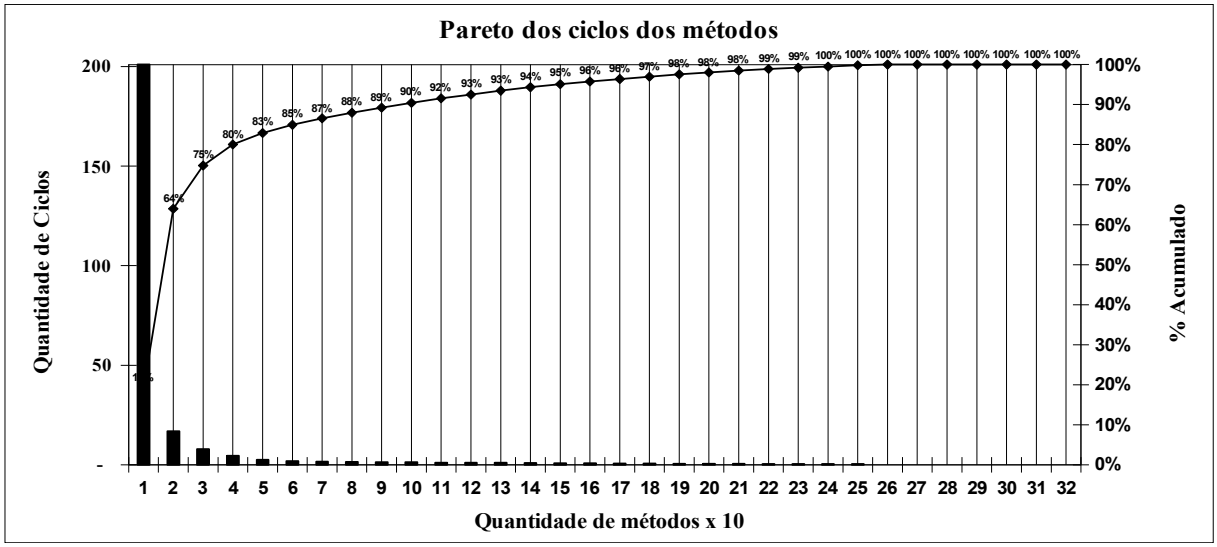

Figura 44: Pareto dos cilos dos métodos do compilador *inSOA*.

Como podemos perceber, no *pareto* dos ciclos do compilador *inSOA,* apresentados na figura 44, 10% dos aproximadamente 320 métodos da aplicação equivalem a 80% do custo de processamento da aplicação. Se analisarmos detalhadamente os métodos da tabela 26, podemos concluir que estes métodos são em sua maioria utilizados para quebrar os códigos em *tokens*.

Os tempos médios em segundos da compilação de cada composição *inSOA,* bem como, as médias de caracteres e *tokens* de cada composição, são apresentados na tabela 27. Assim como a quantidade de ciclos, os tempos médios foram obtidos a partir da replicação da compilação de cada uma das composições dez vezes com diferentes parâmetros. Como pode-se perceber, o tempo médio global de compilação foi 1,29 segundos para uma composição com 793 caracteres e 97 *tokens* em média.

Tabela 27: Tempos médios de execução das composições *inSOA*.

Identificação			Medidas		Tempo médio total em segundos
EC	Id	Composição	Caracteres	Tokens	
1	1	*getInformationTravel*	1323	219	1,285
1	11	*getInformationCountry*	1333	142	1,266
1	12	*getAirportInformation*	375	46	1,111
1	13	*validateAddress*	770	89	1,400
1	111	*getInformationWeather*	620	80	1,143
1	112	*getInformationGMT*	341	52	1,108
1	113	*getInformationMoney*	675	85	1,146
1	114	*getInformationUK*	630	90	1,160
1	131	*validateUS*	390	59	1,272
1	132	*validateUK*	414	60	1,308
2	1	*getBooks*	2084	219	1,615
2	11	*getInformationBooks*	908	107	1,372
2	12	*validateCredentials*	888	98	1,412
2	13	*validateFinancial*	1500	156	1,481
2	111	*getBook*	1095	131	1,388
2	112	*sumValuesBooks*	574	76	1,313
2	121	*validateCardInformation*	379	47	1,261
2	122	*validatePersonalInformation*	758	91	1,321
2	131	*validateCPF*	498	63	1,244
2	132	*validateCheck*	886	103	1,399
2	133	*validateLawsuits*	471	63	1,252
2	1222	*validateInformationSERASA*	524	65	1,199

Identificação			Medidas		Tempo médio total em segundos
EC	Id	Composição	Caracteres	Tokens	
Média Global			793	97	1,293

Obtido o tempo médio de compilação de cada composição com suas medidas de quantidade média de caractere e *token*, foi possível desenvolver um meta-modelo do compilador *inSOA*. Meta-modelo é uma fórmula matemática gerada a partir da análise de regressão linear. Análise de regressão linear é uma técnica estatística que permite explorar e inferir a relação de uma variável dependente, chamada de variável resposta ou de desempenho, com variáveis independentes específicas, chamadas de variáveis de prognóstico.

Para gerar o meta-modelo foi utilizada a ferramenta *MiniTab*[20]. O *MiniTab* é um programa voltado para fins estatísticos. Utilizando os dados de medidas de caracteres e *tokens* como variáveis de prognóstico e o tempo médio de compilação como variável resultado, obteve-se o seguinte meta-modelo: ***TM = 1.15 + 0.000408 * Caracteres - 0.00185 * Tokens***. Onde, *TM* é o tempo médio total, enquanto caracteres e *tokens* são as medidas discutidas anteriormente.

