MARCOS AURÉLIO CARRERO

UM SISTEMA DE TEMPO DE EXECUÇÃO PARA A LINGUAGEM PEWS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Martin A. Musicante

CURITIBA

2006

MARCOS AURÉLIO CARRERO

UM SISTEMA DE TEMPO DE EXECUÇÃO PARA A LINGUAGEM PEWS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Martin A. Musicante

CURITIBA

2006

Dedico esta dissertação de mestrado aos meus pais, Dorival e Nélcia.

AGRADECIMENTOS

Aos meus pais, Dorival e Nélcia, pelo amor e constante apoio no aprimoramento da minha educação.

A Eduardo Sant'Ana, um grande amigo que havia começado os estudos no mestrado e que me motivou realmente a iniciar este novo desafio.

A Martin A. Musicante, meu orientador, pela motivação, pelos desafios e principalmente pela paciência por ele demonstrada nos momentos iniciais da preparação desta dissertação.

A Antonio Miguel Vieira Monteiro, chefe do Departamento de Processamento de Imagens do INPE e ao professor Paulo Justiniano Ribeiro Júnior, chefe do Laboratório de Estatística e GeoInformação da UFPR, pela oportunidade de participar de um projeto de pesquisa que foi de grande importância para minha formação científica.

Aos demais amigos do Laboratório de Estatística e GeoInformação da UFPR, professor Joel Rosa, Pedro Ribeiro de Andrade Neto, Thiago Mello, Elias Teixeira Krainski, pelos bons momentos e pela troca de informações durante minha passagem pelo laboratório.

Aos amigos do mestrado do Laboratório EsCeL, Marcelo Araújo, Rubens Barbosa Filho, Evandro Regolin, Leila Vriesmann e Alexandre Hausen, pelas incontáveis horas que passamos juntos, discutindo problemas da escrita da dissertação, problemas de implementação e também pelos momentos de descontração que ajudavam a reduzir o cansaço despendido pelo trabalho.

A Capes, pela bolsa concedida.

A minha “segunda família”, Viviani, Bonival e Sônia que muito me ajudaram nos momentos de dificuldade.

SUMÁRIO

LISTA DE FIGURAS v

RESUMO vi

ABSTRACT vii

1 INTRODUÇÃO 1

2 SERVIÇOS WEB 4

2.1 XML 4

2.2 Tecnologias para Serviços Web 5

2.2.1 SOAP 8

2.2.2 WSDL 11

2.2.3 UDDI 15

2.2.4 Provedor de Serviços Web 16

3 COMPOSIÇÃO DE SERVIÇOS WEB 20

3.1 Linguagens para Composição 20

3.1.1 BPEL 21

3.1.2 WSCI 22

3.1.3 PEWS 23

3.1.3.1 Semântica de Ações 24

3.1.3.2 Sintaxe de PEWS 26

3.1.3.3 Semântica de PEWS 28

3.1.3.4 XPEWS 32

3.1.3.5 Coreografia 34

3.1.3.6 Orquestração 37

4 PROGRAMAÇÃO CONCORRENTE **39**

4.1 Sincronização entre Processos 40

4.2 Serviços Web e Processos 41

4.2.1 CSP 41

4.2.2 Implementação Java da CSP 45

5 IMPLEMENTAÇÃO **49**

5.1 Arquitetura 50

5.2 Geração de Código 52

6 ESTUDO DE CASO **61**

6.1 Arquitetura para o Estudo de Caso 61

6.2 Descrição do cenário 63

7 CONCLUSÕES E TRABALHOS FUTUROS **70**

BIBLIOGRAFIA **74**

APÊNDICE A **82**

A.1 XPEWS Schema 82

APÊNDICE B **89**

A.2 Documentos WSDL referentes ao estudo de caso 89

APÊNDICE C **98**

A.3 Documentos XPEWS referentes ao estudo de caso 98

LISTA DE FIGURAS

2.1 Modelo de arquitetura de um serviço Web [64]. 6
2.2 Estrutura básica do protocolo SOAP [40]. 9
2.3 Exemplo de chamada de procedimento usando-se SOAP-RPC. 10
2.4 Exemplo de resposta a uma chamada de procedimento usando-se SOAP-RPC. 10
2.5 Descrição de um documento WSDL no nível abstrato 13
2.6 Descrição de um documento WSDL no nível concreto 14
2.7 Estrutura do protocolo UDDI [40]. 15
2.8 Gerenciamento de mensagens pelo servidor Axis [22] 18
2.9 Gerenciamento de mensagens pelo cliente [22] 19

3.1 Sintaxe da linguagem PEWS 27
3.2 Programa XPEWS referente à tradução do serviço loja. 33
3.3 Especificação da composição de serviços Web através da coreografia 34
3.4 Especificação da composição de serviços Web através da orquestração . . . 38

4.1 Combinação entre as operações entrada/saída [57] 42
4.2 Procedimento utilizado para a execução de operações 43
4.3 Especificação WSDL para o serviço P 44
4.4 Especificação WSDL para o serviço Q 44
4.5 Processo Produtor-Consumidor 46

5.1 Arquitetura do *back-end* da linguagem PEWS. 51
5.2 Algoritmo para avaliação de atributos sintetizados 53
5.3 Tradução dirigida pela sintaxe para avaliação das *path expressions* 54
5.4 Árvore sintática abstrata 55
5.5 Código gerado pelo *back-end* para uma operação. 56
5.6 Código gerado pelo *back-end* para um serviço. 60

6.1 Arquitetura do *back-end* para o Estudo de Caso 62

6.2 Composição de serviços Web utilizada no estudo de caso 63

6.3 Programa XPEWS referenta à tradução do serviço Banco. 66

6.4 Comunicação entre processos 68

6.5 Documento WSDL referente ao serviço Loja 68

7.1 *Workflow* não-estruturado 72

RESUMO

PEWS é uma linguagem para implementar interfaces de serviços Web. O objetivo de PEWS não é apenas ser usada na especificação de serviços Web simples ou compostos, mas também como um formalismo para a definição e raciocínio de propriedades dos serviços Web. O foco desta dissertação é no desenvolvimento de um sistema de tempo de execução para a linguagem PEWS. Nossa proposta enfatiza a idéia de que os serviços Web podem ser vistos como processos, onde a comunicação e a sincronização ocorre através de troca de mensagens, cujas operações executam em diferentes provedores de serviços. A ferramenta irá gerar um esqueleto de classes java a partir de programas XPEWS, uma versão XML de PEWS. O esqueleto de classes gerado utiliza a biblioteca JCSP para implementar a sincronização do serviço, o qual deverá ser estendido por programadores, a fim de gerar o sistema de tempo de execução da linguagem. O sistema é então registrado como um serviço em um servidor HTTP, o qual se comunica com clientes (e outros serviços) através do protocolo SOAP.

ABSTRACT

PEWS is a language for the implementation of web service interfaces. PEWS aims to be used not only in the specification of simple or composite web services, but also as a formalism over which we can reason on web Service properties. The focus of this thesis is the development of a run-time system for PEWS. Our proposal stresses that web services can be seen as processes, where the communication and synchronization is based on message-passing. These processes can be running in different service providers. Our tool generates a Java class skeleton from XPEWS programs (XPEWS is the XML version of PEWS). The generated class skeleton uses the JSCP library to implement the synchronization part of the service, which must be extended by the (programmer) user to generate the runtime system of the language. The runtime is registered as a service at the HTTP server, and communicates with clients (and other services) using SOAP.

CAPÍTULO 1

INTRODUÇÃO

Desde seu surgimento, a Internet está em constante evolução. Novas tecnologias estão sendo desenvolvidas para uso nesta área. Recentemente, a tecnologia dos serviços Web têm chamado a atenção das pessoas, tanto no meio acadêmico quanto na indústria. Informalmente, serviços Web podem ser vistos como uma coleção de programas, disponibilizados por diferentes empresas e que podem ser acessados pela Internet. Desta forma, uma desafiadora e importante tarefa é o estudo da composição de tais serviços a fim de se construir serviços complexos [34].

A descrição das interfaces dos serviços Web são geralmente especificadas pelo uso de linguagens com formato XML [69, 76]. As linguagens de interface, tal como a WSDL [29, 15], descrevem as interfaces de uma maneira estática (nomes, tipos e parâmetros das operações) e não descrevem o *comportamento* (ou *workflow*) dos serviços Web, isto é, elas não especificam a ordem de execução das operações. A descrição das interfaces em WSDL não especificam *como* a interação entre os serviços podem ser construídas, elas não possibilitam a verificação de sistemas dinâmicos e não cumprem os recentes requisitos do *W3C Choreography working group* [8].

A fim de solucionar este problema, várias abordagens têm sido propostas, desde a definição de novas linguagens até o uso de modelos formais. No nível de implementação, existem linguagens criadas para especificar o comportamento (ou *workflow*) dos serviços Web. Dentre estas linguagens, pode-se citar a WSCI [7], XLANG [62], WSFL [41], BPML [35], BPEL4WS [6], LCWS [17], WSTL [53] e PEWS [10].

Pelo lado teórico, pode-se verificar o uso de modelos de concorrência para especificar a composição dos serviços Web. Estes modelos incluem Álgebra de Processos [58], Máquina de Estados Finitos [11], Redes de Petri [34] e *Path Expressions* [4].

As linguagens e formalismos acima mencionados trabalham com a definição de com-

posição de serviços Web. Um serviço Web composto é um novo serviço construído pela combinação de serviços existentes. A composição de novos serviços Web a partir de serviços existentes é uma tarefa desafiadora, sendo necessário garantir a correta interação entre os serviços que são desenvolvidos por diferentes *softwares* e plataformas.

No trabalho introduzido por Ba et al. [10] uma nova linguagem foi criada para descrever a composição dos serviços Web. PEWS é uma linguagem simples, mas possibilita descrever de maneira expressiva a ordem e restrições das operações disponibilizadas pelos serviços Web. Uma versão XML de PEWS, chamada XPEWS, foi criada para que a linguagem possa ser utilizada para publicação e busca. Porém, a implementação automática do serviço encontra-se em aberto, sendo este problema objeto de estudo desta dissertação.

O desenvolvimento da linguagem PEWS foi dividida paralelamente como tema de duas dissertações de mestrado. O sistema será composto por um *front-end* e por um *back-end*. A dissertação [54] irá abordar o *front-end* da linguagem, o qual está sendo desenvolvido como um plug-in da plataforma eclipse [19]. O *front-end* é responsável pela edição de programas em PEWS e por gerar os programas equivalentes em XPEWS.

O *back-end*, estudo desta dissertação, tem por objetivo a implementação de um sistema de tempo de execução para a linguagem PEWS: dada uma especificação em XPEWS, que é uma representação XML de um programa codificado em PEWS, será construído um esqueleto de classes em Java que implementarão a composição do serviço. A implementação irá seguir as especificações dadas no artigo [9]. A linguagem de programação Java foi escolhida para implementar o *back-end*, por ser multiplataforma, pela implementação de bibliotecas úteis ao *back-end*, as quais implementam as tecnologias mais importantes usadas pelos serviços Web e por sua rica documentação.

Esta dissertação é composta por 6 capítulos. O Capítulo 2 apresenta conceitos básicos sobre a tecnologia dos serviços Web. Dá-se uma visão geral dos padrões atualmente discutidos na literatura. O Capítulo 3 discute as principais linguagens utilizadas na composição dos serviços Web. É feita a descrição da sintaxe e da semântica da linguagem PEWS e finaliza mostrando as duas abordagens principais usadas para compor serviços Web. O Capítulo 4 discute as primitivas da programação concorrente que serão utilizadas

na implementação do *back-end* da linguagem PEWS. O Capítulo 5 discute o modelo da arquitetura utilizada pelo *back-end*, bem como detalhes da geração de código a partir dos programas XPEWS. O Capítulo 6 discute um cenário para especificar a composição de serviços Web usando-se a linguagem PEWS. O Capítulo 7 apresenta conclusões, problemas encontrados e sugestões para trabalhos futuros. Este trabalho contém 3 apêndices. O Apêndice A mostra a definição do XML Schema desenvolvido para a linguagem PEWS. O Apêndice B contém os documentos WSDL utilizados no estudo de caso e o Apêndice C contém os documentos XPEWS utilizados no estudo de caso.

CAPÍTULO 2

SERVIÇOS WEB

2.1 XML

Extensible Markup Language (XML) [69, 76] é uma linguagem para descrição de dados, sendo um subconjunto da *Standard Generalised Markup Language* (SGML) [36]. XML possui um formato que fornece informações estruturadas para a Web, possibilitando que os usuários possam definir suas próprias *tags* relativas ao contexto do documento. XML faz a separação do conteúdo e da formatação e descreve a estrutura do texto dentro de um documento, isto é, contém regras explícitas que determinam *onde* a estrutura de um documento começa e termina.

Um documento XML é composto de elementos (*elements*), definido por uma *tag* de abertura (`<tag>`) e por outra de fechamento (`</tag>`). As informações contidas entre as as *tags* são chamados de conteúdo (*content*) de um elemento. Os elementos podem conter outros elementos (sub-elementos) e atributos. Os atributos são definidos dentro da *tag* de início de um elemento e são compostos por um *nome* e por um *valor*. Um documento sintaticamente correto, isto é, cujos elementos estejam corretamente definidos de acordo com as regras mencionadas, é dito *bem formado*.

Os esquemas para XML possuem a função de descrever as restrições de estrutura e de semântica, sendo que qualquer documento que corresponda a este modelo (instância) deve satisfazer estas restrições [48]. Um documento XML é *válido* se ele é bem formado e se o uso dos elementos e atributos contidos no documento satisfazem um esquema especificado. Um exemplo típico de restrição estrutural para documentos XML está relacionada na especificação de um elemento (por exemplo, um elemento cujo nome é A pode conter apenas elementos cujo nome é B). Um exemplo de restrição semântica é a especificação de chaves (por exemplo, um atributo a que pertence a um elemento A, deve possuir um único valor entre todos os valores do atributo a de um documento XML). Existem

várias definições de esquema para XML, sendo que as mais populares são: DTD [69], XML Schema [63] e Relax NG [14]. Sendo que os serviços Web se utilizam do sistema de tipos provido pelo XML Schema na definição de tipos usados nas trocas de mensagens e na descrição de serviços [15], esta dissertação irá usar o XML Schema na definição das restrições dos documentos XML.

Zisman, em [76], enfatiza algumas características de XML que são de fundamental importância no contexto dos serviços Web. XML oferece um padrão para busca, manipulação e troca de dados na Web. De fato, XML é o coração dos serviços Web [51]. XML é usado na descrição de serviços, como um formato para troca de mensagens entre serviços e aplicativos, e para descrever a estrutura das mensagens. Devido à sua flexibilidade, o formato XML facilita o desenvolvimento, composição e manutenção dos serviços Web.

2.2 Tecnologias para Serviços Web

Os serviços Web são uma área promissora e emergente, que envolve importantes desafios tecnológicos [58]. Eles têm por objetivo prover reusabilidade e são tipicamente projetados para interagirem com outros serviços, a fim de que possam compor grandes aplicativos. A definição do termo ainda não é um consenso [65]. Vários autores definem serviços Web de maneiras diferentes. Dentre elas podem-se citar:

> "Serviços Web são componentes de software que utilizam tecnologias padrões da Internet para interagirem uns com os outros dinamicamente" [3].

> "Serviços Web são módulos de software que disponibilizam suas funcionalidades sobre a Internet através de interfaces bem definidas" [39].

> "Serviços web são módulos de aplicações independentes que podem ser descritos, publicados, localizados e executados sobre uma rede, geralmente, a Web" [61].

A fim de se disponibilizar uma definição padrão do termo Serviços Web e prover informações sobre os padrões utilizados, o W3C Web Services Architecture Working elaborou o documento Web Services Architecture, cuja definição para Serviços Web é:

> "Um Serviço Web é um programa projetado para prover a interoperabilidade entre máquinas, interagindo sobre uma rede. Ele possui uma interface descrita em um formato processável por máquina, em particular, WSDL. Outros sistemas interagem com o Serviço Web, de maneira pré-definida, usando-se mensagens SOAP, tipicamente conduzidas usando-se HTTP com serialização XML, em conjunto com outros padrões da Web" [47].

Os esforços em pesquisa sobre serviços Web são o de automatizar os processos de negócios, possibilitando acesso a recursos e software na Internet [20]. Exemplos de serviços Web são sites de vendas *on-line*[1] e agência de viagens[2]. Um modelo de arquitetura de serviços Web, como ilustra a Figura 2.1, é composto por três entidades: *service provider*, *service requester* e *service broker/registry* [18, 37, 12, 64]. Os participantes definem operações que descrevem a interação entre eles, as quais são: publicação (*publish*), busca (*find*) e ligação (*bind*).

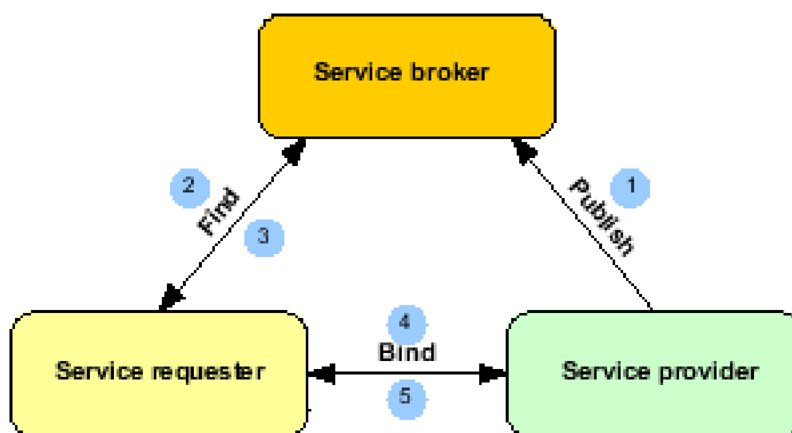


Figura 2.1: Modelo de arquitetura de um serviço Web [64].