Regression Analysis: Total versus Caracteres; Tokens

The regression equation is
Total = 1.15 + 0.000408 Caracteres - 0.00185 Tokens

Predictor	Coef	SE Coef	T	P
Constant	1.14990	0.04039	28.47	0.000
Caracteres	0.0004081	0.0001270	3.21	0.005
Tokens	-0.001849	0.001141	-1.62	0.121

S = 0.0834247 R-Sq = 60.8% R-Sq(adj) = 56.6%

Analysis of Variance

Source	DF	SS	MS	F	P
Regression	2	0.20489	0.10245	14.72	0.000
Residual Error	19	0.13223	0.00696		
Total	21	0.33713			

Source	DF	Seq SS
Caracteres	1	0.18660
Tokens	1	0.01829

Unusual Observations

Obs	Caracteres	Total	Fit	SE Fit	Residual	St Resid
1	1323	1.2846	1.2849	0.0802	-0.0003	-0.01 X
2	1333	1.2657	1.4313	0.0317	-0.1656	-2.15R
11	2084	1.6154	1.5955	0.0584	0.0199	0.33 X

Figura 45: Análise de regressão linear.

20 *Minitab - http://www.minitab.com/*

Depois de criado o meta-modelo no *MiniTab*, apresentado detalhadamente na figura 45, gerou-se na mesma ferramenta um gráfico de probabilidade normal. Este gráfico tem por objetivo checar a adequação da fórmula em relação aos dados gerados através da simulação. Quanto mais próximo os pontos de simulação encontram-se da reta resultante da fórmula do meta-modelo, mais aderência tem o meta-modelo com à simulação.

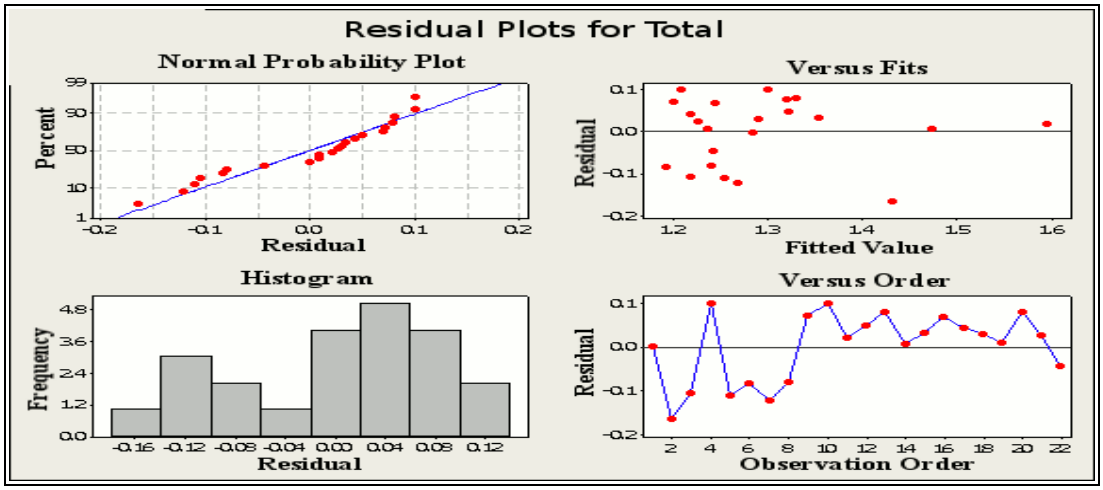

Figura 46: Gráfico de probabilidade normal.

Analisando o gráfico da figura 46 pode-se concluir que a fórmula do meta-modelo é aderente à simulação, pois a maioria dos pontos está próximo às linhas das retas geradas pelo meta-modelo. Também é possível chegar a esta conclusão a partir do *R2* que é 60,8%. Isto quer dizer que o meta-modelo acertará o resultado em 60,8% das vezes. Este percentual, apesar de aceitável por indicar uma correlação moderada, pode ser melhorado se o meta-modelo for gerado com maior número de composições.

O meta-modelo do compilador *inSOA* é útil para determinar o tempo em que uma compilação irá levar antes de essa ser iniciada. Ele pode ser utilizado também como parte do cálculo de uma analisador de impacto, tal como, o *gImpact*.

Na tabela 28 são apresentados os tempos médios de execução das composições no motor *SmallSOA*. Nesta tabela são listados apenas os tempos médios totais das composições principais de cada cenário do estudo de caso. Estes tempos incluem a execução de todos os *WS* e composições em cascata.

Tabela 28: Tempo de execução das composições no motor *SmallSOA*.

#	Cenário I – getInformationTravel			Cenário II - getBooks		
	Horário Inicial	Horário Final	Tempo de Execução	Horário Inicial	Horário Final	Tempo de Execução
1	01:25:11,03	01:25:36,25	25,230	03:58:33,23	03:58:55,58	22,350
2	01:25:48,54	01:26:23,17	34,630	03:59:04,48	03:59:27,96	23,480
3	01:26:32,90	01:26:59,39	26,490	03:59:33,77	03:59:59,98	26,210
4	01:29:01,41	01:29:28,17	26,760	04:00:27,76	04:01:03,96	36,200
5	01:29:46,32	01:30:10,74	24,420	04:01:14,31	04:01:51,62	37,310
6	01:31:13,02	01:31:35,54	22,520	04:02:06,41	04:02:30,65	24,240
7	01:32:32,42	01:33:01,28	28,857	04:03:34,64	04:04:07,28	32,640
8	01:33:54,88	01:34:28,83	33,951	04:05:06,37	04:05:33,82	27,451
9	01:34:36,26	01:35:01,86	25,600	04:05:36,25	04:06:01,54	25,290
10	01:35:03,94	01:35:33,40	29,460	04:06:11,08	04:06:37,08	25,993
Média de Tempo de Execução			**27,792**			**28,116**

Na tabela 29 é apresentada uma comparação do tamanho das composições desenvolvidas em *BPEL* e *inSOA*. As composições utilizadas nesta comparação foram extraídas dos exemplos fornecidos com o servidor de composição *ActiveVOS*[21].

Tabela 29: Comparação de composições *BPEL* e *inSOA*.