No modelo ilustrado pela Figura 2.1, as operações ocorrem da seguinte maneira:

1. O *service provider* registra seus serviços em um registro que é mantido pelo *service broker*. O service broker representa um conjunto de interfaces de software para serviços Web publicados.

[1]Um exemplo de serviço Web de vendas é: http://www.amazon.com
[2]Um exemplo de serviço Web de viagens é: http://www.opentravel.org

2. O *service requester* faz uma solicitação ao registro do *service broker*, procurando pelo serviço desejado e por instruções de como utilizar o serviço.

3. Uma vez que o *service requester* encontrou o serviço correto, o *service broker* retorna os detalhes do serviço.

4. O *service requester* executa o serviço que se encontra no *service provider*

5. Finalmente o *service provider* retorna o resultado da execução do serviço ao *service requester*, finalizando a transação.

Para realizar tais operações, algumas tecnologias apoiadas na linguagem XML têm sido propostas para dar suporte ao desenvolvimento dos serviços Web. As tecnologias utilizadas no desenvolvimento dos serviços Web dividem-se em três áreas: protocolos de comunicação, descrição de serviços e reconhecimento de serviços, sendo que especificações estão sendo desenvolvidas para cada área. As seguintes especificações podem ser usadas para definir um serviço Web: [15, 65]

SOAP: é um protocolo com formato XML utilizado para a transmissão de mensagens entre os serviços Web;

***Web Services Description Language* (WSDL):** é uma linguagem XML que provê uma descrição da interface do serviço Web;

***Universal Description, Discovery, and Integration* (UDDI):** é um registro que contém a descrição dos serviços Web, sendo utilizado para busca de serviços disponíveis.

Dado o modelo de serviços Web ilustrado pela Figura 2.1, e as especificações que podem ser usadas para definir um serviço Web, pode-se então descrever a relação entre o modelo e padrões mencionados. Em um cenário típico, o *service provider* define um documento WSDL e o publica em um *service broker* através de um registro UDDI. O *service requester* utiliza a operação de busca para obter a descrição do serviço armazenado nos registros UDDI e faz uso da descrição do serviço para realizar a ligação com o *service provider*. A seguir é feita uma descrição sobre as funcionalidades do SOAP, WSDL e UDDI.

2.2.1 SOAP

O SOAP [67] é um protocolo de comunicação com um formato XML para troca de mensagens em um ambiente descentralizado e distribuído. Ao invés de definir um novo protocolo de transporte, o SOAP faz uso de protocolos existentes, sendo que o HTTP [28] é o mais utilizado [15]. É possível também utilizar outros protocolos tal como o *Simple Mail Transport Protocol* (SMTP) [55] e o FTP [27]. O XML juntamente com o HTTP fazem do SOAP um protocolo robusto, extensível, independente de plataforma e linguagem, sendo o suporte para alcançar a interoperabilidade entre diferentes plataformas[25].

Um dos objetivos do protocolo SOAP é facilitar a troca de mensagens, sendo que existem diferentes primitivas para realizar a transferência de mensagens. O protocolo pode ser utilizado simplesmente para enviar uma única mensagem ou para tarefas mais complexas, como troca de mensagens do tipo *request-response* [40] ou *Remote Procedure Call* (RPC) [26]. Utilizando SOAP é possível codificar as definições e chamadas a métodos e procedimentos usados em linguagens de programação [68]. Em particular, as trocas de mensagens utilizando-se SOAP são independentes da plataforma de software ou da linguagem de programação usada. Em um documento SOAP, o sistema de tipos e a estrutura do documento estão definidos por um esquema. A Figura 2.2 ilustra a estrutura básica do protocolo SOAP, o qual consiste de três partes: um *envelope* obrigatório, um cabeçalho (*header*) opcional e um corpo (*body*) obrigatório [40].

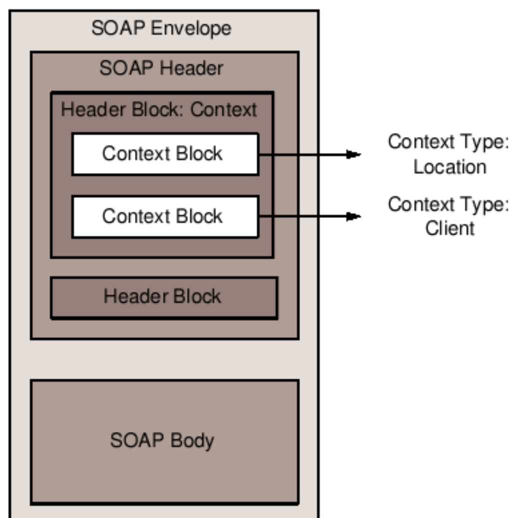


Figura 2.2: Estrutura básica do protocolo SOAP [40].

O *envelope* é o elemento raiz de uma mensagem e é composto por dois filhos, o elemento *header* e o elemento *body*. O elemento *header* pode conter meta-informações, tais como especificações para autenticação, roteamento, assinatura e transações. As informações que serão transmitidas pela mensagem são adicionadas ao elemento *body*.

O seguinte exemplo mostra o uso de SOAP para executar um serviço Web.

Exemplo 2.2.1 A Figura 2.3 ilustra o uso de SOAP para executar um serviço Web disponibilizado por algum provedor de serviços. Neste exemplo, sabe-se que um provedor disponibilizou um serviço para calcular a soma da série de *Fibonnaci*. O serviço receberá como entrada um número inteiro positivo, e devolverá como resultado a soma dos números de Fibonnaci de 1 até o número desejado. Sabe-se que o serviço possui um procedimento chamado *fib* que irá realizar esta operação e que o mesmo deverá ser executado através de RPC.

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>
  <ns1:fib
    soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:ns1="fib">
   <arg1 xsi:type="xsd:int">10</arg1>
  </ns1:fib>
  </soapenv:Body>
 </soapenv:Envelope>
```

Figura 2.3: Exemplo de chamada de procedimento usando-se SOAP-RPC.

A Figura 2.3 mostra como é possível codificar uma chamada de procedimento usando-se SOAP. O procedimento a ser executado, cujo nome é *fib*, está descrito na linha 7 pelo elemento `ns1:fib`. Esse elemento possui um único filho, o elemento `arg1`, definido pela linha 10, indicando que o procedimento possui apenas um parâmetro de entrada. O atributo do elemento `arg1`, denominado `xsi:type`, indica que o parâmetro é do tipo inteiro, o qual é definido em XML Schema. Neste exemplo, deseja-se calcular o valor da soma dos números de *Fibonnaci* para o argumento de valor 10. Enviando-se esse documento SOAP ao serviço corretamente, uma resposta a esta chamada é mostrada na Figura 2.4.

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>
  <ns1:fibResponse
    soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:ns1="fib">
   <ns1:fibReturn xsi:type="xsd:int">55</ns1:fibReturn>
  </ns1:fibResponse>
 </soapenv:Body>
</soapenv:Envelope>
```

Figura 2.4: Exemplo de resposta a uma chamada de procedimento usando-se SOAP-RPC.

A Figura 2.4 ilustra a mensagem de resposta enviada pelo serviço após a execução do

procedimento *fib*. Na linha 10, o conteúdo do elemento `ns1:fibReturn`, cujo valor é 55, indica o resultado da execução da soma dos números de *Fibonnaci* cujo valor de entrada é 10. O atributo `xsi:type` do elemento `ns1:fibReturn`, indica que o tipo do valor de retorno é um número inteiro, o qual é especificado em XML Schema. □

2.2.2 WSDL

Web Services Description Language (WSDL) [29, 15] é uma linguagem XML desenvolvida pela IBM e Microsoft para descrever as interfaces de um serviço Web. Um documento WSDL descreve a interface de um serviço Web como uma coleção de mensagens que pode ser usada para interagir com o serviço. Os objetivos da descrição de um serviço WSDL são: (*i*) prover uma descrição do serviço em um nível de aplicação, ou seja, indicar quais operações (ou métodos) o serviço disponibiliza e (*ii*) determinar os detalhes específicos do protocolo que são necessários para acessar o serviço [10]. De acordo com esses objetivos, é possível separar as definições abstratas dos detalhes de implementação de um serviço.

O nível abstrato, também chamado de descrição em nível de aplicação, provê uma descrição do serviço, por exemplo, quais operações ou métodos o serviço disponibiliza. Por outro lado, o nível concreto indica detalhes específicos do protocolo e de localização do serviço. Essa separação é importante visto que podem existir serviços semelhantes desenvolvidos em diferentes plataformas e executando em diferentes protocolos de comunicação. Dessa maneira possibilita-se o reuso das definições abstratas do serviço. Como uma forma de melhor expor as funcionalidades do WSDL, serão descritas primeiramente as definições abstratas do serviço e em seguida as suas definições concretas.

No nível abstrato, o elemento *message* é uma lista de *part*. As mensagens descrevem uma definição abstrata do tipos de dados que um serviço pode enviar e receber. Cada elemento *part* possui um *nome* e um *tipo* associado. WSDL usa um sistema de tipos externo que define os tipos dos dados utilizados na troca de informações, por exemplo XML Schema [63]. O elemento *operation* agrupa mensagens rotuladas como *input*, *output* ou *fault* e representa as interações suportadas pelo serviço. As operações podem definir quatro primitivas para troca de mensagens:

One-way Uma operação *one-way* possui apenas uma mensagem de entrada. Ela ocorre quando um serviço Web recebe uma mensagem de um cliente mas não retorna nenhuma mensagem como resposta.

Notification Uma operação *notification* é o oposto da operação *one-way*, ou seja, possui apenas uma mensagem de saída. Ela ocorre quando um serviço Web envia uma mensagem ao cliente mas não espera receber nenhuma mensagem como resposta.

Request-response Uma operação *request-response* possui uma mensagem de entrada e outra de saída, nesta ordem. Ela ocorre quando um cliente envia uma mensagem de requisição e o serviço Web responde a ele com outra mensagem.

Solicit-response Uma operação *solicit-response* é o oposto da *request-response*, ou seja, possui uma mensagem de saída e outra de entrada, nesta ordem. Ela ocorre quando um serviço Web envia uma mensagem ao cliente e espera receber outra mensagem como resposta.

Um *portType* é uma coleção de operações suportadas por um *end point* (pela localização das operações). Como observado em [10], um *portType* pode ser comparado com uma biblioteca de funções (um módulo, ou uma classe) das linguagens de programação tradicionais.

Para ilustrar as funcionalidades de um WSDL, adaptou-se um exemplo de [10] onde a interface de um serviço Web para uma *loja* será considerado. O serviço *loja* interage com outros serviços a fim de realizar as seguintes operações: receber um pedido, enviar uma fatura, receber um pagamento e enviar um recebimento.

Exemplo 2.2.2 A Figura 2.5 ilustra uma possível descrição em WSDL do serviço *loja* no nível abstrato.

```
<message name="pedidoEntradaProduto">
  <part name= "codProduto" type="xsd:string"/>
  <part name= "quantidade" type="xsd:integer"/>
</message>
<message name="pedidoSaidaProduto">
  <part name= "preço" type="xsd:real"/>
</message>
<message name="enviarRecebimentoCliente">
  <part name= "codProduto" type="xsd:string"/>
  <part name= "dataPagamento"  type="xsd:date"/>
</message>
<message name="enviarFatura">
  <part name= "codProduto" type="xsd:string"/>
  <part name= "total" type="xsd:real"/>
</message>
<message name="ackFatura">
  <part name= "codProduto" type="xsd:string"/>
</message>
...
<portType name="Loja">
  <operation name="pedido">
   <input message="pedidoEntradaProduto"/>
   <output message="pedidoSaidaProduto"/>
  </operation>
  <operation name="fatura">
   <output message="enviarFatura"/>
   <input message="ackFatura"/>
  </operation>
  <operation name="pagamento">
   <input message="obterPagamento"/>
  </operation>
  <operation name="recebimento">
   <output message="enviarRecebimentoCliente"/>
  </operation>
</portType>
```

Figura 2.5: Descrição de um documento WSDL no nível abstrato

Na Figura 2.5, o fragmento mostra definições de tipo de dados (com tipos inteiro, string, etc definidos em XML Schema) e um elemento *portType* que agrupa quatro diferentes operações, que são pedido, fatura, pagamento e recebimento. Nesse exemplo, pedido é uma operação do tipo *request-response*, visto que ela primeiro recebe como entrada a mensagem pedidoEntradaProduto e depois retorna a mensagem pedidoSaidaProduto como resposta. Por outro lado, fatura é uma operação do tipo *solicit-response*, enviando primeiramente a mensagem enviarFatura como solicitação e recebendo depois a mensagem

ackFatura como uma resposta. A operação pagamento é uma operação *one-way* que possui a mensagem obterPagamento como entrada, enquanto que recebimento é uma operação *notification* que apenas envia uma mensagem de saída. □

Até o momento, as funcionalidades discutidas foram utilizadas para definir serviços Web em uma camada abstrata. Para completar essa descrição, é preciso definir as funcionalidades concretas que ele implementa, sendo que o nível concreto consiste de três partes:

- *qual* protocolo de comunicação será usado (por exemplo SOAP sobre HTTP);
- *como* realizar a interação de um dado serviço sobre esse protocolo;
- *onde* terminar a comunicação (o endereço da rede).

O seguinte exemplo mostra uma possível descrição do serviço *loja* no nível concreto.

Exemplo 2.2.3 Na Figura 2.6, o elemento *bind* especifica as interações sobre um protocolo específico. Pode-se verificar, nas linhas 3 e 4, que os atributos *style* e *protocol* definem qual a maneira de executar a operação e qual o protocolo utilizado, respectivamente. Em particular, o protocolo utilizado pelo serviço é o SOAP sobre HTTP, e a execução da operação pedido será através de RPC.

```
1  <wsdl:binding name="LojaSoapBinding" type="impl:Loja">
2   <wsdlsoap:binding
3     style="rpc"
4     transport="http://schemas.xmlsoap.org/soap/http"/>
5    <wsdl:operation name="pedido">
6     <wsdl:input name="requisiçãoPedido"/>
7     <wsdl:output name="respostaPedido"/>
8    </wsdl:operation>
9  </wsdl:binding>
10 <wsdl:service name="ServiçoLoja">
11  <wsdl:port binding="impl:LojaSoapBinding" name="Loja">
12   <wsdlsoap:address location="http://localhost:8080/axis/Loja.jws"/>
13    </wsdl:port>
14 </wsdl:service>
```

Figura 2.6: Descrição de um documento WSDL no nível concreto

Finalmente, resta agora descrever *onde* o serviço está implementado. Na linha 10, o elemento *service* define o *end point* como uma combinação de um *binding* e um endereço da rede. □

2.2.3 UDDI

O *Universal Description, Discovery, and Integration* (UDDI) [49] é um protocolo que define um padrão para publicação e descoberta de serviços. O UDDI descreve um registro de serviços Web e interfaces de programação para publicar, recuperar e gerenciar informações sobre esses serviços. Desta forma um registro pode oferecer mecanismos para classificar, catalogar e gerenciar serviços Web, fazendo com que eles possam ser descobertos e executados por outros aplicativos.

A especificação provida pelo UDDI define serviços que apóiam a descrição e descoberta de: (a) negócios, empresas e outros serviços Web, (b) dos serviços Web disponibilizados por ele, e (c) das especificações das interfaces que serão usadas para o acesso e gerenciamento dos serviços. XML Schema é utilizado para descrever tipos e estruturas de dados nesse contexto. A Figura 2.7 mostra a organização de um registro UDDI e seus principais elementos: *businessService, businessEntity, bindingTemplate* e *tModel*.

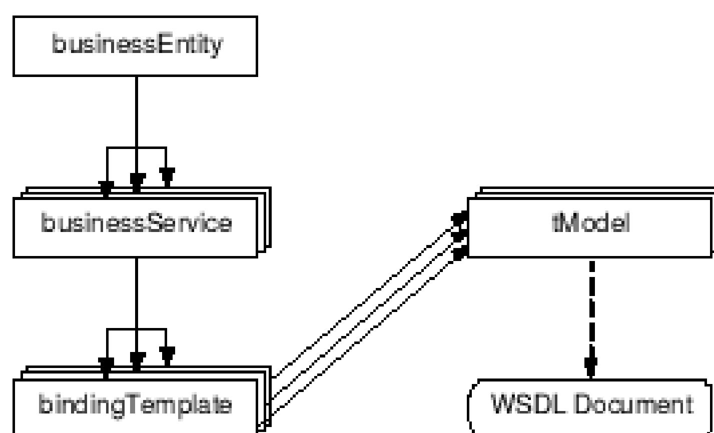


Figura 2.7: Estrutura do protocolo UDDI [40].