#	Composição	*BPEL*		*inSOA*		% (1 - *inSOA* / *BPEL*)	
		Caracteres	Palavras	Caracteres	Palavras	Caracteres	Palavras
1	*ForEach*	5370	748	262	39	95,12%	94,79%
2	*IsolatedScope*	3000	406	231	40	92,30%	90,15%
3	*MultiStartReceives*	5703	795	374	61	93,44%	92,33%
4	*RepeatUntil*	4444	520	225	29	94,94%	94,42%
5	*While*	4698	656	173	26	96,32%	96,04%
Médias Totais		**4643**	**625**	**253**	**39**	**94,55%**	**93,76%**

Como pode-se perceber na tabela 29, o *inSOA* na médias das composições analisadas, é 94,55% caracteres e 93,76% palavras menor que a linguagem *BPEL*. Este percentual justifica-se porque a linguagem *inSOA* não possui *tags* e declarações de *namespaces*, tal como, a linguagem *BPEL,* que é baseada em *XML*. Além disso, a linguagem *inSOA* é mais leve e direta na declaração de composições e, possui estruturas *xPath* que facilitam na navegação dos resultados.

21 ***ActiveVOS*** - *http://www.activevos.com/*

5.2.2 *Patterns* e Trabalhos Relacionados

No desenvolvimento dos cenários do estudo de caso foram testados alguns *patterns* de composição de serviços implementados na linguagem *inSOA*. Na tabela 30 são apresentados os *patterns* que foram testados no estudo de caso. Apesar dos *patterns* de composição terem sido implementados quase na sua totalidade, como discutido anteriormente no capítulo 4, nem todos os *patterns* foram testados nos cenários do estudo de caso. Isto ocorreu devido a falta de implementação destes *patterns* no motor de composição ou porque não encontrou-se um cenário real que pudesse contemplar todos os *patterns*.

Tabela 30: *Patterns* testados nos cenários do estudo de caso.

#	*Pattern*	Implementado?	Testado?
1	*Sequence*	Sim	Sim
2	*Parallel Split*	Sim	Sim
3	*Synchronization*	Sim	Sim
4	*Exclusive Choice*	Sim	Não
5	*Simple Merge*	Sim	Sim
6	*Multi-choice*	Sim	Não
7	*Synchronizing Merge*	Sim	Sim
8	*Multi-merge*	Sim	Sim
9	*Discriminator*	Sim	Não
10	*Arbitrary Cycles*	Sim	Não
11	*Implicit Termination*	Sim	Sim
12	*Multiple Instances Without Synchronization*	Sim	Não
13	*Multiple Instances With a Priori Design Time Knowledge*	Sim	Não
14	*Multiple Instances With a Priori Runtime Knowledge*	Sim	Não
15	*Multiple Instances Without a Priori Runtime Knowledge*	Sim	Não
16	*Deferred Choice*	Sim	Não
17	*Interleaved Parallel*	Sim	Não
18	*Milestone*	Não	Não
19	*Cancel Activity*	Sim	Sim
20	*Cancel Case*	Sim	Sim
21	*Exception Handling*	Sim	Não

Por fim, na tabela 31 é apresentado um comparativo dos aspectos dos trabalhos relacionados com a linguagem *inSOA*. Apesar da linguagem *CBPEL* estar classificada como uma linguagem do tipo declarativa, ela se diferencia da *inSOA* neste aspecto, por ter seus códigos escritos utilizando o padrão *XML*, enquanto a *inSOA* é desenvolvida em texto livre com padrão próprio, permitindo com isto um desenvolvimento mais leve e simples. Outro

ponto a ser observado é que o ambiente de execução destas linguagens são o *desktop* enquanto a *inSOA* é a plataforma *Android*.

Tabela 31: Comparativo com os trabalhos relacionados.

Linguagem	Ano	Tipo	Autoria?	Embutida?	Ambiente	Linguagem
CLAM	2000	Imperativa	Não	Não	*Desktop*	*CHAIMS*
PICCOLA	2001	Imperativa	Não	Não	*Desktop*	*Java*
CBEL	2005	Declarativa	Sim	Não	*Desktop*	*XML*
inSOA	2009	Declarativa	Não	Sim	*Android*	*Java*

6 CONCLUSÃO

Desde o início da *web* até os dias atuais o principal objetivo da *internet* tem sido a colaboração entre as pessoas. Esta colaboração, mais recentemente, tem sido estendida para troca de dados e informações entre aplicativos. A mais promissora das tecnologias para suportar estas trocas é a arquitetura *SOA*, que é baseada em *WS* e inspirada na orquestração e composição de serviços. Por sua vez, a composição de serviços realizada por linguagens de composição de serviços, têm estimulado ainda mais a colaboração e aproveitamento de serviços na *web*. Serviços esses, que estão crescendo e se popularizando na *internet* na mesma velocidade em que os dispositivos móveis vêm substituindo os dispositivos de mesa.

Este trabalho apresentou *inSOA*, uma linguagem declarativa de composição de serviços desenvolvida para rodar em dispositivos móveis. Inicialmente foi realizada uma revisão bibliográfica das principais tecnologias e trabalhos relacionados, seguido pela apresentação da linguagem *inSOA* e finalizada com um estudo de caso abrangendo dois cenários.

O estudo das tecnologias relacionadas, possibilitou refletir sobre a relação e caráter evolutivo entre as tecnologias *web, XML*, *WS, XPath* e *SOA,* além de evidenciar as oportunidades existentes na composição de serviços. A revisão do conceito de composição de serviços levou ao entendimento mais aprofundado das diferentes abordagens de linguagens de composição, que foram demonstradas no estudo das linguagens *BPEL* e *OWL-S.* Estudo esse, que permitiu o entendimento dos principais *patterns* de composição de serviços, além do funcionamento das linguagens de programação e da definição destas via *BNF*. Complementando o aprendizado, foi realizado ainda, uma análise das principais plataformas de desenvolvimento para dispositivos móveis: *.Net*, *JME* e *Android*.

No estudo dos trabalhos relacionados aprofundou-se em algumas linguagens de composição de serviços. As linguagens apresentadas, além de possuírem características distintas foram concebidas com diferentes abordagens. A *CBPEL* é uma extensão da *BPEL* com suporte a processos de negócios interorganizacionais. A *PICCOLA* é focada em composição de componentes. Enquanto a *CLAM*, é focada em um conceito definido como mega-módulo que pode ser escrito em diferentes linguagens.