Um *businessEntity* possui informações sobre uma empresa ou organização que publicou o serviço. Esses dados incluem o nome da empresa, descrição, serviços oferecidos,

informações para contato, etc [50]. Essas informações são chamadas de *white pages* do UDDI [15]. Cada *businessEntity* pode associar uma lista de *businessServices*. Um *businessService* descreve informações sobre os serviços oferecidos, sendo chamada de *yellow pages* do UDDI. Um *businessService* descrever os detalhes técnicos de um serviço, sendo chamado de *green pages* do UDDI. Um *businessService* pode ter um ou mais *bindingTemplates*. O componente mais importante do *bindingTemplate* é o ponto de acesso, ou seja, informar a localização do serviço para que o mesmo possa ser executado [40]. Um *bindingTemplate* pode fazer referência a vários *tModels*. Um *tModel* possui diversos atributos, tais como a classificação de um serviço, o tipo de transporte, assinatura digital entre outros. Ele também pode conter um link para um documento WSDL que possui informações que descrevem as interfaces dos serviços.

2.2.4 Provedor de Serviços Web

A *Apache Software Foundation* [21] oferece suporte para a comunidade *Apache* em projetos que envolvem software de código aberto. Uma gama de projetos de alta qualidade têm sido desenvolvidos pela comunidade, no qual o *Apache Axis* [23, 22] está inserido em um dos projetos da área de serviços Web. O Axis é uma implementação do protocolo SOAP, podendo ser usado tanto como um cliente como um provedor de serviços Web. Atualmente existem versões para as linguagens Java e C++, sendo que o enfoque desta dissertação será na versão Java deste software. Desta forma, o Axis foi escolhido como provedor de serviços Web, por ser um projeto de código aberto, compatível com a linguagem Java e pelo seu grande número de usuários, tanto no meio acadêmico como corporativo.

O Axis é executado sobre o servidor de aplicações Web Apache Tomcat. Ele possibilita que métodos implementados por uma classe Java fiquem acessíveis e possam ser executados através do SOAP, sem a necessidade de se trabalhar diretamente com a criação do documento SOAP. O Axis gera um documento WSDL automaticamente e faz o tratamento de codificação e decodificação das mensagens SOAP. Se o Axis for usado como um servidor, existem duas maneiras para disponibilizar um serviço Web [38]:

- Renomear o arquivo xxx.java para xxx.jws e colocá-lo dentro do diretório do servi-

dor. Todos os métodos ficarão acessíveis através do SOAP e o documento WSDL correspondente será gerado automaticamente. Esta maneira de disponibilizar serviços Web é muito simples, porém não é muito configurável, não aceitando por exemplo pacotes Java.

- O Axis possui um arquivo no formato *Web Service Deployment Descriptor* (WSDD) que pode ser usado para descrever quais métodos estarão habilitados para serem disponibilizados como serviços Web e para especificar os componentes *handlers* e *chains*, os quais serão descritos no decorrer desta seção.

Como um serviço cliente, há duas maneiras para executar um serviço Web:

- Usando a ferramenta `WSDL2Java` para gerar o código Java que irá habilitar a um programador as chamadas aos serviços Web, como se os serviços fossem métodos localizados na própria máquina
- Fazendo uso das classes da API do Axis

O sistema de execução da linguagem PEWS irá usar a segunda opção, ou seja, utilizar diretamente a API do Axis para construir o serviço cliente. A primeira opção não foi escolhida porque a ferramenta `WSDL2Java` gera código incompleto para criar as chamadas a métodos(operações) de um cliente, necessitando que o programador tenha que editar o código gerado pela ferramenta para completá-lo.

A Figura 2.8 mostra como o servidor Axis gerencia o processamento das mensagens. Os cilindros menores representam os *handlers* e os cilindros maiores, que os englobam, representam os *chains*.

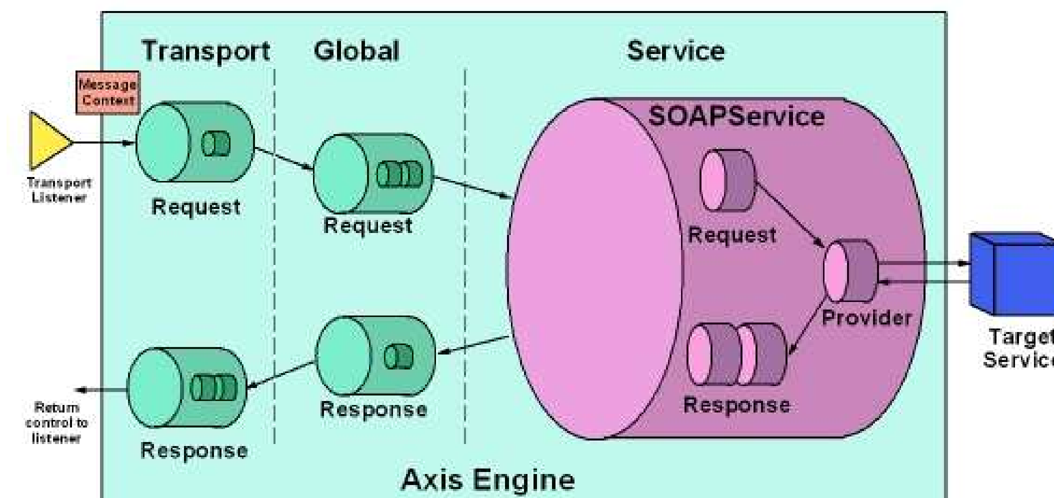


Figura 2.8: Gerenciamento de mensagens pelo servidor Axis [22]

Quando o servidor Axis recebe uma mensagem, o contexto *Message Context* é criado e a mensagem é instalada nele. Então a mensagem irá seguir por um caminho que consiste por um fluxo de requisições(*requests*), pelo seu processamento e possivelmente por um fluxo de respostas(*requests*). Estes fluxos contém *handlers*, os quais são módulos responsáveis pelo tratamento das mensagens. Um *chain* contém uma seqüência de *handlers*, o qual é responsável por executá-los em uma ordem pré-definida.

Se o Axis é usado como um servidor, os *chains* do *transport request* e do *global request* são executados quando uma requisição for recebida. Um destes *chains* contém um *handler* que modifica a propriedade do serviço, inserida no contexto *Message Context*. Isto possibilita com que o Axis selecione o serviço correto para a requisição. Um serviço consiste de um *chain* de requisição e resposta. O provedor(*provider*) é um *handler* responsável pela execução do serviço. A mensagem de resposta segue o caminho da resposta de mensagem do provedor, passando pelo serviço e pelos *chains global* e *transport*.

Pelo lado do cliente, a Figura 2.9 mostra o gerenciamento do processamento das mensagens.

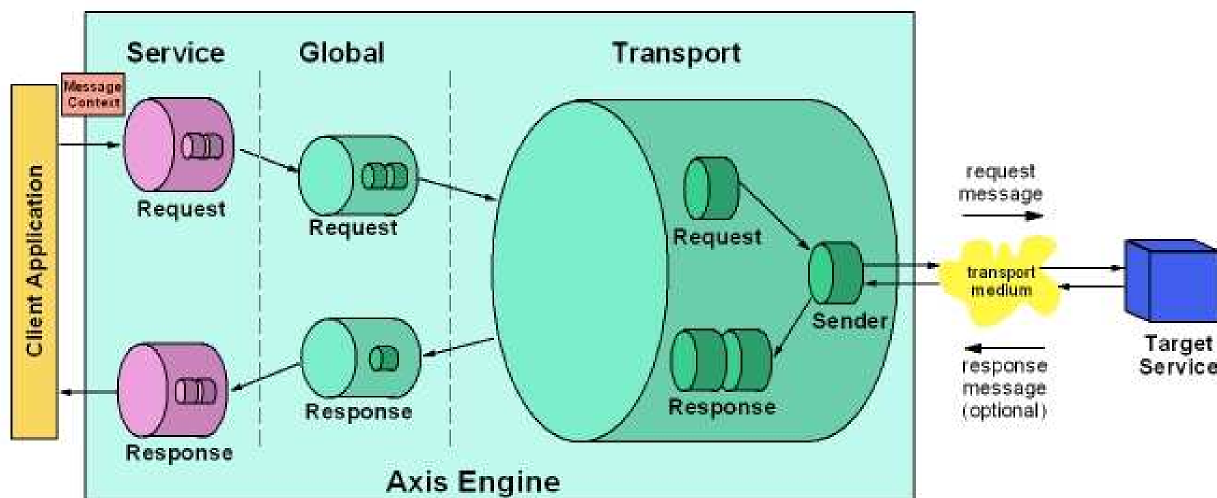


Figura 2.9: Gerenciamento de mensagens pelo cliente [22]

O caminho das mensagens quando o Axis está atuando como um cliente é quase o mesmo, porém a ordem dos *chains* é inversa: primeiro pelo *chain* do serviço, em seguida pelos *chains global* e *transport*. A mensagem de resposta segue o mesmo caminho de maneira inversa.

CAPÍTULO 3

COMPOSIÇÃO DE SERVIÇOS WEB

Com a evolução dos padrões para a criação de serviços Web, diversos serviços são disponibilizados pelos provedores de serviços. Por conseguinte, a habilidade de compor e criar novos serviços com a finalidade de auxiliar na solução de problemas complexos torna-se importante. A composição de serviços Web pode ser definida como o processo de criação de um novo serviço através da combinação de serviços existentes [1].

Para desenvolver novos serviços através da composição, dois enfoques podem ser usados: a coreografia (*choreography*) e a orquestração (*orchestration*) [58]. A orquestração é o problema de se resolver tarefas complexas utilizando-se de serviços simples, através do desenvolvimento de um *novo serviço*, o qual é chamado de *orquestrador*. O orquestrador é, neste caso, o gerenciador dos serviços existentes, coordenando a troca de mensagens entre eles. Por outro lado a coreografia não se utiliza de um serviço central para gerenciar a troca de mensagens, sendo que o problema de se resolver tarefas complexas é solucionado pela interação entre os serviços. Um dos desafios da coreografia é realizar a composição dinâmica de serviços, pois há ocasiões onde não é possível saber *a priori* quais os objetivos, participantes e quais interações existirão na composição [58].

3.1 Linguagens para Composição

Quando um serviço é publicado na Web, os detalhes de sua implementação tornam-se ocultos aos usuários. Se a descrição da interface for reduzida uma lista de declarações de mensagens e tipos, tal como em um documento WSDL, então não existe nenhuma informação a respeito das seqüências de mensagens que o serviço pode enviar e receber. Os documentos WSDL não especificam o comportamento da interface do serviço, isto é, a ordem na qual as operações do serviço podem ser executadas não estão definidas pelo padrão WSDL. Para isso é necessário utilizar outras linguagens que descrevam o com-

portamento dos serviços Web [24]. Exemplos dessas linguagens são BPEL [6], WSCI [7], LCWS [17], WSTL [53] e PEWS [10]. Um modelo geral, proposto em [58], classifica as linguagens de serviços em três camadas:

1. Nível abstrato. São utilizados métodos formais para especificação de serviços, tais como Álgebra de processos, Máquinas de estados finitos, Redes de Petri, *Path Expressions* entre outras.

2. Nível público ou de interface. São utilizadas linguagens com formato XML que descrevem o comportamento do serviço.

3. Nível privado ou de implementação. São utilizadas linguagens de programação para implementar os serviços Web.

A seguir é feita uma descrição das linguagens utilizadas na composição de serviços, tendo como base os conceitos aqui introduzidos.

3.1.1 BPEL

A Linguagem de Execução de Processos de Negócios para Serviços Web (Business Process Execution Language for Web Services (BPEL4WS) [6], ou simplesmente BPEL) foi desenvolvida através da combinação das linguagens XLANG [62] e WSFL [41], criadas respectivamente pela Microsoft e IBM [74]. Desta maneira, BPEL compartilha características de uma linguagem estruturada em blocos, herdada de XLANG, com grafos direcionados originário da WSFL. BPEL pode ser usada tanto no nível de interface quanto no nível de implementação.

BPEL é uma linguagem com um formato XML que define a composição de Serviços Web através da descrição de processos de negócios [43, 16]. Um processo pode ser definido de duas maneiras: processo abstrato ou executável [42]. Um processo executável provê uma definição completa sobre o modelo do processo tal como ele será interpretado. Por outro lado, um processo abstrato descreve apenas o comportamento do serviço e pode ocultar informações que não são relevantes para os outros participantes da interação.

Em BPEL, o resultado da composição é chamado de processo (*process*), os serviços que interagem em um processo são chamados de participantes (*partners*) e o modo como ocorrem as trocas de mensagens entre os participantes chama-se atividade (*activity*). Um processo é composto por um conjunto de atividades e pode interagir com serviços externos através de uma interface WSDL.

Existem dois tipos de atividades: básica e estruturada [75]. As atividades básicas incluem primitivas para troca de mensagens, como `<invoke>`, `<receive>` e `<reply>`, os quais correspondem às operações definidas pelo *portType* de um WSDL. Outras operações como manipular dados `<assign>`, especificar a pausa em um processo `<wait>`, encerrar um processo `<terminate>`, não executar nenhuma atividade `<empty>` e realizar tratamento de exceções ou falhas `<throw>` completam as atividades básicas. As atividades estruturadas compreendem: controle de seqüência `<sequence>`, execução paralela `<flow>`, condicional `<switch>`, resposta a um evento externo `<pick>` e iteração `<while>`.

3.1.2 WSCI

Web Services Choreography Interface (WSCI) [7] é uma linguagem com um formato XML utilizada para descrever a coreografia, ou seja, o fluxo das trocas de mensagens entre os serviços Web [13]. Diferentemente de BPEL, a WSCI define apenas o *comportamento observável* dos serviços Web e não o processo executável, sendo portanto classificada como uma linguagem de nível de interface [52]. WSCI pode ser utilizada em conjunto com qualquer linguagem para descrição de serviços Web, em particular a WSDL. Ambas podem ser vistas como linguagens de interface. A diferença é que a WSDL descreve uma interface estática das operações de um serviço, enquanto que a WSCI descreve uma interface dinâmica, possibilitando o relacionamento entre várias operações através de troca de mensagens.

Em um documento WSCI, o construtor `interface` descreve o comportamento observável de um serviço Web. Uma interface é composta por várias ações. Cada ação mapeia uma operação definida em um `portType` de um documento WSDL. As ações podem ser atômicas ou complexas. Elas constituem a unidade básica do comportamento de

um serviço Web. Ações atômicas incluem operações para enviar e receber mensagens, ou esperar por um tempo determinado. Ações complexas incluem primitivas como execução paralela, execução sequencial, condicionais e iterações. Um processo *process* é um conjunto de ações. O componente *exceptions* pode ser associado à um conjunto de atividades para tratamento de exceções.

3.1.3 PEWS

Predicate path-Expression for Web Services (PEWS) [10] é uma linguagem que permite especificar a ordem em que as operações de um dado serviço Web são executadas. Desta forma, PEWS pode ser usada na especificação de sistemas e também como um guia na implementação de cada operação envolvida nesse sistema. O objetivo da linguagem PEWS é descrever as interfaces dos serviços Web, através do uso das *predicate path expressions* para definir o comportamento dos serviços Web. PEWS foi projetada não apenas para ser utilizada na especificação de um serviço Web simples ou composto, mas também como uma forma sintática na qual seja possível raciocinar sobre as propriedades dos serviços Web.

Predicate path expressions [4], uma extensão das *path expressions*, foram introduzidas como uma ferramenta para expressar a sincronização de operações em objetos. *Path expressions* são uma ferramenta para a sincronização de processos concorrentes, utilizadas para restringir as seqüências permitidas das operações em um objeto. Por exemplo, dadas as operações a, b e c, a *path expression* $a^*.(b \mid\mid c)$ define que a execução paralela das operações b e c deve ser precedida por zero ou mais execuções de a.

Predicate path expressions estendem as *path expressions* através da adição de predicados. As *predicate path expressions* preservam as vantagens das *path expressions* mas são mais poderosas, visto que podem conter variáveis e predicados. Por exemplo, a *predicate path expression* $a^*.([P]b + [\mathsf{not}\ P]c)$ define que b ou c deverá ser executada de acordo com o valor verdade do predicado P.

Em PEWS, uma *path expression* combina os nomes das operações do serviço Web usando os operadores de `seqüência` (.), `escolha exclusiva` (|), `paralelismo` (||), `repetição`

(*), `repetição paralela` ({...}) e `prefixo de predicado` (([...] ...), os quais serão descritos nas próximas seções.

3.1.3.1 Semântica de Ações

As descrições da sintaxe e da semântica da linguagem PEWS foram expressas através do uso da *Semântica de Ações*. Semântica de Ações [44, 70] é uma ferramenta formal para especificação de semântica, desenvolvida para prover descrições "legíveis" de linguagens da vida real. As descrições em Semântica de Ações são composicionais, isto é, elas definem funções semânticas para mapear objetos sintáticos abstratos para entidades semânticas. Funções semânticas são definidas indutivamente usando equações. Entidades semânticas são ações, que possibilitam uma maneira natural para expressar conceitos presentes na computação.

Semântica de Ações usa uma notação especial para descrever ações. Esta notação é denominada Notação de Ações (*Action Notation*), e é usada nas descrições da semântica de ações exatamente como a notação λ é usada na semântica denotacional [60, 70]. Os símbolos usados na Notação de Ações são propositadamente expressos verbalmente, por isso frases podem ser usadas, formalmente completas, para expressar a maioria dos conceitos presentes na computação.

As funcionalidades que a Semântica de Ações oferece são similares a outros formalismos para semântica. Ela é semelhante à Semântica Denotacional, a qual usa funções semânticas para descrever o significado dos objetos. No entanto as ações possuem mais características operacionais do que as funções. Neste sentido, as Ações Semânticas atenuam as diferenças entre as Semânticas Denotacional e Operacional [73].