Na apresentação da linguagem *inSOA* foi possível demonstrar não só a definição detalhada da linguagem e sua inserção no projeto *U-SOA,* bem como, uma comparação das

características da *inSOA* e os outros paradigmas de composição existentes no mercado, conforme reapresentação na tabela 32. Discutiu-se ainda, a preocupação da linguagem *inSOA* com os patterns de composição de serviços, os quais foram detalhados na tabela 19, onde se verificou que a maioria dos padrões foram atendidos. Além disso, foi apresentado a análise, o projeto e o desenvolvimento da linguagem *inSOA*. Desenvolvimento este, que incluiu um protótipo do compilador *inSOA* que atendeu todos os componentes propostos pelo trabalho: gramática da linguagem, análise léxica e parser, análise semântica e otimizador, e interface *XML* otimizada de integração com o motor de composição *SmallSOA* e o analisador de impacto *gImpact*, além do tratamento de erros e mensagens do compilador.

Tabela 32: Comparação das características das linguagens de composição.

#	Característica	*BPEL*	*OWL-S*	*inSOA*
a	**Insensível a capitalização**	NÃO	NÃO	SIM
b	**Declaração Flexível**	NÃO	NÃO	SIM
c	**Validação de Padrões**	NÃO	NÃO	SIM
d	**Tratamento *XML***	Externo	Externo	Interno
e	**Opcionalidade**	NÃO	NÃO	SIM
f	**Tratamento de Erros**	SIM	SIM	SIM
g	**Orquestração**	SIM	SIM	SIM
h	**Saída Customizável**	NÃO	NÃO	SIM
i	**Invocação Otimizada**	NÃO	NÃO	SIM
j	**Embutida**	NÃO	NÃO	SIM
k	**Leve**	NÃO	NÃO	SIM
l	**Abordagem**	Funcional	Orientada para Objetos	Declarativa
m	**Baseada**	XML	XML	Texto

Por fim, realizou-se um estudo de caso com dois cenários: consulta de viagem e consulta de livros. Este estudo de caso possibilitou avaliar as funcionalidades da linguagem e compilador *inSOA*. Ele foi feito de forma integrada com o motor de composição *SmallSOA* e o analisador de impacto *gImpact*. O estudo de caso permitiu também avaliar os tempos e performance do compilador *inSOA,* onde inclusive gerou-se o meta-modelo dos tempos médios de execução do processo de compilação. Além disso, foi apresentada a relação de *patterns* testados no estudo de caso e desenvolvido um comparativo com os trabalhos relacionados.

De forma complementar, pode-se dizer que a linguagem *inSOA* propõe um paradigma declarativo diferenciado das outras linguagens de composição estudados ao apresentar uma forma de edição textual e leve, podendo inclusive no futuro ser embutida em outras linguagens de propósito geral.

6.1 Contribuições

O término da construção da linguagem *inSOA* permite elencar as contribuições proporcionadas por ela. As principais delas são destacadas a seguir:

- Componente e linguagem esperada pelo projeto *U-SOA*.
- Protótipo de compilador funcional que roda sobre a plataforma *Android*.
- Gerador de *XML* otimizado da composição *inSOA* para integração com outras ferramentas.
- Linguagem declarativa específica para o domínio de orquestração e composição de serviços.
- Interface com outros componentes e ferramentas através de padrões da indústria baseados em *XML*.
- Desenvolvimento de composição de serviços *inSOA* com qualquer editor de texto.
- Tratamento dos retornos dos *WS* diretamente na linguagem com o *XPath* embutido.

6.2 Limitações

A *inSOA* é uma linguagem declarativa de composição de serviços. Devido a arquitetura e o objetivos propostos, algumas limitações são identificadas:

- O *pattern* 18[22] discutido na seção 4.3 não foi implementado na linguagem.
- Falta de tratamento de objetos complexos. Apesar de previstos com o tipo *base64*, não existem elementos na linguagem para sua manipulação e uso.
- Implementação de comunicação real entre celulares baseados na plataforma *Android*.
- Os *WS* declarados no comando *invoke* não suportam *WS REST*.

6.3 Trabalhos Futuros

Para complementação do trabalho, podem ser considerados como possíveis trabalhos futuros:

22 *Pattern 18 - Milestone*

- Executar *inSOA* como um *driver* de composição de serviços, não necessitando assim de um motor de execução.
- Desenvolver *bytecodes Java*, ao invés de *XML*, para ganhar em performance.
- Embutir comandos ou subcomandos na linguagem com mecanismos de suporte à segurança e autenticação, por exemplo, *WS-Security*.
- Colocar *hints*, tipo *Oracle*[23], para lidar com *QoS*, integrando esses aos custos do *gImpact*.
- Definir e implementar um sub-comando no comando *invoke* para suportar *WS REST*.
- Desenvolver uma ferramenta de edição de *scripts*, colocando os seguintes recursos: visualização gráfica do retorno da composição como os *parsers XML* dos navegadores, desenvolvimento visual de *scripts inSOA* como os disponíveis em ferramentas de desenvolvimento *SQL* e *profiling* com o *gImpact.*
- Comparação mais detalhada da *inSOA* com as linguagens *BPEL* e *OWL-S*.
- Implementar a comunicação *REST* com os *WS*.

23 *Oracle – http://www.oracle.com/*

7 REFERÊNCIAS BIBLIOGRÁFICAS

[Aalst 2003] Aalst, W.M.P van der; Dumas, M. e Hofstede, A.H.M. Ter. **Web service composition languages: old wine in New bottles?**. Euromicro Conference. IEEE. ISBN: 0-7695-1996-2. 2003.

[Aho 2006] Aho, Alfred V.; Lam, Monica S.; Sethi, Ravi e Ullman, Jeffrey D. **Compilers: Principles, Techniques, & Tools**. Pearson Addison Wesley. 1000 Pág. ISBN: 978-0321486813. 2006.

[Android 2008] Android Google. **What is Android?**. http://code.google.com/android/what-is-android.html. Dez/08.