Ações são usadas para descrever o significado da computação. Ações podem ser executadas para processar informação, cuja execução pode produzir diferentes resultados:

complete: indica término normal - a execução procedeu normalmente;

escape: indica término anormal - resultado inesperado;

fail: a execução resultou em falha;

diverge: não finalizou;

A Notação de Ações disponibiliza algumas ações primitivas e diferentes combinações para compor ações complexas, que correspondem aos principais conceitos das linguagens de programação. Estas podem ser: *seqüenciadas*, ou seja, uma sub-ação é executada antes de outra; *intercaladas*, as suas sub-ações atômicas são executadas em ordem aleatória; *escolhidas exclusivamente*, onde apenas uma das sub-ações é escolhida para execução.

Uma Notação de Dados é usada para descrever o processamento da informação pelas ações. A Notação de Dados padrão provê uma coleção de tipos abstratos de dados, incluindo números, caracteres, strings, tuplas, mapeamento, etc.

Juntamente com as ações e os dados, existe também uma terceira classe especial de entidade, denominada *yielders*. Uma *yielder* é uma expressão que representa uma informação não-processada, cujo valor depende da informação atual disponível para a ação primitiva na qual ela ocorre. *Yielders* são avaliadas para produzir informação. *Bindings* são mapeamentos de identificadores para dados. Um exemplo de uma *yielder* padrão é o mapeamento de dados para I, o qual depende do *binding* atual que são recebidos pela ação primitiva correspondente. Um exemplo de mapeamento é mostrado a seguir, na descrição da faceta declarativa.

Ações podem representar apenas controle ou podem processar diferentes tipos de informações. O comportamento das ações são representadas pelas facetas. Cada faceta é responsável por gerenciar uma parte da informação processada pela ação. Para cada ação, existem cinco facetas:

faceta básica: trata apenas do fluxo de controle;

faceta funcional: corresponde às ações que processam dados transitórios (recebem ou fornecem dados). Por exemplo, dada a ação primitiva "sucessor de um número natural" e, dado um número natural n como dado transitório, a ação é executada produzindo como resultado $n+1$. O combinador funcional *then* estabelece a seqüenciação e transferência de dados transitórios entre as ações. A ação composta *A1 then A2* define que *A1* deve ser executada primeiro, recebendo os valores transitórios de

entrada; todos os valores transitórios são então repassados para a execução de *A2*. A ação primitiva *choose* D, onde *D* é um conjunto de dados, realiza a escolha não-determinística de um elemento do conjunto *D*;

faceta declarativa: engloba as ações que definem *bindings*. Por exemplo, a execução da ação primitiva "bind max-length to 256", faz o mapeamento do token *maxlength* com o número natural 256;

faceta imperativa: esta faceta interage com a memória (*storage*). Uma memória é uma Notação de Ação que mapeia uma célula de memória com um valor armazenado. Por exemplo, considere a ação "allocate a cell then store 256 in the given cell", a qual tem características das facetas funcional e imperativa;

faceta comunicativa: esta faceta provê um sistema de agentes, os quais podem ser contratados para executar uma ação específica. Inicialmente, apenas um agente especial está ativo. Agentes podem comunicar-se usando comunicação assíncrona por troca de mensagens. Cada agente possui seu próprio canal de comunicação, no qual todas as mensagens enviadas para o agente são armazenadas.

3.1.3.2 Sintaxe de PEWS

A gramática ilustrada pela Figura 3.1 descreve uma notação padrão de dados definida por [44]. Símbolos não-terminais (lado esquerdo das definições) são utilizados como termos de cada equação. Os colchetes duplos definem as árvores sintáticas e os símbolos terminais são identificados pelas aspas duplas.

Nessa gramática, a notação □ significa que o lado direito da regra não é definido aqui (todos os casos definidos acima são especificados diretamente). Uma interface é definida como uma seqüência de definições def seguida de uma *path expression*. Cada definição declara variáveis inteiras para serem usadas em predicados. O valor das variáveis é obtido pela avaliação de uma expressão aritmética que envolve contadores pré-definidos e biblioteca de funções. Os contadores definem a ordem em que as operações serão executadas durante a avaliação de um predicado. Cada contador é composto por um par de inteiros

$$
\begin{array}{lll}
(1) & \textsf{interface} & = [\![\ \textsf{portType def* path}\ ]\!] \\
(2) & \textsf{path} & = [\![\ \textsf{opname}\ ]\!] \mid [\![\ \textsf{path}\ \text{``.''}\ \textsf{path}\ ]\!] \mid [\![\ \textsf{path}\ \text{``|''}\ \textsf{path}\ ]\!] \mid \\
 & & \quad [\![\ \textsf{path}\ \text{``||''}\ \textsf{path}\ ]\!] \mid [\![\ \textsf{path}\ \text{``*''}\ ]\!] \mid [\![\ \text{``\{''}\ \textsf{path}\ \text{``\}''}\ ]\!] \mid \\
 & & \quad [\![\ \text{``[''}\ \textsf{pred}\ \text{``]''}\ \textsf{path}\ ]\!] \\
(3) & \textsf{pred} & = [\![\ \text{``true''}\ ]\!] \mid [\![\ \text{``false''}\ ]\!] \mid [\![\ \text{``not''}\ \textsf{pred}\ ]\!] \mid \\
 & & \quad [\![\ \textsf{pred boolOp pred}\ ]\!] \mid [\![\ \textsf{arith-expr relOp arith-expr}\ ]\!] \\
(4) & \textsf{def} & = [\![\ \text{``def''}\ \textsf{var}\ \text{``=''}\ \textsf{arith-expr}\ ]\!] \\
(5) & \textsf{portType} & = [\![\ \textsf{operation+}\ ]\!] \\
(6) & \textsf{operation} & = [\![\ \textsf{opname}\ \text{``(''}\textsf{opArg}\ \text{``)''}]\!] \\
(7) & \textsf{opArg} & = [\![\text{``in:''}\ \textsf{msgName}]\!] \mid [\![\text{``out:''}\ \textsf{msgName}]\!] \mid \\
 & & \quad [\![\text{``in-out:''}\ \textsf{msgName}\ \text{``, ''}\ \textsf{msgName}\ ]\!] \mid \\
 & & \quad [\![\text{``out-in:''}\ \textsf{msgName}\ \text{``, ''}\ \textsf{msgName}\ ]\!] \\
(8) & \textsf{msgName} & = \Box \\
(9) & \textsf{opname} & = \Box \\
(10) & \textsf{arith-expr} & = [\![\ \textsf{var}\ ]\!] \mid [\![\ \textsf{arith-expr arithOp arith-expr}\ ]\!] \mid [\![\ \text{``now()''}\ ]\!] \mid \\
 & & \quad [\![\ \text{``act(''}\ \textsf{opname}\ \text{``)}.val\text{''}\ ]\!] \mid [\![\ \text{``act(''}\ \textsf{opname}\ \text{``)}.time\text{''}\ ]\!] \mid \\
 & & \quad [\![\ \text{``term(''}\ \textsf{opname}\ \text{``)}.val\text{''}\ ]\!] \mid [\![\ \text{``term(''}\ \textsf{opname}\ \text{``)}.time\text{''}\ ]\!] \\
(11) & \textsf{boolOp} & = \Box \\
(12) & \textsf{relOp} & = \Box \\
(13) & \textsf{arithOp} & = \Box \\
(14) & \textsf{var} & = \Box
\end{array}
$$

Figura 3.1: Sintaxe da linguagem PEWS

$(val, time)$. O componente val representa o próprio contador enquanto que o componente $time$ indica o momento em que o contador foi modificado pela última vez. Para cada operação O de um serviço Web, supõe-se a existência de três contadores:

req(O): O componente val descreve o número de vezes que um solicitador requisitou a execução a operação O. O componente $time$ indica o momento da última requisição do serviço.

act(O): O componente val descreve o número de vezes que um solicitador começou a

executar a operação O. O componente *time* indica o momento da última ativação do serviço.

term(O): O componente *val* descreve o número de vezes que um solicitador finalizou a execução da operação O. O componente *time* indica o momento da última conclusão do serviço.

Para ilustrar o uso de PEWS como um formalismo que possa ser utilizado na especificação de sistemas e para descrever o comportamento dos serviços Web, considere o seguinte exemplo retirado de [10]:

Exemplo 3.1.1 Em uma loja, suponha que o pagamento deverá ser feito dentro de 48 horas após o envio da fatura. Se este prazo não for respeitado, todo o processo é abortado. Um possível programa PEWS que expressa esta situação poderia ser escrito da seguinte forma:

def t^{pgto} = now() - term(enviarConta).$time$

(fazerPedido.enviarConta.([$t^{pgto} \leq 48h$] enviarPagamento.emitirRecibo | [$t^{pgto} > 48h$] abortarOperação))*

No exemplo acima, o valor da variável t^{pgto} é computada pela avaliação da expressão now() - term(enviarConta).$time$, a qual envolve a função now(), que calcula a hora atual. Os predicados que aparecem na expressão representam guardas, as quais provêem a semântica de uma construção condicional. As guardas são avaliadas até que uma delas seja verdadeira. Note que o valor da variável t^{pgto} é alterado a cada avaliação, desde que seu valor depende da hora atual. □

3.1.3.3 Semântica de PEWS

Esta seção apresenta, usando-se a ferramenta Semântica de Ações, as características mais significativas das ações semânticas definidas para a linguagem PEWS.

- execute _ : interface → Action

(1) execute ⟦ T: portType D: def* P: path ⟧ =
| elaborate ⟦T⟧
before elaborate ⟦D⟧
before execute ⟦P⟧

A semântica da função execute é definida sobre uma interface. Esta função é responsável pelas definições dos *bindings* das operações dos serviços e pelas variáveis inteiras usadas dentro dos predicados. Além disso, a função execute efetua também as ações estabelecidas pelas *path expressions* da interface.

- execute _ : path → Action

(1) execute ⟦ O:opname ⟧ =
| indivisibly
| | | | give the natural stored in the cell bound to act(O).val
| | | then store successor of it in the cell bound to act(O).val
| | and store current-time in the cell bound to act(O).$time$
and then
| enact the service bound to O
and then
| indivisibly
| | | | give the natural stored in the cell bound to term(O).val
| | | then store successor of it in the cell bound to term(O).val
| | and store current-time in the cell bound to term(O).$time$

A execução da operação de um serviço é precedida pela atualização do contador act, e em seguida pela atualização do contador term. Estas atualizações consistem em incrementar o componente *val* e pelo uso do *yielder* current-time, atualizando o componente *time* de cada contador. O *yielder* current-time não faz parte da Semântica de Ações padrão. Supõe-se que ele gerencia o tempo atual, absoluto e global do sistema. Este *yielder* representa uma chamada de função do sistema em uma implementação da vida real.

(2) execute ⟦ P_1:path "." P_2:path ⟧ = execute ⟦ P_1 ⟧ and then execute ⟦ P_2 ⟧

(3) execute ⟦ P_1:path "|" P_2:path ⟧ = execute ⟦ P_1 ⟧ or execute ⟦ P_2 ⟧

(4) execute ⟦ P_1:path "||" P_2:path ⟧ = execute ⟦ P_1 ⟧ and execute ⟦ P_2 ⟧

As equações acima definem diferentes composições de operações sobre as *path expressions*; denominadas, composição sequencial, escolha não-determinística e composição paralela.

(5) execute ⟦ P_1:path "*" ⟧ =
unfolding
| | complete
| or
| | execute ⟦ P_1 ⟧ and then unfold

(6) execute ⟦ "{" P_1:path "}" ⟧ =
unfolding
| execute ⟦ P_1 ⟧ and unfold

O operador de repetição (∗) define uma execução ilimitada, sequencial e repetitiva de uma *path expression* P_1. O operador {_} especifica a execução paralela ilimitada da *path expression* P_1.

(7) execute ⟦ "[" B:pred "]" P_1:path ⟧ =
unfolding
| | evaluate ⟦ B ⟧ and enabled ⟦P_1⟧
| then
| | | | check (the given tuple is ⟨true, true⟩)
| | | and then commit and then execute ⟦ P_1 ⟧
| | or
| | | | check not (the given tuple is ⟨true, true⟩)
| | | and then unfold

A definição acima considera o caso onde a execução da *path expression* P_1 depende do resultado da avaliação do predicado B. A fim de executar P_1, a ação espera até que as duas seguintes condições sejam simultaneamente verificadas: (i) a avaliação de B resulta em *true* e (ii) P_1 está pronto para executar. Esta segunda condição, especificada por enabled _, consiste em verificar se as mensagens de entrada estão disponíveis para serem lidas por P_1 (se P_1 começa com uma operação *response-request* ou *one-way*).

- elaborate _ : portType → Action

(1) elaborate ⟦Q_1:operation+ Q_2:operation+ ⟧ =
elaborate ⟦Q_1⟧ and elaborate ⟦Q_2⟧

(2) elaborate ⟦ O:opname "(" A:opArg ")" ⟧ =
| allocate a cell then
| | store 0 in it and bind it to act(O).*val*
and
| allocate a cell then bind it to act(O).*time*
and
| allocate a cell then
| | store 0 in it and bind it to term(O).*val*
and
| allocate a cell then bind it to term(O).*time*
and
| bind kindOf ⟦ A ⟧ to kind(O)
and
| *GetAbstractionForService*(O) then bind it to O

O *elaboration* do portType cria os contadores para cada operação (cada contador é formado por um par de células). A ação*GetAbstractionForService*_ representa a identificação de uma

abstração externa, que será associada com a operação do serviço. Em uma implementação da vida real, esta função é executada pelo servidor de HTTP.

- elaborate _ : def* → Action

(1) elaborate ⟦ ⟧ = complete

(2) elaborate ⟦ G_1: def* G_2: def* ⟧ = elaborate ⟦ G_1 ⟧ and elaborate ⟦ G_2 ⟧

(3) elaborate ⟦ "def" I:var "=" E:arith-expr ⟧ =
    bind I to closure abstraction of evaluate ⟦ E ⟧

As declarações de variáveis em PEWS são uma forma simples de definição de função. A expressão que define a variável será avaliada toda vez que a variável for usada durante a execução da *path expression*.

- evaluate _ : arith-expr → Action[giving an (integer | time)]
- evaluate _ : pred → Action[giving a truth-value]
- enabled _ : path Action[giving a truth-value]

(1) enabled ⟦ O:opname ⟧ =

```
indivisibly
| | check (the datum bound to kind(O) is a (out | out-in)) and then give true
| or
| | | check (the datum bound to kind(O) is a (in | in-out))
| | and then
| | | | choose a message[containing ⟨ O, data ⟩]
| | | |                    [in set of items of the current buffer]
| | | then
| | | | | check (it is a message) and then give true
| | | | or
| | | | | check (it is nothing) and then give false
```

(2) enabled ⟦ P_1:path "." P_2:path ⟧ = enabled ⟦ P_1 ⟧

(3) enabled ⟦ P_1:path "*" "." P_2:path ⟧ = enabled ⟦ P_1 ⟧ or enabled ⟦ P_2 ⟧

(4) enabled ⟦ P_1:path "|" P_2:path ⟧ = enabled ⟦ P_1 ⟧ or enabled ⟦ P_2 ⟧

(5) enabled ⟦ P_1:path "||" P_2:path ⟧ = enabled ⟦ P_1 ⟧ or enabled ⟦ P_2 ⟧

(6) enabled ⟦ P_1:path "*" ⟧ = enabled ⟦ P_1 ⟧

(7) enabled ⟦ "{" P_1:path "}" ⟧ = enabled ⟦ P_1 ⟧

(8) enabled $[\![$ "[" B:pred "]" P_1:path $]\!]$ = enabled $[\![$ P_1 $]\!]$

A função semântica enabled _ retorna um valor verdade, sendo *true* se a operação principal na expressão estiver habilitada para ser executada, isto é, se uma operação de saída ou saída/entrada ou se existe uma mensagem esperando por ela nos canais de comunicação.

3.1.3.4 XPEWS

O modelo formal disponibilizado pelas *path expressions* faz de PEWS uma linguagem para descrição de comportamento de serviços em um nível abstrato. Por outro lado, para se ter o armazenamento da descrição do serviço para publicação, pesquisa e composição, é preciso traduzir PEWS para uma representação concreta e padronizada. Para este fim, construiu-se uma versão XML de PEWS, chamada XPEWS. XPEWS é uma linguagem XML que estende a descrição das interfaces de um documento WSDL pela adição de restrições de comportamento. Os programas em XPEWS podem ser gerados de maneira intuitiva. Por exemplo, o seguinte programa PEWS

def t^{pgto} = now() - term(enviarConta).$time$

(fazerPedido.enviarConta.([$t^{pgto} \leq 48h$] enviarPagamento.emitirRecibo + [$t^{pgto} > 48h$] abortarOperação))*

é traduzido em XPEWS como ilustra a Figura 3.2. A tradução de PEWS para XPEWS segue a definição do esquema de XPEWS, definido em XML Schema, ilustrado pelo Apêndice A.1. O elemento raiz, definido na linha 1, de um documento XPEWS é o `<envelope>`. Ele pode conter definições dos possíveis comportamentos do serviço Web. Por exemplo, ao se definir o acesso aos dados, pode-se especificar diferentes ordens nas quais as instâncias destes dados podem ser acessadas. A Figura 3.2 ilustra apenas um comportamento para o serviço loja. Ele verifica uma condição de *timeout* de 48 horas entre o envio de uma conta e a execução do pagamento. Na linha 2, o elemento `<behaviour>` especifica uma maneira na qual o cliente pode interagir com o serviço Web.

```
1  <envelope xmlns="http://aquarius.inf.ufpr.br" xmlns:xsi="http://www.
      w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
      aquarius.inf.ufpr.br pews.xsd"
2  <behaviour name="Loja" xmlns:loja="http://localhost:8080/axis/samples
      /loja/loja.wsdl">
3    <operations>
4      <operation name="fazerPedido" portType="loja" refersTo="
          loja:fazerPedido"/>
5      <operation name="enviarConta" portType="loja" refersTo="
          loja:enviarConta"/>
6      <operation name="enviarPagamento" portType="loja" refersTo="
          loja:enviarPagamento"/>
7      <operation name="emitirRecibo" portType="loja" refersTo="
          loja:emitirRecibo"/>
8    </operations>
9   <vardef name="tPgto">
10   <minus>
11    <libFunction name="now" unit="hours"/>
12    <pewsCounter opname="enviarConta" name="term" component="time"
         unit="hours"/>
13   </minus>
14  </vardef>
15  <pathExp>
16   <star>
17    <seq>
18     <operation name="fazerPedido" />
19     <operation name="enviarConta" />
20     <choice>
21      <seq>
22       <pred>
23        <leq>
24         <var name="tPgto" />
25         <const value="48" />
26        </leq>
27        <operation name="enviarPagamento" />
28       </pred>
29       <operation name="emitirRecibo" />
30      </seq>
31      <pred>
32       <gt>
33        <var name="tPgto" />
34        <const value="48" />
35       </gt>
36       <operation name="abortarOperação" />
37      </pred>
38     </choice>
39    </seq>
40   </star>
41  </pathExp>
42 </behaviour>
43 </envelope>
```

Figura 3.2: Programa XPEWS referente à tradução do serviço loja.

Isso é feito por uma *predicate path expression*, a qual envolve as operações do serviço. Note que o elemento `<behaviour>` tem dois atributos. O atributo `nome` identifica o elemento. O segundo atributo é um *namespace*, que faz referência a um documento WSDL, o qual define as operações que serão usadas pelo programa XPEWS. Na linha 3, a *tag* `operations` agrupa as operações usadas na composição. Esta *tag* possui um atributo chamado *refersTo*, que associa o *namespace* utilizado pela operação, declarado pela *tag* `behaviour`. A linha 9 mostra a definição de uma variável, cujo valor é dado pela função `now()` (hora corrente). Na linha 15, a tag `pathExpr` contém as tags que descrevem a ordem de execução das operações definidas pelas *Paty expressions*.

3.1.3.5 Coreografia

Na Seção 4.2.1, viu-se o modelo utilizado para executar uma operação entre dois serviços (processos), o uso dos canais de comunicação e um exemplo de documento WSDL dos respectivos serviços. O próximo passo será mostrar o uso de PEWS na composição de serviços Web. Hull [34] e Salaün [58] demonstraram, em comum, duas abordagens principais para compor serviços Web: a `coreografia` (*peer-to-peer*) e a `orquestração` (*mediated*). PEWS fornece suporte para as duas abordagens, sendo que os detalhes sobre a orquestração serão vistos na Seção 3.1.3.6.

A coreografia é o problema de se garantir que serviços Web possam interagir apropriadamente, a fim de cooperarem para resolver uma tarefa específica. O seguinte cenário, ilustrado pela Figura 3.3, mostra uma composição contendo várias operações executando concorrentemente.

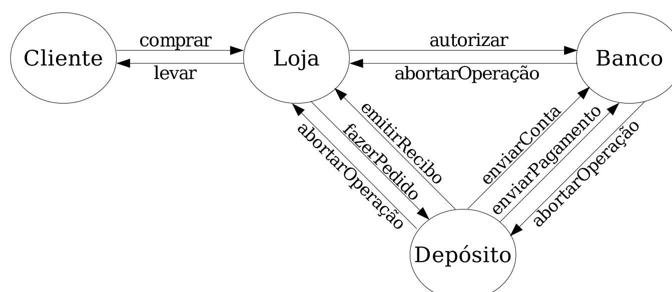


Figura 3.3: Especificação da composição de serviços Web através da coreografia

O objetivo desta composição é aprovar (ou não) um pedido de compra feita pelos clientes de uma loja. Uma restrição feita é que o pagamento deve ser feito dentro de 48 horas após o envio da conta. A Figura 3.3 mostra os quatro serviços envolvidos na composição (*Cliente*, *Loja*, *Depósito* e *Banco*), bem como as operações, tais como comprar, autorizar e enviarPedido. As setas descrevem as ligações (sincronizações) entre os serviços. Os detalhes da implementação de cada serviço são ocultos, estando disponíveis ao programador somente a descrição da interface do serviço, especificada pelo padrão WSDL. Dada uma especificação WSDL do serviço, a descrição de sua interface provê informações que habilitam o descobrimento, o método de acesso e a execução de uma operação de um serviço. Porém a ordem em que as operações serão executadas é dada por um programa PEWS. Considere o serviço *Cliente* mostrado pela Figura 3.3. Neste cenário, um cliente da loja requisita uma compra e se esta for aceita ele pode levar os produtos. Um programa PEWS que expressa esta situação é o seguinte:

$\mathsf{Cliente} = \mathsf{comprar}^{\oplus\ominus}\ .\ \mathsf{levar}^{\ominus}$

Por uma questão de legibilidade, a cada operação foi associado o seu tipo, dados pelos símbolos $^{\oplus}$ e $^{\ominus}$. O símbolo $^{\oplus}$ indica que a operação possui uma mensagem de saída e o símbolo $^{\ominus}$ indica que a operação possui uma mensagem de entrada. A combinação entre eles mostra uma operação que possui uma mensagem de *entrada/saída* ($^{\ominus\oplus}$) e vice-versa. Este serviço é visto como um "cliente" dos outros serviços envolvidos na coreografia, pois é ele quem irá iniciar o processo da composição. O operador `seqüência` (.) define que primeiro será executada a operação comprar e em seguida a operação levar. Para executar uma operação, um serviço envia uma mensagem, correspondente a uma operação, para outro serviço. Quando o serviço recebe a mensagem, ele executa a operação. Sendo assim, o processo inicia-se quando o *Cliente* envia uma mensagem através do canal de saída da operação comprar e, em seguida, ficará bloqueado aguardando por uma mensagem no canal de entrada da operação comprar. Note que isto ocorreu devido ao tipo da operação (saída/entrada). Quando a mensagem for recebida, o serviço é acordado e a instrução seguinte será executar a operação levar. Para executá-la, o processo irá esperar por uma uma mensagem no canal de entrada da respectiva operação. A seguir, serão descritos os

demais serviços. Por simplicidade, os detalhes das trocas de mensagens serão omitidos, pois eles estão implícitas nas operações e ocorrem de maneira semelhante ao exemplo mostrado pelo serviço *Cliente*.

O próximo serviço a ser codificado em PEWS, o serviço *Loja*, gerencia as requisições de compras enviadas pelos seus clientes. Ao receber uma nova requisição(mensagem) correspondente à operação comprar, esta será executada e em seguida inicia-se uma sincronização com o serviço *Banco*, através da operação autorizar. Se a compra for autorizada, geram-se os pedidos de compra a serem enviados ao *Depósito* e espera-se por receber os respectivos recibos. Se o banco não autorizar a compra, a operação abortarOperação é executada, e a composição como um todo é interrompida. A operação abortarOperação é uma operação interna da linguagem PEWS, não sendo implementada por um serviço Web. Um programa PEWS que expressa o comportamento do serviço *Loja* é mostrado a seguir:

$$\mathsf{Loja} = \mathsf{comprar}^{\ominus\oplus}.\mathsf{autorizar}^{\oplus\ominus}.\mathsf{fazerPedido}^{\oplus\ominus}.\ \mathsf{emitirRecibo}^{\ominus\oplus}.(\mathsf{levar}^{\oplus\ominus}+\mathsf{abortarOperação}^{\ominus})$$

Agora, será descrito o comportamento do serviço *Depósito*. Este serviço recebe pedidos de compra de uma loja e, em seguida, envia a conta e o pagamento para o banco executá-los. Se o pagamento for realizado dentro de um prazo estipulado, por exemplo, dentro de 48 horas, os recibos dos pedidos de compra são enviados para a loja. Caso contrário, a operação abortarOperação é executada. A seguir, mostra-se um programa PEWS que descreve esse comportamento.

$$\mathsf{def}\ t^{pgto} = \mathsf{now()\ -\ term(enviarConta)}.time$$

$$\mathsf{Depósito} = \begin{array}{l} \mathsf{fazerPedido}^{\ominus\oplus}.\mathsf{enviarConta}^{\oplus\ominus}. \\ ([t^{pgto} \leq 48h]\mathsf{enviarPagamento}^{\oplus\ominus}.\mathsf{emitirRecibo}^{\oplus\ominus} \\ \mid [t^{pgto} > 48h]\ \mathsf{abortarOperação}^{\oplus}) \end{array}$$

Nesse exemplo, os predicados que aparecem no programa representam guardas, que expressam a semântica do contrutor de `escolha exclusiva` (|), mostrando que o pagamento deve ser feito dentro de 48 horas, após a conta ser enviada.

Por fim, será descrito o comportamento do serviço *Banco*. Este serviço irá receber requisições pedindo a autorização de clientes que estão efetuando compras em uma *Loja*.

Em seguida, o banco receberá as contas para efetuar os pagamentos. Se o pagamento for realizado dentro de um prazo estipulado, então a composição pode prosseguir. Caso contrário, todo o fluxo é interrompido. A seguir, mostra-se um programa PEWS que ilustra esse comportamento.

def t^{pgto} = now() - term(conta).$time$

Banco = autorizar$^{\ominus\oplus}$.enviarConta$^{\ominus\oplus}$.
([$t^{pgto} \leq 48h$] enviarPagamento$^{\ominus\oplus}$ | [$t^{pgto} > 48h$] abortarOperação$^{\oplus}$)

Uma vez que os serviços foram codificados individualmente, é preciso fazer a interação entre eles. Visto que os serviços são processos, a execução deles deve ocorrer em paralelo, de modo que cada serviço irá executar seu próprio programa PEWS e a troca de mensagens através dos canais irão coordenar a concorrência. Um programa PEWS que expressa essa situação é dado a seguir:

Composição = (Cliente || Loja || Depósito || Banco)

Essa forma de composição possui uma limitação, ela aceita apenas um pedido de compra, faz o processamento e encerra a composição. Porém em um sistema real, é interessante que o sistema possa aceitar várias requisições e processá-las. Uma maneira é fazer com que a composição possa executar os pedidos de compra em seqüência, à medida em que eles forem chegando. Um programa PEWS que expressa esta situação é dado a seguir:

Composição = (Cliente || Loja || Depósito || Banco)*

Esse programa possibilita que um cliente possa requisitar tantos pedidos quanto ele quiser, porém eles serão processados um por vez. Uma outra forma de expressar a composição é descrita pelo seguinte programa PEWS:

Composição = { Cliente || Loja || Depósito || Banco }

Nesta situação, o cliente pode requisitar tantos pedidos quanto ele quiser, sendo que para cada requisição que chegar uma nova *thread* irá processar os pedidos em paralelo.

3.1.3.6 Orquestração

Nesta seção, o objetivo será mostrar como PEWS pode ser usado para especificar a composição de serviços Web através da orquestração. A orquestração é o problema de se

resolver problemas complexos usando-se serviços simples, através do desenvolvimento de um novo serviço, chamado de *orquestrador* ou *mediador*. O mediador é um novo serviço que irá coordenar os serviços participantes e a ordem de execução das operações, como ilustra a Figura 3.4.

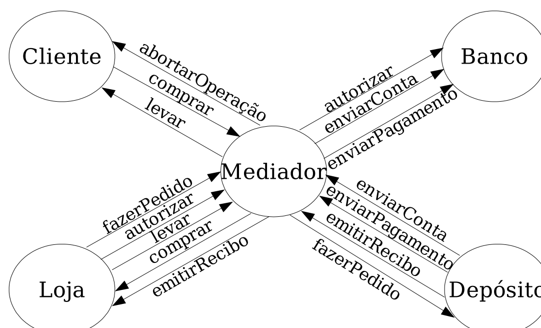


Figura 3.4: Especificação da composição de serviços Web através da orquestração

Esse exemplo é o mesmo que foi utilizado pela coreografia, ou seja, o objetivo da composição é aprovar (ou não) os pedidos de compra requisitados pelos clientes da loja. Como restrição, o pagamento deve ser feito dentro de 48 horas após o envio da conta. Um programa PEWS que codifica o comportamento do mediador é dado a seguir:

def t^{pgto} = now() - term(enviarConta).$time$

Mediador =
comprar$^{\ominus\oplus}$.autorizar$^{\ominus\oplus}$.fazerPedido$^{\ominus\oplus}$.enviarConta$^{\ominus\oplus}$.
([$t^{pgto} \leq 48h$] enviarPagamento$^{\ominus\oplus}$.emitirRecibo$^{\ominus\oplus}$
| [$t^{pgto} > 48h$] abortarOperação$^{\oplus}$).levar$^{\oplus}$

O mediador, o qual deverá fazer com que os serviços trabalhem em conjunto, a fim de resolverem um problema, recebe os pedidos de compra feitos pelos clientes da loja e troca informações com os demais serviços. Para que um serviço possa executar uma operação de outro serviço, uma mensagem deverá antes passar pelo mediador, e ele delegar para o serviço correspondente. Por exemplo, para que o serviço *Cliente* faça a requisição de uma compra, uma mensagem correspondente à esta operação (comprar) deve ser enviada ao mediador e ele irá repassá-la ao serviço que irá executar a operação, no caso o serviço *Loja*.

CAPÍTULO 4

PROGRAMAÇÃO CONCORRENTE

A primeira tarefa verificada na etapa de implementação da linguagem, foi estudar como seriam implementadas as primitivas de sincronização de PEWS. Foi visto na seção 3.1.3 que PEWS faz uso dos operadores de sincronização das *predicate path expressions* para combinar as operações dos serviços Web. Implementar os operadores de *paralelismo* ou *escolha exclusiva*, requer a implementação de primitivas de controle de concorrência. Por isso, serão abordados alguns aspectos importantes da programação concorrente e a sua aplicabilidade na implementação da linguagem PEWS.

Um *programa sequencial* especifica a execução sequencial de uma lista de instruções, sendo sua execução chamada de *processo* [5]. Um *programa concorrente* especifica que dois ou mais programas sequenciais podem ser executados paralelamente como *processos paralelos*. Por exemplo, um sistema de reservas de passagens que envolve o processamento de várias transações de diferentes operadoras possui naturalmente uma especificação como um programa concorrente, no qual cada operadora é controlada por seu próprio processo sequencial. Mesmo quando os processos não são executados simultaneamente, muitas vezes é mais fácil estruturar um sistema como uma coleção de processos sequenciais concorrentes, em vez de um único processo sequencial.

Um programa concorrente pode ser executado tanto pela possibilidade de processos compartilharem um ou mais processadores, quanto por executarem em seu próprio processador. A primeira abordagem refere-se à *multiprogramação*, na qual o sistema operacional gerencia o escalonamento dos processos no processador (ou processadores). A segunda abordagem refere-se ao *multiprocessamento*, se os processadores compartilham uma memória em comum, ou ao *processamento distribuído* se os processadores estão conectados por uma rede.

4.1 Sincronização entre Processos

O conceito fundamental de programação concorrente é o conceito de *processo*. A cooperação entre processos concorrentes acontece através de comunicação e sincronização. A comunicação é o que permite que a execução de um processo influencie a execução de outro. A comunicação interprocessos pode ser realizada pelo compartilhamento de variáveis ou por troca de mensagens [31].

Para que ocorra a comunicação, um processo deve executar uma ação que seja detectada pelo outro, como a alteração de uma variável compartilhada ou o envio de uma mensagem. A execução dos processos progride com velocidades variáveis e imprevisíveis. Isso torna necessário o uso de um mecanismo de sincronização para garantir que os eventos "executa a ação" e "percebe a ação" aconteçam nesta ordem. Esta dissertação dará enfoque à primitiva de concorrência troca de mensagens, cujas operações básicas são "enviar a mensagem" e "aceitar a mensagem", sendo ambas usadas para comunicação e sincronização. A sincronização é um efeito colateral da comunicação: uma mensagem só pode ser recebida depois de ter sido enviada.

Um *canal de comunicação* deve ser estabelecido entre a fonte e o destino das mensagens, isto é, entre o processo que envia as mensagens e o que as recebe. A comunicação *direta*, se os nomes dos processos fonte e destino devem ser especificados para que o canal seja estabelecido. A comunicação direta pode ser implementada de uma forma muito simples, mas é também muito pouco flexível. Em relações do tipo *cliente/servidor*, onde um ou mais processos usam os serviços proporcionados por um ou mais servidores, um canal deve ser estabelecido entre cada um dos possíveis pares cliente/servidor.