[ANTLRWorks 2008] ANTLRWorks. **The ANTLR GUI Development Environment.** http:// www.antlr.org/works/index.html. Dez/08.

[Appel 2004] Appel, W. Andrew. **Modern Compiler Implementation in Java. 2nd Edition.** Cambridge. 501 Pág. ISBN: 978-0521820608. 2004.

[Archemann 2001] Archemann, Franz; Lumpe, Markus; Schneider, Jean-Guy e Nierstrasz, Oscar. **PICCOLA – a Small Composition Language**. Cambridge University Press. ISBN: 3-540-66954-X. 2001.

[Balani 2003A] Balani, Naveen. **Using kXML to access XML files on J2ME devices**. https:// www6.software.ibm.com/developerworks/education/wi-kxml/wi-kxml-a4.pdf. Dez/08.

[Balani 2003B] Balani, Naveen. **Deliver Web services to mobile apps**. https://www6.software.ibm.com/developerworks/education/wi-wsvs/wi-wsvs-a4.pdf. Dez/08.

[Berners-Lee 1989] Berners-Lee, T. **Information Management: A Proposal. CERN, March 1989, The Original Document File**. http://www.w3.org/History/1989/proposal.html. Dez/08.

[Berners-Lee 2001] Berners-Lee, T.; Hendler, J. e Lassila, O. **The Semantic Web**. Scientific American. 2001.

[Buschmann 1996] Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter e Stal, Michael. **Pattern – Oriented Software Architecture - A System of Patterns**. John Wiley & Sons. 476 Pág. ISBN: 978-0471958697. 1996.

[Cerani 2002] Cerani, Ethan. **Web Services Essentials: Distributed Applications with XML-RPC, SOAP, UDDI & WSDL**. O'Reilly. 304 Pág. ISBN: 978-8173663390. 2002.

[Coyle 2002] Coyle, Frank P.. **XML, Web Services, and the Data Revolution**. Addison Wesley. 400 Pág. ISBN: 978-0201776416. 2002.

[Crespo 2000] Crespo, Sérgio. **Composição em WebFrameworks**. Tese de Doutorado. PUC-RIO. 2000.

[Denzin 2000] Denzin, Norman K. e Lincoln, Yvonna. **Case Studies: Handbook of Qualitative Research.** Sage Publications. 2nd Edition. 1143 Pág. ISBN: 978-0761915126. 2000.

[Dykes 2005] Dykes, Lucinda e Tittel, Ed. **XML for Dummies. 4th Edition**. Wiley Publishing. 384 Pág. ISBN: 978-0-7645-8845-7. 2005.

[Erl 2005] Erl, Thomas. **Service-Oriented Architecture: Concepts, Technology, and Design**. Prentice Hall PTR. 792 Pág. ISBN: 978-0131858589. 2005.

[Fielding 2000] Fielding, Roy Thomas. **Architectural Styles and the Design of Network-based Software Architectures.** Doctoral dissertation. University of California. Irvine, 2000.

[Friedman 2008] Friedman, Daniel P. e Wand, Mitchell. **Essentials of Programming Languages**. MIT Press. 410 Pág. 2008.

[Gammar 1994] Gammar, E.; Helm, R.; Johnson, R. e Vlissides, J.M. **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley. 416 Pág. ISBN: 978-0201633610. 1994.

[Graham 2002] Graham, S.; Simeonov, S.; Boubez, T.; Davis, D.; Daniels, G.; Yuichi, N. e Neyama, R.. **Building Web Services With Java**. Pearson Education. 450 Pág. ISBN: 978-0672321818. 2002.

[Hackmann 2006] Hackmann, G.; Haitjema, M.; Gill, C. e Roman, G.-C. Sliver: **A BPEL Workflow Process Execution Engine for Mobile Devices**. Washington University, Department of Computer Science and Engineering, St. Louis, Missouri. (published in Proceedings of 4th International Conference on Service Oriented Computing (ICSOC 2006)) .

[Hamilton 2006] Hamilton, Kim e Miles, Russel. **Learning UML 2.0.** O'Reilly. 286 Pág. ISBN: 978-0596009823. 2006.

[Hansen 2003] Hansen, R. **Gluescript: uma linguagem específica de domínio para composição de web services**. UNISINOS. 2003.

[Harold 2002] Harold, Elliotte R.; Means, W. Scott. **XML in a Nutshell. 2nd Edition**. O'Reilly. 634 Pág. ISBN: 9780596002923. 2002.

[Haseman 2008] Haseman, Chris. **Android Essentials**. Apress. 166 Pág. ISBN: 978-1430210641. 2008.

[Havey 2005] Havey, Michael. **What Is Business Process Modeling**. http://www.onjava.com/lpt/a/6041. Dez/08.

[Holzner 2003] Holzner, Steven. **Sams Teach YourSelf XML in 21 Days**. Sams Publishing. 888 Pág. 2003.

[IBM 2007] IBM. **Site oficial da IBM. Arquitetura Orientada a Serviço (SOA).** http://www-306.ibm.com/software/br/info/topic/openenvironment/soa/. Dez/08.

[IBM 2008] IBM. **Go Mobile, Grow.** http://www-935.ibm.com/services/us/gbs/bus/pdf/gbe03051-usen-03_gomobile.pdf. Jan/09.

[Juric 2007] Juric, B. Matjaz; Loganathan, Ramesh; Sarang, Poornachandra e Jennings, Frank. **SOA Approach to Integration: XML, Web services, ESB, and BPEL in real-world SOA projects**. Packt Publishing. 382 Pág. ISBN: 978-1904811176. 2007.

[Kalasapur 2005] Kalasapur, Swaroop; Kumar, Mohan e Shirazi, Behrooz. **Seamless Service Composition (SeSCo) in Pervasive Environments**. MSC - Multimedia service composition. Singapore. 2005.

[Kelly 2008] Kelly, Steven e Juha-Pekka, Tolvanen. **Domain-Specific Modeling: Enabling Full Code Generation**. Wiley-IEEE Computer Society Pr. 427 Pág. ISBN: 978-0470036662. 2008.