As primitivas que implementam as trocas de mensagens podem ser do tipo *bloqueante* ou *não-bloqueante*. Uma primitiva do tipo bloqueante pode suspender temporariamente a execução do processo que a invocou; isto nunca acontece com primitivas não-bloqueantes. Quando as mensagens são enfileiradas entre o envio e a recepção, o envio de mensagens é dito *assíncrono*, isto é, o processo que envia as mensagens nunca é bloqueado (pelo menos, enquanto houver espaço na fila de mensagens). O envio assíncrono permite que a execução do processo fonte das mensagens progrida independentemente da execução do

processo destino. Neste caso, as mensagens podem conter informações desatualizadas. Se as mensagens não forem enfileiradas, o envio deve ser bloqueante. O processo que envia uma mensagem fica bloqueado até que ela seja recebida pelo processo destino. Esta forma de sincronização é dita síncrona. Na comunicação síncrona, as mensagens sempre contêm informações atualizadas.

4.2 Serviços Web e Processos

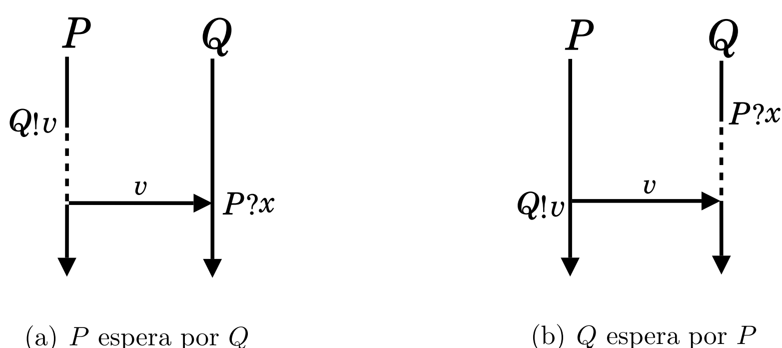
Se considerarmos os serviços Web como processos e dado que a comunicação entre os serviços é feita atráves de troca de mensagens, faz-se necessário o uso de uma ferramenta que possa auxiliar a linguagem PEWS na implementação dos processos e da comunicação. Uma ferramenta que possui a noção de processos que se comunicam por troca de mensagens é a linguagem *Communicating Sequential Process* (CSP) [33] (veja a seção 4.2.1). Visto que a linguagem Java foi escolhida para implementar o sistema de tempo de execução, uma implementação Java da CSP é descrita na seção 4.2.2.

Além da noção de processos e da comunicação através de mensagens, a CSP possui também os operadores de `seqüência`, `paralelismo`, `escolha seletiva` e `comandos com guarda`, os quais são semanticamente equivalentes aos operadores `seqüência` (.), `paralelismo` (||), `escolha exclusiva` (|) e prefixo de predicado (([. . .] . . .) de PEWS, respectivamente. Sendo assim, a implementação Java da CSP será responsável por implementar os processos, a comunicação e os operadores de PEWS. Os demais operadores (de PEWS), `repetição` (*) e `repetição paralela` ({. . .}) não são mapeados de maneira direta, os quais serão implementados pelo sistema de execução.

4.2.1 CSP

Uma característica que a linguagem CSP [33] possui é a noção de um conjunto de processos sequenciais executando em paralelo e comunicando-se através de troca de mensagens síncronas [57]. A linguagem possui operações de entrada, $P?x$ (leia o valor do processo P e armazene-o na variável x); operações de saída, $P!e$ (envie o valor da expressão e para

o processo P), e, um comando para criar processos. Os processos e a estrutura de comunicação entre eles é estática, visto que não há criação dinâmica de processos e os "nomes" dos processos são usados para nomear as comunicações. Se um processo P executa $Q!v$, ele deve permanecer bloqueado até que o processo Q execute $P?x$ (e vice-versa). Dois processos comunicam-se se houver uma combinação dos pares de operações *saída/entrada*, como ilustra a Figura 4.1. As linhas pontilhadas de tempo, mostradas pela figura, representam o período de ociosidade enquanto há a espera pela combinação entre as operações *saída/entrada*.



(a) P espera por Q (b) Q espera por P

Figura 4.1: Combinação entre as operações entrada/saída [57]

Uma das idéias principais da linguagem CSP é a noção de *comandos com guarda*, no qual as operações de entrada são habilitadas como guardas. Um comando com guarda com uma operação de entrada será selecionada para execução se a operação de entrada estiver pronta para executar a operação de saída correspondente. Se mais de uma operação de entrada na guarda estiver pronta para executar, então uma delas é escolhida não-deterministicamente. Esse mecanismo possibilita que processos possam comunicar-se uns com os outros, quando a ordem das comunicações é desconhecida. Por exemplo, um processo "servidor" que recebe requisições de muitos clientes pode não saber qual cliente irá enviar a próxima requisição.

Lembrando da associação de serviços Web com processos e do uso de canais de comunicação, pode-se agora modelar uma arquitetura para a execução das operações de serviços Web. Embora a especificação WSDL descreva os modelos para comunicação entre serviços, tais como a comunicação *one-way* ou *request-response*, os detalhes de como

conectar tais serviços não são descritos. Para tanto, criou-se um modelo para o problema, que é mostrado pela Figura 4.2.

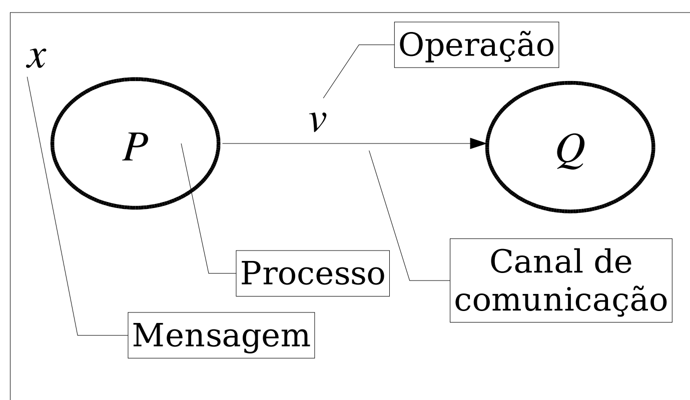


Figura 4.2: Procedimento utilizado para a execução de operações

A Figura 4.2, mostra uma composição simples entre dois serviços (processos) P (o "cliente") e Q (o "servidor"). A seta indica qual operação será executada, associada a um canal de comunicação síncrono. Um canal ligando P a Q é definido como *saída/entrada*, representando uma troca de mensagens do tipo *enviar/receber*. A fim de executar a operação, P deve enviar uma mensagem pelo canal e Q deverá lê-la; O processo para executar uma operação envolve os seguintes passos:

1. Armazenar o dado a ser transferido em uma estrutura de dados local x

2. P envia x através do canal de comunicação

3. Q lê x através do canal de comunicação

4. A operação v é executada por Q

Porém existe a possibilidade da operação retornar algum valor. Neste caso, haverá um canal adicional ligando Q a P, onde Q irá enviar o valor de retorno e P irá recebê-lo. Dada esta configuração, os documentos WSDL de P e Q serão descritos, como ilustram as Figuras 4.3 e 4.4.

Exemplo 4.2.1 *A Figura 4.3 mostra um documento WSDL que descreve a interface do serviço Web P, o qual foi descrito pela Figura 4.2.*

```
1  <wsdl:message name="vOut">
2    <wsdl:part name="v_id" type="xsd:int"/>
3    <wsdl:part name="v_total" type="xsd:real"/>
4  </wsdl:message>
5  <wsdl:message name="vIn">
6    <wsdl:part name="v_ok" type="xsd:string"/>
7  </wsdl:message>
8  <wsdl:portType name="P">
9    <wsdl:operation name="v">
10     <wsdl:output message="impl:vOut" name="vOut"/>
11     <wsdl:input message="impl:vIn" name="vIn"/>
12   </wsdl:operation>
13 </wsdl:portType>
```

Figura 4.3: Especificação WSDL para o serviço P

A linha 9 mostra o nome da operação (v), enquanto as linhas 10 e 11 definem o tipo da operação (*saída/entrada*). Sendo assim, P terá dois canais de comunicação: um para o canal de saída e outro para o canal de entrada. O canal de saída será usado para transmitir uma mensagem a fim de executar a operação v, desde que aconteça uma combinação entre os tipos das mensagens da operação de P e do outro serviço. O tipo da mensagem de saída está descrito nas linhas 1 a 4. Por outro lado, o canal de entrada será usado para ler o valor de retorno do processo que executou v.

Exemplo 4.2.2 *A Figura 4.4, mostra um documento WSDL que descreve a interface do serviço Web Q, o qual foi descrito pela Figura 4.2.*

```
1  <wsdl:message name="vIn">
2    <wsdl:part name="v_id" type="xsd:int"/>
3    <wsdl:part name="v_total" type="xsd:real"/>
4  </wsdl:message>
5  <wsdl:message name="vOut">
6    <wsdl:part name="v_ok" type="xsd:string"/>
7  </wsdl:message>
8  <wsdl:portType name="Q">
9    <wsdl:operation name="v">
10     <wsdl:input message="impl:vIn" name="vIn"/>
11     <wsdl:output message="impl:vOut" name="vOut"/>
12   </wsdl:operation>
13 </wsdl:portType>
```

Figura 4.4: Especificação WSDL para o serviço Q

A linha 9 mostra o nome da operação (v), enquanto as linhas 10 e 11 definem o tipo da operação (*entrada/saída*). Sendo assim, Q terá dois canais de comunicação: um para o canal de entrada e outro para o canal de saída. O canal de entrada será usado para ler uma mensagem e executar a operação v, desde que aconteça uma combinação entre os tipos das mensagens da operação de Q e do serviço que enviou a mensagem (P). O tipo da mensagem de saída está descrita nas linhas 1 a 4. Note que existe uma compatibilidade para executar a operação v, entre a mensagem de saída de P, a mensagem de saída de Q. Há também uma compatibilidade para o retorno da operação v, dado pela mensagem de saída de Q e pela mensagem de entrada de P.

4.2.2 Implementação Java da CSP

A linguagem de programação Java possui primitivas para programação concorrente e paralela, tais como *threads*, monitores, *sockets* e *Remote Method Invocation* (RMI) [59]. No entanto, há uma grande preocupação sobre a maneira no qual este suporte é disponibilizado. Hansen [30] e Welch [71] citam alguns problemas, tais como a implementação incorreta de monitores e a dificuldade de se programar com *threads*. As primitivas de sincronização em Java são de baixo nível, inseguras e difíceis de serem usadas corretamente. Por exemplo, o modelo de monitores implementado por Java, embora seja fácil entender suas primitivas, demonstra ser muito difícil de ser aplicado com confiança em sistemas de nível de complexidade médio ou superior.

A linguagem proposta por Hoare, denominada *Communicating Sequential Processes* (CSP) [33], especifica completamente a sincronização de *threads*, fundamentando-se em processos, composições e canais de comunicação. A notação matemática provida pelo CSP descreve padrões de comunicação usando expressões algébricas e contém provas formais para analisar, verificar e eliminar condições indesejáveis, tais como *race hazards*, *deadlocks*, *livelock* e *starvation*.

Vários pesquisadores têm investigado o uso da CSP para gerenciar a concorrência de *threads* em Java. Como resultado, duas bibliotecas foram implementadas para esta finalidade: a *Communicating Threads for Java* (CTJ) [32], desenvolvida pela Universidade

Twente e a *Communicating Sequential Processes for Java* (JCSP), desenvolvida pela Universidade de Kent [72]. Elas são muito semelhantes em suas funcionalidades no entanto, para esta dissertação, será discutida as funcionalidades da biblioteca JCSP.

Um processo em CSP é um componente que encapsula estrutura de dados e algoritmos para manipular dados. Os Algoritmos e os dados são ambos privados. Sendo assim, outros processos não podem ver os dados nem executar os algoritmos. Cada processo está ativo, executando seu próprio algoritmo sobre seus próprios dados. Os processos interagem somente através de primitivas de sincronização, tais como os canais, e não através de chamadas a métodos de um ou de outro processo.

A biblioteca JCSP tem como base para sua implementação as primitivas da CSP. Ela habilita que sistemas *multithreads* possam ser projetados, implementados e raciocinados inteiramente sobre as primitivas da CSP, as quais serão discutidas no decorrer desta seção. Um exemplo de como utilizar a JCSP para implementar a comunicação interprocessos, é ilustrado na Figura 4.5.

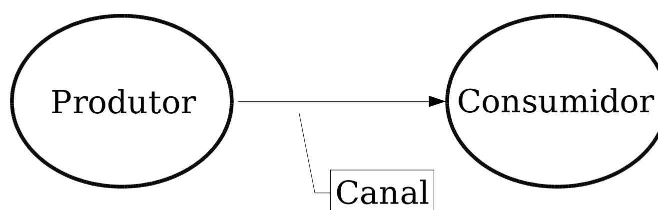


Figura 4.5: Processo Produtor-Consumidor

A Figura 4.5 mostra dois processos, implementando um esquema de processos Produtor-Consumidor. O Produtor e o Consumidor estão conectados por um canal que provê um meio de comunicação sobre o qual os dois processos podem interagir. Em JCSP, cada processo é uma instância da classe que implementa a interface *CSProcess*, como mostra o exemplo a seguir:

```
public interface CSProcess {
  public void run();
}
```

O exemplo mostra que o comportamento de um processo é definido pelo código do método `run()`, da classe que será implementada.

A maneira mais fácil de interagir processos é pela sincronização de troca de mensagens através do uso de canais. Os canais não possuem a capacidade de armazenar dados, sendo apenas um meio de transmití-los. Para evitar a perda de dados não detectados, a comunicação no canal é sincronizada. Isto significa que se o Produtor transmitir antes que o Consumidor estiver pronto, então o Produtor irá ficar no estado de bloqueado. De maneira análoga, se o Consumidor tentar receber dados antes do Produtor transmití-los, o Consumidor irá permanecer no estado de bloqueado. Quando ambos estiverem prontos, o dado é transferido diretamente do Produtor para o Consumidor. O seguinte fragmento de código mostra um exemplo de como transmitir uma mensagem que é um número inteiro através do uso de canais.

```
1 // Classe Produtor
2 ChannelOutputInt out;
3 out.write(100);
```

```
1 // Classe Consumidor
2 ChannelInputInt in;
3 valor = in.read();
```

No fragmento de código acima, a linha 2 mostra a declaração dos canais de comunicação para as classes Produtor e Consumidor. A linha 3 da classe Produtor mostra o valor inteiro sendo transmitido pelo canal de escrita, enquanto que a linha 3 da classe Consumidor mostra o valor inteiro 3 sendo recebido pelo canal de leitura.

Em adição aos canais de comunicação, a biblioteca disponibiliza alguns construtores para a coordenação de vários serviços concorrentes. Por exemplo, para executar os processos em seqüência, pode-se usar o construtor *Sequence*. O construtor *Parallel* pode ser usado para executar os processos em paralelo. Para tanto, ao invés de usar os construtores de Java para implementar o paralelismo interprocessos, usa-se o construtor *Parallel* da JCSP, sendo que o uso dos canais eliminam a necessidade direta do uso de monitores para controlar a concorrência entre processos [46].

O acesso a vários processos simultaneamente é um dos problemas chave da computação concorrente. JCSP fornece suporte para tratar este problema, através do construtor *Alternative*. Esse é um importante construtor, o qual é definido por um array de eventos, chamados de guardas em JCSP, cuja finalidade é esperar e escolher eventos. Quando um

evento é ativado, o operador de seleção deste comando irá retornar o seu índice no array e uma ação correspondente será executada. Se mais de um evento estiver ativado, então um deles será escolhido não-deterministicamente.

CAPÍTULO 5

IMPLEMENTAÇÃO

No trabalho introduzido em [10], uma nova abordagem foi usada para definir o comportamento dos serviços Web. Nesse trabalho, propõe-se o uso das *predicate path expressions* [4] para restringir a ordem das possíveis seqüências das operações de um serviço Web. Para este fim, foi definida uma nova linguagem para descrição de interface, chamada PEWS, sendo uma ferramenta para descrever a composição dos serviços Web.

Foi criada também uma versão XML de PEWS, chamada XPEWS, fazendo com que PEWS possa ser utilizada tanto no nível abstrato quanto no nível de interface das linguagens para serviços Web. Detalhes da classificação dos níveis das linguagens foram feita apresentados na Seção 3.1. No entanto, o trabalho de Ba et al. [10] deixou em aberto, entre outras, a implementação do serviço usando-se XPEWS, de forma a utilizar a linguagem também no nível concreto.

Para implementar a composição em PEWS, inicialmente foram analisadas duas alternativas: criar um protótipo BPEL a partir de XPEWS, realizando o mapeamento das funcionalidades de XPEWS para BPEL, ou criar um conjunto de classes em Java que implementam a composição do serviço. Com relação ao primeiro enfoque, verificou-se que os programas em XPEWS descrevem a composição de maneira mais sucinta, sendo que a tradução de XPEWS para BPEL resultaria em um programa incompleto e mais complexo, necessitando da intervenção do usuário para completar o código final gerado. Isto acontece porque em uma composição em PEWS é verificado apenas as possíveis ordens de execução das operações, sendo que as mensagens trocadas entre os serviços estão implícitas nas operações. Por outro lado a composição em BPEL leva em consideração além da ordenação da execução das operações, as trocas de mensagens entre os serviços. Por isso o enfoque utilizado para implementar a composição será através da construção de um conjunto de classes Java. A implementação da composição em PEWS foi toda de-

senvolvida na linguagem Java sobre a plataforma Linux. As próximas seções apresentam os principais aspectos da implementação.

5.1 Arquitetura

As etapas iniciais da implementação da linguagem PEWS foram a especificação de XPEWS e a construção de modelo de arquitetura para o sistema de execução da linguagem.No trabalho introdutório sobre a linguagem PEWS [10], mostrou-se um esboço de uma versão XML de PEWS, denominada XPEWS, o qual representa uma árvore sintática, no formato XML, dos construtores da linguagem PEWS. A etapa de geração de código, descrita na Seção 5.2, utiliza essa árvore sintática para gerar as classes Java que implementam a composição. Porém esse esboço de XPEWS precisava de uma descrição mais detalhada e padronizada. Para essa finalidade, uma contribuição dada por essa dissertação foi a definição de um esquema para XPEWS, descrito no Apêndice A.1. A linguagem XPEWS estende as funcionalidade da WSDL de uma maneira independente. XPEWS incorpora aos serviços Web a possibilidade de coordenar a execução das operações dos serviços Web existentes, sem a necessidade de mudanças nas definições WSDL das interfaces dos mesmos.

Definido o esquema de XPEWS, foi desenvolvida uma arquitetura para implementar o ambiente de tempo de execução da linguagem PEWS, mostrada pela Figura 5.1.

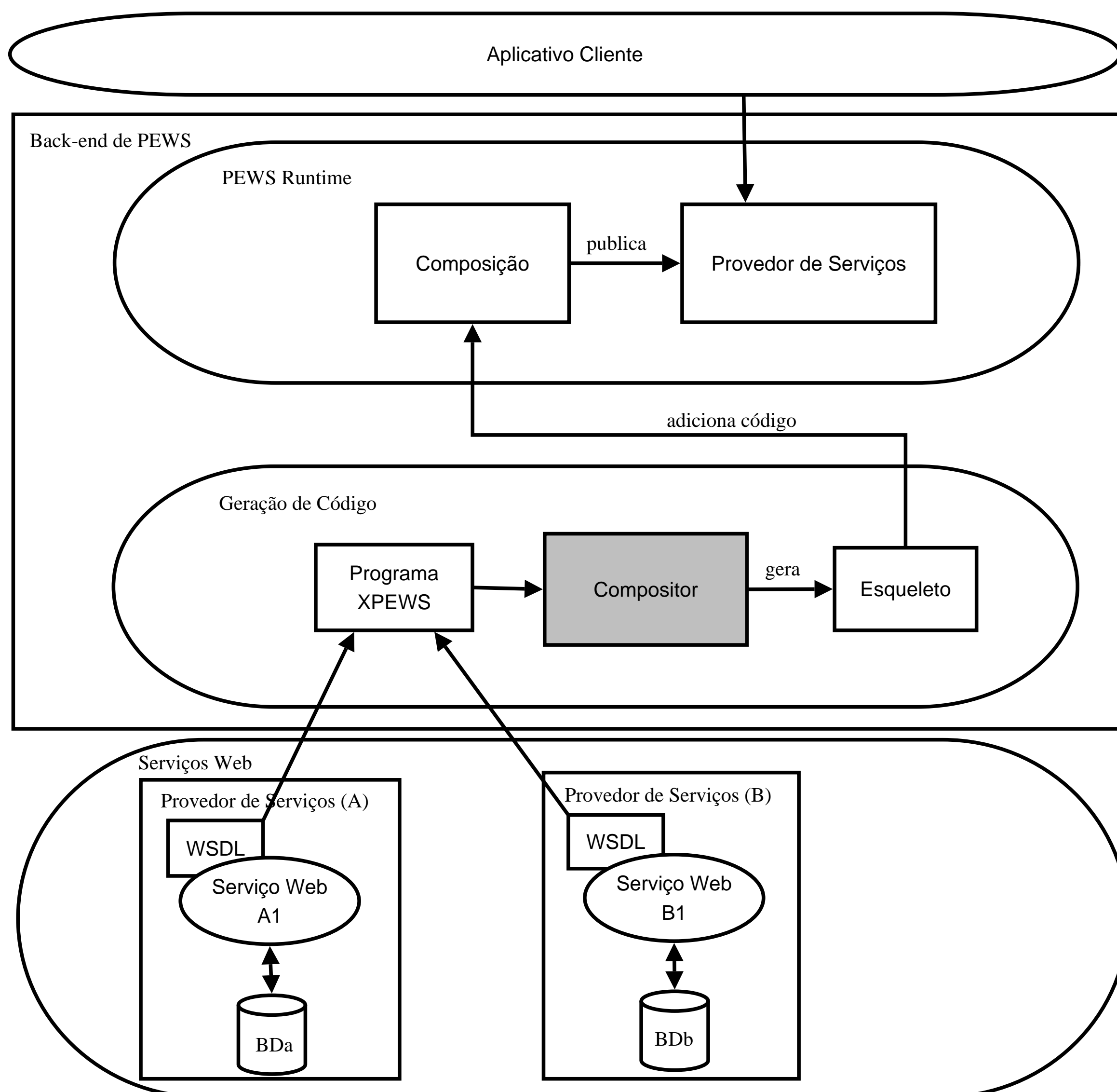


Figura 5.1: Arquitetura do *back-end* da linguagem PEWS.

A Figura 5.1 mostra uma visão em camadas do modelo da arquitetura proposta para realizar a composição em PEWS, cada qual contendo componentes responsáveis por descrever a dinâmica de sua respectiva camada. Na camada inferior têm-se os serviços Web, possivelmente disponibilizados por diferentes provedores, juntamente com suas respectivas descrições em WSDL, que servem como base no processo de composição de serviços em PEWS. A camada de `geração de código` é formada pelo módulo `Compositor`, que é o responsável pela etapa de geração de código da linguagem PEWS. Detalhes sobre o método utilizado na geração de código será visto na Seção . O módulo `Compositor` utiliza uma descrição do comportamento da execução das operações dos serviços Web,

dada por um programa XPEWS, e gera um esqueleto de classes Java que implementam a composição do serviço. O esqueleto de classes implementa a composição como um conjunto de processos, que representam os serviços Web, onde a execução da operação de um serviço será feita utilizando o modelo proposto na Figura 4.2. Nesse modelo, é preciso que um processo envie uma mensagem a outro processo que possui a operação a ser executada. Por isso é necessário que o programador adicione código ao esqueleto, por exemplo, geração dos dados que serão usados nas trocas de mensagens.

Na camada PEWS *runtime*, mostra-se o resultado da composição sendo registrado em provedor de serviços. A composição é uma extensão do esqueleto de classes Java, através da adição de código feita pelo programador. As classes Java que representam a composição, denominada PEWS *runtime*, são então registradas em um provedor de serviços, ficando disponível para que programas clientes possam utilizar suas funcionalidades.

5.2 Geração de Código

Tradicionalmente na construção de compiladores, o *front-end* da linguagem traduz o programa fonte para uma representação intermediária, a partir da qual o *back-end* gera o código final. Desta maneira, o *front-end* terá como entrada um programa escrito em PEWS e, após as etapas de análise léxica e sintática, irá gerar uma representação em XML do programa PEWS (XPEWS). Esta representação é de fato uma árvore sintática abstrata que representa os construtores da linguagem.

A etapa de geração de código é realizada pelo módulo `Compositor`, descrita na Figura 5.1. Para implementar o módulo, a linguagem Java foi escolhida, por ser multiplataforma, possuir uma rica documentação e pela grande diversidade de ferramentas disponibilizadas pela comunidade para a implementação de serviços Web, desde bibliotecas para manipulação de documentos XML/WSDL até a implementação de um provedor de serviços Web, o qual foi descrito na seção 2.2.4.

Uma técnica utilizada para gerar código é a `tradução dirigida por sintaxe`, onde ações semânticas são associadas às regras de produção da gramática de modo que, quando uma dada produção é processada (por derivação ou redução de uma forma sentencial

no processo de reconhecimento), essas ações são executadas [2, 56]. As ações podem gerar ou interpretar código, armazenar informações em uma tabela de símbolos, emitir mensagens de erro, etc. Para tornar as ações semânticas mais efetivas, podem-se associar variáveis aos símbolos (terminais e não-terminais) da gramática. Assim, os símbolos gramaticais passam a conter atributos (ou parâmetros) capazes de armazenar valores durante o processo de reconhecimento. Toda vez que uma regra de produção é usada no processo de reconhecimento de uma sentença, os símbolos gramaticais dessa regra são "alocados" juntamente com seus atributos. Pensando na árvore de derivação da sentença sob análise, é como se a cada nó da árvore (símbolo gramatical) correspondesse uma instanciação de um símbolo de suas variáveis. É como se o nó contivesse campos para armazenar os valores (atributos) correspondentes ao símbolo.

No esquema de tradução usado, os símbolos terminais possuem apenas atributos sintetizados, sendo que a tradução será feita através da implementação de um analisador redutivo (*bottom-up*). Para avaliar os atributos sintetizados, usou-se o seguinte algoritmo, ilustrado pela Figura 5.2.

```
visitaNodo(n:nodo)
Para cada filho m de n, da esquerda para direita, faça:
    visitaNodo(m)
Calcule os atributos sintetizados de n
```

Figura 5.2: Algoritmo para avaliação de atributos sintetizados

Exemplo 5.2.1 Um esquema de tradução dirigida pela sintaxe, usada para avaliar as *path expressions* de um programa PEWS, é mostrada pela Figura 5.3. Os símbolos não-terminais S e operation, e o terminal id, possuem dois atributos sintetizados do tipo texto: *nome* e *cod*, enquanto que o símbolo path possui apenas o atributo sintetizado *cod*. O atributo *cod* de S contém o código final gerado para uma *path expression*.

Produção	Regras Semânticas
S → id path	S.*nome* := id.*nome* S.*cod* := "class " + id.*nome* + " implements CSProcess { ..... path.*cod* ..... }"
path → path | path	path.*cod* := "new Choice(" + path1.*cod* + "," + path2.*cod* + ")"
path → path . path	path.*cod* := "new Seq(" + path1.*cod* + "," + path2.*cod* ")"
path → operation	path.*cod* := "new " + operation.*nome* + "()"
operation → id	operation.*nome* := id.*nome* "class " + id.*nome* + " implements CSProcess { ..... canal ..... }"

Figura 5.3: Tradução dirigida pela sintaxe para avaliação das *path expressions*

O símbolo +, que aparece no lado direito das regras, é usado para concatenar texto (código). O token *id* possui um atributo sintetizado *nome*, cujo valor é fornecido pelo analisador léxico. □

A Figura 5.4 mostra a árvore sintática de derivação, com os atributos associados aos nodos, para o seguinte programa PEWS:

$$\text{Depósito} = \text{fazerPedido}^{\ominus\oplus} \ .\ \text{emitirRecibo}^{\oplus\ominus} + \text{abortarOperação}^{\oplus}$$

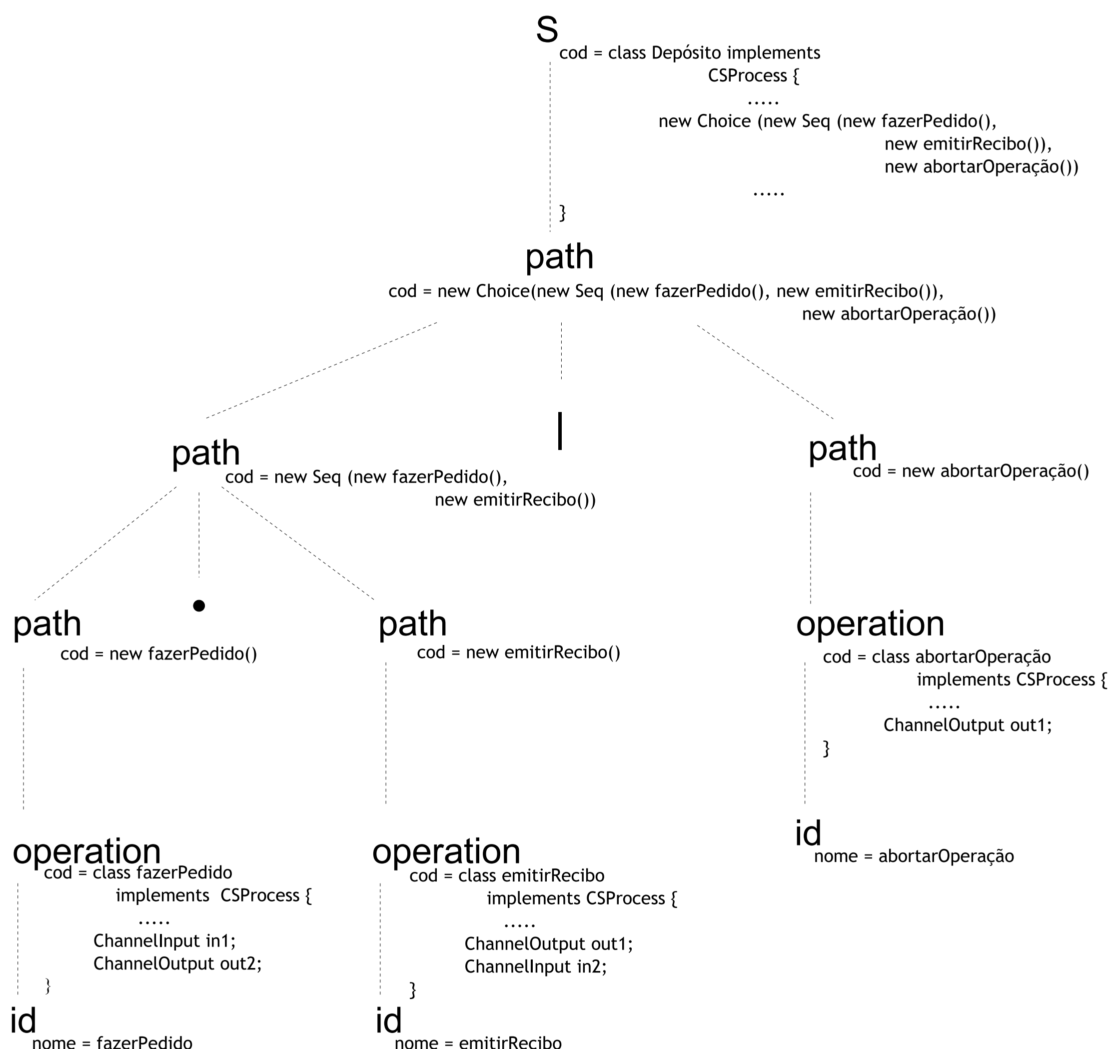


Figura 5.4: Árvore sintática abstrata

Por uma questão de legibilidade, a árvore sintática ilustra apenas parte do código gerado, sendo que os detalhes serão discutidos no decorrer da seção. Para mostrar como os atributos são calculados, considere a regra de produção operation → id. Percorrendo a árvore de maneira *bottom-up*, da esquerda para a direita, a regra semântica op.*nome* := id.*nome* define que o atributo *nome* de operation naquele nodo possui valor fazerPedido, porque o valor de id.*nome* do filho neste nodo possui o valor fazerPedido. Para o atributo *cod* desta mesma produção a tradução foi definida da seguinte maneira:

Produção	Regras Semânticas
operation → id	operation.*nome* := id.*nome*
	"class " + id.*nome* + " implements CSProcess { ..... canal ..... }"

Para essa regra de produção, o atributo *cod* de operation receberá como valor uma instrução de código que implementa uma classe em JCSP. O nome da classe é dado pelo atributo sintetizado id.*nome*. Como em JCSP uma classe é um processo, a comunicação deste processo com os demais será através do uso dos canais de comunicação. No exemplo ilustrado pela Figura 5.4, o código detalhado gerado pelo *back-end* para a operação fazerPedido é apresentada na Figura 5.5.

```
1  class fazerPedido implements CSProcess {
2   private ChannelInput in1; // canal de entrada
3   private ChannelOutput out2; // canal de saída
4
5   private static int reqVal = 0; // contador de requisição (req.val)
6   private static int reqTime = 0; // contador de requisição (req.time)
7   private static int actVal = 0; // contador de ativação (req.val)
8   private static int actTime = 0; // contador de ativação (req.time)
9   private static int termVal = 0; // contador de término (req.val)
10  private static int termTime = 0; // contador de término (req.time)
11
12  public fazerPedido(ChannelInput in1, ChannelOutput out2) {
13   this.in1 = in1;
14   this.out2 = out2;
15   this.reqVal++;
16   this.reqTime = System.currentTimeMillis();
17  }
18
19  public void run() {
20              .....
21   this.actVal++;
22   this.actTime = System.currentTimeMillis ();
23              .....
24   // chamada da operação
25              .....
26   this.termVal();
27   this.termTime = System.currentTimeMillis();
28  }
29 }
```

Figura 5.5: Código gerado pelo *back-end* para uma operação.

Pelo código ilustrado na Figura 5.5, pode-se verificar que a implementação da operação fazerPedido será um processo JCSP. As linhas 2 e 3 mostram a definição dos canais de comunicação, que serão usados na comunicação e sincronização entre os processos. Pelo exemplo, a operação fazerPedido possui dois canais: um canal de entrada e outro de saída. Pelo canal de entrada chegará uma mensagem e, em seguida, a operação fazerPedido será executada; pelo canal de saída será enviada uma resposta gerada em decorrência da execução dessa operação. Paralelamente, existirá um outro processo responsável por enviar essa mensagem, o qual implementa os canais de maneira inversa (saída/entrada). Sendo assim, pelo exemplo da coreografia ilustrado pela Figura 3.3, a operação fazerPedido, do tipo `entrada/saída`, é implementada pelo serviço *Depósito*, enquanto que a mesma operação do tipo `saída/entrada` é implementada pelo serviço *Loja*.