[Leymann 2001] Leymann, F.. **Web Services Flow Language (WSFL 1.0).** IBM, May 2001. http://xml.coverpages.org/WSFL-Guide-200110.pdf. Dez/08

[Martin 2004] Martin, D. at al. **OWL-S: Semantic Markup for Web Services**. W3C. http://www.w3.org/Submission/OWL-S/. 2004.

[Metsker 2002] Metsker, Steven John. **Building Parsers with Java**. Addison Wesley. 400 Pág. ISBN: 978-0201719628. 2002.

[Microsoft 2006] Microsoft. **Criando web services Seguros**. http://www.microsoft.com/brasil/security/guidance/topics/devsec/secmod85.mspx. Dez/08.

[MSDN 2008] Microsoft. **.NET Compact Framework**. http://msdn2.microsoft.com/en-us/library/ms950380.aspx. Dez/08.

[Mühlen 2002] Mühlen, M. zur. **Workflow-based Process Controlling. Foundation, Design, and Application of Workflow-driven Process Information Systems.** Logos Verlag Berlin. 315 Pág. ISBN: 978-3832503888. 2002.

[Nakamura 2004] Nakamura, Masahide; Igaki, Hiroshi; Tamada, Haruaki e Matsumoto, Ken-Ichi. **Implementing Integrated Services of Networked Home Appliances using service oriented Architecture**. 2nd International Conference on Service Oriented Computing. 2004.

[Naur 1960] Naur, Peter. **Revised Report on the Algorithmic Language ALGOL 60**. Communications of the ACM, Vol. 3 No.5, pp. 299-314. 1960.

[Newcomer 2002] Newcomer, E. **Understanding web services independent technology guide**. Series Editor. 2002.

[Oscar 2005] Oscar, Mauricio; Rendón, Caicedo; Pábon, F.O.M; Vargas, M.J.G e Guacal, J.A.H.. **Architectures for Web Services Access from Mobile Devices**. IEEE. ISBN: 0-7695-2471-0. 2005.

[Parr 2007] Parr, Terence. **The Definitive Antlr Reference: Building Domain-Specific Languages.** Pragmatic Bookshelf. 376 pag. ISBN: 978-0978739256. 2007.

[Pautasso 2008] Pautasso, Cesare; Zimmermann, Olaf e Leymann, Frank. **Restful web services vs. "big"' web services: making the right architectural decision.** International WWW Conference. Pág. 805-814. ISBN: 978-1-60558-085-2.

[Pijanowski 2007] Pijanowski, Keith. V**isibility and Control in a Service-Oriented Architecture. MSDN Architecture Center, Technical Articles.** http://msdn2.microsoft.com/en-us/library/bb507204.aspx. Dez/08.

[Rosen 2008] Rosen, Mike.; Lublinsky, Boris; Smith, T. Kevin e Balcer, Marc J.. **Applied SOA: Service-Oriented Architecture and Design Strategies.** Wiley. 698 Pág. ISBN: 978-0470223659. 2008.

[Sample 2000] Sample, Neal; Beringer, Dorothea; Melloul, Laurence e Wiederhold, Gio. **CLAM: Composition Language for Autonomous Megamodules.** Springer Berlin/Heidelberg. 2000.

[Scopel 2005] Scopel, Marcelo. **WSMEL uma arquitetura para integração de serviços educacionais usando dispositivos móveis na formação de comunidades virtuais espontâneas.** UNISINOS. 2005.

[Sebesta 2006] Sebesta, Robert W. Concepts of Programming Languages. Pearson Education. 744 Pág. ISBN: 9780321330253. 2006.

[Sharpe 2007] Sharpe, Ben M.J.. SOA for the Business Developer: Concepts, BPEL, and SCA. MC Press. 328 Pág. ISBN: 978-1583470657. 2007.

[Shaw 2008] Shaw, John e Evans, Simon. **Pro ADO.NET Data Services: Working with RESTful Data.** Apress. 336 Pág. ISBN: 978-1430216148.

[Skonnard 2002] Skonnard, Aaron e Gudgin, Martin. **Essential XML. Quick Reference: A Programmer's Reference to XML, XPath, XSLT, XML Schema, SOAP, and More.** Addison-Wesley Professional. 432 Pág. ISBN: 978-0201740950. 2002.

[Srirama 2006] Srirama, S; Jarke, M. e Prinz, W. **Mobile Host: A feasibility analysis of mobile Web Service provisioning.** Proceedings of the CAISE*06 Workshop on Ubiquitous Mobile Information and Collaboration Systems UMICS '06, 2006.

[Staab 2003] Staab, S.; van der Aalst, W.; Benjamins, V.R.; Sheth, A.; Miller, J.A.; Bussler, C.; Maedche, A.; Fensel, D. e Gannon, D.. **Web services: been there, done that?.** IEEE, DOI: 10.1109/MIS.2003.1179197. 2003.

[StringTemplate 2008] StringTemplate. **StringTemplate cheat sheet**. http://www.stringtemplate.org/. 2008.

[SUN 2008] SUN. **Java Technology.** http://java.sun.com/javame/technology/index.jsp. Dez/08.

[Teófilo 2005] Teófilo, António e Silva, Alberto R.. C**BPEL – Linguagem para definição de processos de negócio interorganizacionais.** 3a. Conferência de XML: Aplicações e Tecnologias Associadas (XATA'2005). 2005.

[Terry 1996] Terry, P.D.. Compilers and Compiler Generators – An Introduction with C++. Rhodes University. 435 Pág. 1996.

[Thatte 2001] Thatte, S.. **XLANG Web Services for Business Process Design.** http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm. 2001.

[W3C 2003] W3C. **XML Path Language (Xpath) 2.0.** W3C Working Draft. http://www.w3.org/TR/xpath20/. 2003.

[W3C 2004] W3C. **Web Services Architecture.** http://www.w3.org/TR/ws-arch/#introduction. Dez/08.

[Weerawarana 2005] Weerawarana, Sanjiva; Curbera, Francisco; Leymann, Frank; Storey, Tony e Ferguson, Donald F. **Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More.** Prentice Hall PTR. 456 Pág. ISBN: 978-0131488748. 2005.

[Will 2002] Will, van der Aalst; A.H.M. ter Hofstede; B. Kiepuszewski e A.P. Barros. **Workflow Patterns.** QUT Technical report. FIT-TR-2002-02. Queensland University of Technology. Brisbane. 2002.

[Will 2003] Will, van der Aalst. **Don't go with the flow: Web services composition standards exposed.** IEEE Intelligent Systems. 2003.

[Wohed 2002] Wohed, P.W.M.P. van der Aalst, M. Dumas e A.H.M. ter Hofstede (2002). **Pattern Based Analysis of BPEL4WS.** QUT Technical report, FIT-TR-2002-04, Queensland University of Technology, Brisbane, 2002. http://citeseer.ist.psu.edu/556822.html.

[Woods 2006] Woods, Tom e Mattern, Thomas. **Enterprise SOA: Designing IT for Business Innovation.** O'Reilly. 452 pág. ISBN: 978-0596102388. 2006.

[Zanuz, Barcelos et al. 2008A] Zanuz, L. ; Barcelos, G. ; Filippetto, A. ; Pinto, S. C. C. S. U-SOA - **Towards a Ubiquitous Platform Based on Service - Oriented Architecture.** XIV Simpósio Brasileiro de Sistemas Multimídia e Web (WebMedia). Vila Velha. 2008.

[Zanuz, Barcelos et al. 2008B] Zanuz, L.; Filippetto, A.; Barcelos, G. ; Pinto, S. C. C. S. **SOA Engine: Services Compositions Execution in Ubiquitous Environments.** XIV Simpósio Brasileiro de Sistemas Multimídia e Web (WebMedia). Vila Velha. 2008.

ANEXO I – Gramática da Linguagem *inSOA* com *ANTLR* (sem semântica)

Pacote *inSOA*

```
grammar inSOA;
import inSOAXPath, inSOAURI, inSOALexerRules;

invoke: (invokeAux
   ('id' id)?
   ('tags' tags)?
   (';')? );
invokeAux: ( 'invoke' uri
   ('input' input)?
   ('into' into)?
   ('set' set)?
   ('where' where)?
   ('return' back)?
   ('fail' fail)?);
uri: (uriReference ('namespace' nameSpace)? 'as' VarName ('if' iif)? (comma uriReference ('namespace' nameSpace)? 'as' VarName ('if' iif)?)*);
nameSpace: uriReference|VarName;
iif: (iifTerm (andOr iifTerm)*);
iifTerm: (('not')? iifFact (inComp  ('not')? iifFact)*);
iifFact: term (operFact term)*;
into: (xPath  'as' VarName (comma xPath 'as' VarName)*);
set: setOne (comma setOne)*;
setOne: (ParamTerm assign setExpression);
setExpression: term (operTerm term)*;
term: ( pathTerm  | literal );
where: (whereTerm (andOr whereTerm)*);
whereTerm: (('not')? whereFact (inComp  ('not')? whereFact)*);
whereFact: term (operFact term)*;
back:  backTerm ('type' VarName)? ('name' VarName)? (comma backTerm  ('type' VarName)? ('name' VarName)?)* ;
backTerm: term (operTerm term)*;
fail: ( failVar invokeAux )+;
failVar: VarFail  (comma VarFail)*;
input: (VarName  ('type' VarName)? ('default' literal)? (comma VarName  ('type' VarName)? ('default' literal)?)*);
id: VarName;
tags: VarName (comma VarName)*;
pathTerm:(VarName) => ( (VarName xPath) => VarName xPath) | VarName;
operFact:   ('+'|'-'|'*'|'/');
operTerm: ('+'|'-'|'*'|'/'|'||');
comma: ',';
assign: ':=';
andOr: ('and'|'or');
inComp: '==' | '!=' | '<' | '<=' | '>' | '>=';
```

Pacote regras *Léxicas*

```
lexer grammar inSOALexerRules;

// URI
Uri: (Scheme '://') Authority ('/' Pchar*)* ( '?' Query )? ( '#' Query )?;
fragment Scheme: CHAR ( CHAR | DIGIT | '+' | '-' | '.' )*;
fragment Query: ( Pchar | '/' | '?' )*;
fragment Authority: ( Userinfo '@' )? RegName ( ':' ( DIGIT)+)?;
fragment Pchar: Unreserved | PctEncoded | SubDelims | ':' | '@';
fragment Userinfo: ( Unreserved | PctEncoded | SubDelims| ':' )*;
fragment Unreserved: CHAR | DIGIT | '-' | '.' |  '~';
fragment PctEncoded: '%' HexDigit HexDigit;
fragment SubDelims: '!' | '$' | '&' | '"' | '(' | ')' | '*' | '+' | ',' | ';' | '=';
fragment RegName: ( Unreserved | PctEncoded | SubDelims )*;
fragment HexDigit: (DIGIT| 'a'..'f' | 'A'..'F');

/*
//XPath
ValueComp: 'eq' | 'ne' | 'lt' | 'le' | 'gt' | 'ge'; // [23]
GeneralComp: '=' | '!=' | '<' | '<=' | '>' | '>='; // [22]
NodeComp: 'is' | '<<' | '>>'; // [24]
NcName: CHAR NcNameChar*; // [4] An XML name, minus the ":"
```

```
NcNameChar: CHAR | DIGIT | '.' ; // [5]
StringLiteral:  ('"' ~('"')* '"') | ('\'' ~('\'')* '\''); // [74] ws: explicit
CommentTest: 'comment' '(' ')'; // [58]
TextTest: 'text' '(' ')'; // [57]
AnyKindTest: 'node' '(' ')'; // [55]  */
StringLiteral:  ('"' ~('"')* '"') | ('\'' ~('\'')* '\''); // [74] ws: explicit
NumericLitOrContextItem: ('+'|'-')? (('.' DIGIT+) | (DIGIT+ ('.' DIGIT+)?)) (('e'| 'E') ('+' | '-') DIGIT+)?; // [73] ws: explicit
GeneralComp: '=' | '!=' | '<' | '<=' | '>' | '>='; // [22]
OcurrenceIndicator: '?' | '*' | '+'; // [51] xgs: occurrence-indicators

//General
VarFail: VarName ':';
VarName: (CHAR) (CHAR|DIGIT)*;
ParamTerm: (VarName '.' VarName);
COMMENT: '(:' ( options {greedy=false;} : . )* ':)' {$channel=HIDDEN;}; // [77] ws: explicit
WS  :  ( '\t' | '\u0020' | '\r' | '\n'| '\u000C' )+ {$channel=HIDDEN;};
SL_COMMENT: '--' (~('\n'|'\r'))* ('\n'|'\r'('\n')?) {$channel=HIDDEN;};
ML_COMMENT: '/**' (options {greedy=false;} : .)* '*/' {$channel=HIDDEN;};
fragment DIGIT: '0'..'9'; // 88
fragment CHAR: '_' | 'a'..'z' | 'A'..'Z' ; // [2]
```

Pacote *XPath*

```
parser grammar inSOAXPath;

xPath: expr; // [1]
expr: forexpr  | quantifiedExpr | ifExpr | orExpr; // [3]
forexpr: simpleForClause 'return' expr; // [4]
quantifiedExpr: ('some' | 'every') '$' qName 'in' expr (',' '$' qName 'in' expr)* 'satisfies' expr; // [6]
ifExpr: 'if' '(' expr ')' 'then' expr 'else' expr; // [7]
orExpr: andExpr ( 'or' andExpr )*; // [8]
simpleForClause: 'for' '$' qName 'in' expr (',' '$' qName 'in' expr)* ; // [5]
andExpr: comparisonExpr ( 'and' comparisonExpr )*; // [9]
qName: VarName (':' VarName)?; // [7]
comparisonExpr: rangeExpr ( (valueComp | GeneralComp | nodeComp) rangeExpr )?; // [10]
rangeExpr: additiveExpr ( 'to' additiveExpr )?; // [11]
additiveExpr: multiplicativeExpr ( ('add' | 'sub') multiplicativeExpr )*; // [12]
multiplicativeExpr: unionExpr ( ('mult' | 'div' | 'idiv' | 'mod') unionExpr )*; // [13]
unionExpr: intersectExceptExpr ( ('union' | '|') intersectExceptExpr )*; // [14]
intersectExceptExpr: instanceOfExpr ( ('intersect' | 'except') instanceOfExpr )*; // [15]
instanceOfExpr: treatExpr ( 'instance' 'of' sequenceType )?; // [16]
treatExpr: castAbleExpr ( 'treat' 'as' sequenceType )?; // [17]
castAbleExpr: castExpr ( 'castable' 'as' singleType )?; // [18]
sequenceType: ('empty-sequence' '(' ')')  (itemType OcurrenceIndicator?); // [50]
castExpr: pathExpr ( 'cast' 'as' singleType )?; // [19]
singleType: qName '?'?; // [49]
itemType: kindTest | ('item' '(' ')') | qName; // [52]
pathExpr:  ( '/'|'//') relativePathExpr? ;// | relativePathExpr; // [25] xgs: leading-lone-slash
relativePathExpr: stepExpr (('/' | '//') stepExpr)*; // [26]
stepExpr:   filterExpr | axisStep ; //[27]
filterExpr: primaryExpr predicate*; // [38]
axisStep: (reverseStep | forwardStep) predicate*; // [28]
primaryExpr: literal | varRef  | parenthesizedExpr | functionCall | '.'; // | contextItemExpr ; // [41]
reverseStep: (reverseAxis nodeTest) | abbrevReverseStep; // [32]
forwardStep: (forwardAxis nodeTest) | abbrevForwardStep; // [29]
literal: NumericLitOrContextItem | StringLiteral; // [42]
varRef: '$' qName; // [44]
parenthesizedExpr: '(' expr? ')'; // [46]
functionCall: qName '(' (expr (',' expr)*)? ')'; // [48] xgs: reserved-function-names; gn: parens
predicate: '[' expr ']'; // [40]
reverseAxis: ('parent' '::') | ('ancestor' '::') | ('preceding-sibling' '::') | ('preceding' '::') | ('ancestor-or-self' '::'); // [33]
nodeTest: kindTest | nameTest; // [35]
abbrevReverseStep: '..'; // [34]
forwardAxis: ('child' '::') | ('descendant' '::') | ('attribute' '::') | ('self' '::') | ('descendant-or-self' '::') | ('following-sibling' '::') | ('following' '::') |
('namespace' '::'); // [30]
abbrevForwardStep: ( '@')? nodeTest; // [31]
kindTest: documentTest | elementTest | attributeTest | schemaElementTest | schemaAttributeTest | piTest | commentTest | textTest |
anyKindTest; // [54]
nameTest: qName  | wildCard; // [36]
documentTest: 'document-node' '(' (elementTest | schemaElementTest)? ')'; // [56]
elementTest: 'element' '(' (eleAttrnameOrwildCard (',' qName '?'?)?)? ')'; // [64]
attributeTest: 'attribute' '(' (eleAttrnameOrwildCard (',' qName)?)? ')'; // [60]
schemaElementTest: 'schema-element' '(' qName ')'; // [66]
schemaAttributeTest: 'schema-attribute' '(' qName ')'; // [62]
```

```
piTest: 'processing-instruction' '(' (VarName | StringLiteral)? ')'; // [59]
wildCard: (VarName ':' '*') | ('*' ':' VarName); // Put '*'
eleAttrnameOrwildCard: qName | '*'; // [65]
valueComp: 'eq' | 'ne' | 'lt' | 'le' | 'gt' | 'ge'; // [23]]
nodeComp: 'is' | '<<' | '>>'; // [24]
commentTest: 'comment' '(' ')'; // [58]
textTest: 'text' '(' ')'; // [57]
anyKindTest: 'node' '(' ')'; // [55]
```

Pacote *URI*

```
parser grammar inSOAURI;
// URI: RFC3986
uriReference: Uri;
```