O uso da biblioteca JCSP garante a troca de mensagens entre os processos, mas ela não é responsável pela execução da operação. Para executar uma operação, um processo envia o dado necessário para que outro processo possa executar a operação. Esta idéia foi vista em detalhes na Figura 4.2. Visto que as operações a serem executadas estão localizadas em diferentes provedores de serviços, é preciso que esta chamada esteja de acordo com os padrões de serviços Web vistos nos capítulos anteriores, em particular codificar estas chamadas via SOAP. Um provedor de serviços que possibilita que as operações de seus serviços possam ser executadas via SOAP é o Apache Axis, o qual foi o escolhido como provedor de serviços desta implementação. Dessa forma, para codificar a chamada da operação, o Axis disponibiliza uma API que habilita que programas clientes implementem as chamadas corretamente, enviando-as via SOAP para serem processadas pelo provedor Axis e, se houver uma resposta, codifica e envia a resposta via SOAP ao cliente.

Foi visto na seção 3.1.3.2, na especificação da sintaxe de PEWS, que as operações definem contadores para serem usados juntamente com os predicados. Cada contador possui dois componentes: *val* e *time*. O componente *val* é o próprio contador, definido por um número inteiro, enquanto que o componente *time* define o momento (tempo) em que o contador foi modificado. Na codificação da operação fazerPedido, ilustrada pela Figura 5.5, as linhas 5 a 10 mostram a definição dos contadores de PEWS. Quando o

objeto é instanciado pela primeira vez, todos os contadores são inicializados em zero. Essa classe codifica os seguintes contadores:

req(O): esse contador (de requisição) está implementado como um atributo da classe, formado pelos atributos `reqVal` e `reqTime`, que correspondem aos componentes *val* e *time*, respectivamente. A atualização dos contadores é realizada pelo construtor da classe, mostrado pelas linhas 15 e 16, pois supõe-se que nesse momento é feita a requisição da operação.

act(O): esse contador (de ativação) está implementado como um atributo da classe, formado pelos atributos `actVal` e `actTime`, que correspondem aos componentes *val* e *time*, respectivamente. As linhas 21 e 22 mostram como acontece a atualização dos contadores durante a execução da operação fazerPedido. Note que a atualização é feita antes da execução da operação.

term(O): esse contador (de término) também está implementado como um atributo da classe, e é formado pelos atributos `termVal` e `termTime`, que correspondem aos componentes *val* e *time*, respectivamente. As linhas 26 e 27 mostram a atualização dos contadores durante a execução da operação fazerPedido. A atualização dos contadores é feita logo após o fim da execução da operação.

Para o nodo cuja produção é path → operation, o valor do atributo path.*cod* é definido por:

Produção	Regra Semântica
path → operation	path.*cod* := "new " + operation.*nome* + "()"

O resultado de aplicar a regra semântica neste nodo é gerar uma instrução que é a instância de uma classe, cujo código foi definido anteriormente pela regra de produção operation → id. Na árvore de derivação, mostrada pela Figura 5.4, têm-se a codificação de duas intâncias: a instância da operação fazerPedido e a instância da operação emitirRecibo. O atributo *cod* irá receber a instrução de código gerada, que será utilizado na composição do código do pai desse nodo. Considere agora a regra de produção path → path . path. O valor do atributo path.*cod* é definido por:

Produção	Regra Semântica
path → path . path	path.*cod* := "new Seq(" + path1.*cod* + "," + path2.*cod* + ")"

A regra semântica definida para esse nodo irá gerar uma instrução JCSP que define a execução sequencial de seus argumentos, dados por path1.*cod* e path2.*cod*. Na Figura 5.4, verifica-se que primeiro será executada a operação fazerPedido e em seguida a operação emitirRecibo. De maneira análoga é feita a geração de código para a produção path → path | path, a qual determina que apenas uma das operações serão executadas de maneira não-determinística, isto é, ou a seqüência de operações fazerPedido e emitirRecibo, ou a operação abortarOperação. Por fim, para a regra de produção S → path, o valor do atributo S.*cod* é definido por:

Produção	Regras Semânticas
S → id path	S.*nome* := id.*nome* S.*cod* := "class " + id.*nome* + " implements CSProcess { ..... path.*cod* ..... }"

O atributo *nome* de S é calculado a partir do atributo *nome* de id, que é retornado pelo analisador léxico, o qual corresponde ao nome do serviço associado ao programa PEWS que analisado no momento. Os detalhes do código gerado pelo atributo S.*cod*, são mostrados na Fgura 5.6.

A Figura 5.6 mostra o código gerado pelo *back-end* para o serviço *Depósito*. As linhas 2 a 5 mostram a declaração dos canais de comunicação usados pelas operações contidas no serviço. O método `run()`, mostrado pela linha 14, é quem determina o comportamento da classe. Nesse caso, o comportamento da classe será executar as operações na ordem estabelecida pelos construtores JCSP, mostrados na linha 18. No próximo capítulo será visto um exemplo que mostrará um cenário onde a linguagem PEWS foi usada para implementar a composição de serviços Web.

```
1  class Depósito implements CSProcess {
2   private ChannelInput in1;
3   private ChannelInput in2;
4   private ChannelOutput out1;
5   private ChannelOutput out2;
6
7   public Depósito(ChannelInput in1, ChannelInput in2, ChannelOutput
       out1, ChannelOutput out2) {
8    this.in1 = in1;
9    this.in2 = in2;
10   this.out1 = out1;
11   this.out2 = out2;
12  }
13
14  public void run() {
15   fazerPedido = new fazerPedido(in1, out2);
16   emitirRecibo = new emitirRecibo(out1, in2);
17       .....
18   new Choice(new Sequence(fazerPedido, emitirRecibo), new
       abortarOperação());
19  }
20 }
```

Figura 5.6: Código gerado pelo *back-end* para um serviço.

CAPÍTULO 6

ESTUDO DE CASO

Este capítulo apresenta um estudo de caso cujo objetivo é validar a implementação proposta no Capítulo 5, onde um cenário de composição de serviços Web será utilizado para este fim. A Seção 6.1 irá descrever a arquitetura utilizada para o estudo de caso, a qual é uma arquitetura específica do *back-end* proposto para a implementação, mostrada na Seção 5.1. Nesta arquitetura específica, os componentes foram escolhidos de acordo com o seguinte critério: (i) implementados em Java, por ser uma linguagem multiplataforma; (ii) implementação em código aberto; (iii) implementação com boa documentação e em constante atualização.

A seção 6.2 irá descrever o cenário de composição de serviços Web e os passos necessários para codificá-lo usando-se a arquitetura proposta para o estudo de caso. Para o cenário proposto, dois enfoques poderiam ser usados na solução da composição: a *coreografia* ou a *orquestração*. A linguagem PEWS fornece suporte para ambos os enfoques, porém o exemplo mostrado será o do caso mais geral: a coreografia.

6.1 Arquitetura para o Estudo de Caso

Na Seção 5.1, mostrou-se uma arquitetura geral proposta para implementar o *back-end* da linguagem PEWS. Nesta seção, será mostrada uma arquitetura equivalente específica, onde os componentes genéricos serão substituídos por implementações específicas que atendam ao objetivo proposto. A Figura 6.1 ilustra a arquitetura utilizada para ilustrar o Estudo de Caso. Para desenvolver o estudo de caso utilizou-se as seguintes tecnologias:

- Linguagem Java (SDK) 1.5
- Servidor Web: Apache Tomcat versão 1.4.x
- Apache Axis versão 1.1 - Servidor de Serviços Web como um plug-in do Tomcat

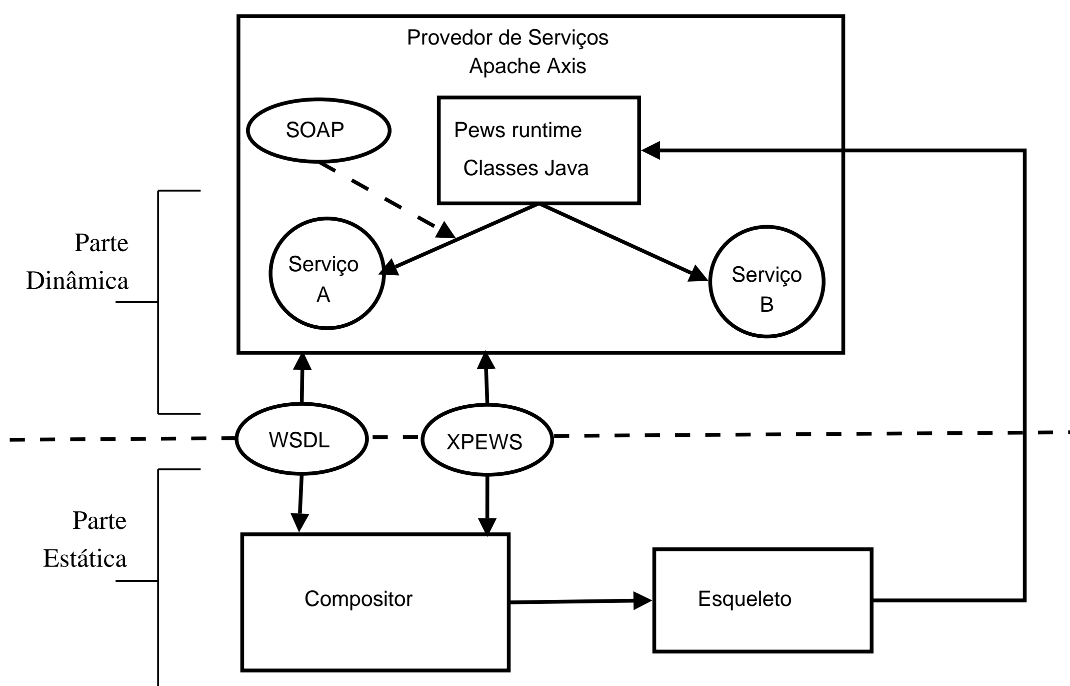


Figura 6.1: Arquitetura do *back-end* para o Estudo de Caso

- Biblioteca JCSP versão 1.0

A Figura 6.1 mostra uma visão global do modelo da arquitetura proposta para realizar a composição em PEWS, sendo esta dividida em duas camadas: uma camada `estática` e uma camada `dinâmica`. A camada estática do *back-end* é formada pelo módulo Compositor, que é o responsável pela etapa de geração de código da linguagem PEWS. O módulo Compositor lê uma descrição da interface e gera um esqueleto de classes para o serviço. A descrição da interface é formada por documentos nos formatos WSDL e XPEWS. Os documentos WSDL irão descrever as interfaces dos serviços Web utilizados pela composição, enquanto que os documentos XPEWS irão coordenar a interação entre os serviços.

Através das informações contidas nesses documentos, o módulo `Compositor` irá gerar um esqueleto de classes em Java, que é uma implementação intermediária do sistema de tempo de execução. O esqueleto de classes gerado pelo módulo `Compositor` utiliza a biblioteca JCSP para implementar as primitivas de comunicação e sincronização, isto é, as *path expressions*. O esqueleto de classes é a base para a implementação do serviço, pois ainda é necessário que o programador adicione código ao esqueleto, por exemplo, geração dos dados que serão usados nas trocas de mensagens.

A parte dinâmica do *back-end* é o sistema de tempo de execução da linguagem. Ela é composta por um conjunto de classes Java (*PEWS runtime*), que é uma extensão do esqueleto de classes, executada pelo programador, das classes Java que implementam o serviço. O programa é então registrado como um serviço em um provedor de serviços, sendo que a comunicação com os clientes e outros serviços será através do protocolo SOAP. Na Figura 5.1, o provedor de serviços utilizado é o Apache Axis, o qual foi descrito na Seção 2.2.4.

6.2 Descrição do cenário

Para ilustrar o estudo de caso, utilizou-se o exemplo visto anteriormente para exemplificar a composição pelas abordagens da coreografia e da orquestração, ilustrado pela Figura 6.2. O cenário, adaptado de [34], mostra uma visão geral de todos os serviços envolvidos na composição, bem como as sincronizações entre eles. O objetivo desta composição é aprovar (ou não) um pedido de compra feita pelos clientes de uma loja. A Figura 6.2 mostra os quatro serviços envolvidos na composição (*Cliente*, *Loja*, *Depósito* e *Banco*), bem como as operações, tais como enviarConta, autorizar e emitirRecibo.

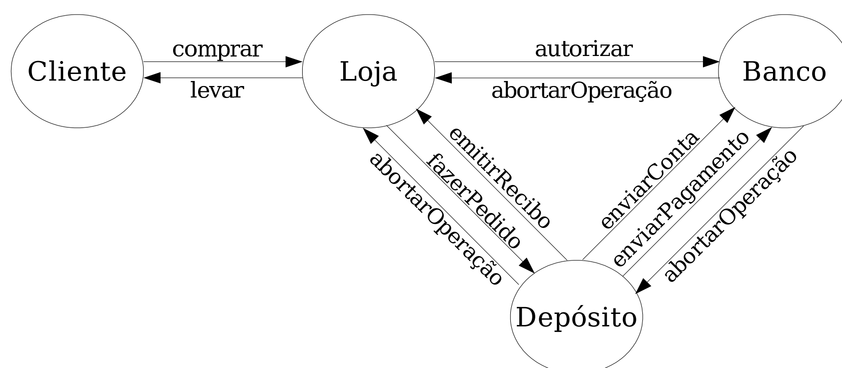


Figura 6.2: Composição de serviços Web utilizada no estudo de caso

Agora será descrito o comportamento descrito para este cenário. Nesta composição, uma loja recebe requisições de pedidos de compra de seus clientes e, se o pedido for aprovado, o cliente pode levar os produtos requisitados. O cliente mantém um banco de dados, em formato xml, que contém informações utilizadas para gerar o pedido de compra.

Estas informações contém o código do cliente, o código do produto, a quantidade do produto e o total do pedido. Ao receber os pedidos de compra, a loja pede autorização ao banco, para que verifique se o cliente possui saldo suficiente para realizar a compra. Para manipulação de dados, o banco mantém um banco de dados xml que contém informações sobre seus clientes. Estas informações contém o código do cliente e o respectivo saldo. O depósito recebe os pedidos de compra da loja e envia a conta e o pedido de pagamento para o banco. Se o pagamento for realizado dentro de 48 horas, o depósito pode emitir o recibo de compra para a loja. Caso contrário toda a composição é encerrada.

Com a definição do cenário utilizado para o estudo de caso, o passo seguinte será codificar as funcionalidades de cada serviço Web separadamente, em Java, e disponibilizá-lo como um serviço Web, através do provedor de serviços Apache Axis. O Axis considera uma classe Java como sendo um serviço Web e seus métodos como sendo operações. As classes e suas operações descrevem a lógica de negócio, no entanto cada classe é responsável por executar uma tarefa específica, determinada por sua operação, e uma classe não comunica-se diretamente com a outra. A composição, por sua vez, será responsável pela troca de mensagens (comunicação) entre as classes, determinando a ordem de execução das operações (métodos).

Realizada a codificação, geram-se os respectivos documentos WSDL que descrevem as interfaces dos serviços. Estes documentos são necessários para a geração de código do PEWS *runtime*, uma vez que a codificação das chamadas das operações dos serviços serão realizadas através das descrições das interfaces, e não diretamente pela instanciação de suas classes. Os documentos WSDL que descrevem cada serviço podem ser vistos no Apêndice A.2.

Em seguida, é preciso codificar os programas PEWS que descrevem o comportamento de cada serviço. Estes programas descrevem a composição em um nível abstrato, os quais foram vistos na Seção 3.1.3.5, como segue:

$\mathsf{Cliente} = \mathsf{comprar}^{\oplus\ominus} \,.\, \mathsf{levar}^{\ominus}$

$\mathsf{Loja} = \mathsf{comprar}^{\ominus\oplus}.\mathsf{autorizar}^{\oplus\ominus}.\mathsf{fazerPedido}^{\oplus\ominus}.\ \mathsf{emitirRecibo}^{\ominus\oplus}.(\mathsf{levar}^{\oplus}+\mathsf{abortarOperação}^{\ominus})$

$\mathsf{def}\ t^{pgto} = \mathsf{now() - term(enviarConta)}.time$

$$\mathsf{Depósito} = \mathsf{fazerPedido}^{\ominus\oplus}.\mathsf{enviarConta}^{\oplus\ominus}.\\ ([t^{pgto} \leq 48h]\mathsf{enviarPagamento}^{\oplus\ominus}.\mathsf{emitirRecibo}^{\oplus\ominus}\\ +[t^{pgto} > 48h]\ \mathsf{abortarOperação}^{\oplus})$$

$$\mathsf{def}\ t^{pgto} = \mathsf{now() - term(enviarConta)}.time$$

$$\mathsf{Banco} = \mathsf{autorizar}^{\ominus\oplus}.\mathsf{enviarConta}^{\ominus\oplus}.\\ ([t^{pgto} \leq 48h]\ \mathsf{enviarPagamento}^{\ominus\oplus}+[t^{pgto} > 48h]\ \mathsf{abortarOperação}^{\oplus})$$

Para cada programa PEWS, o *front-end* irá gerar uma versão XPEWS correspondente, que será utilizada pelo *back-end* para gerar o código que implementa o sistema de tempo de execução da linguagem. Um exemplo de documento XPEWS, para o serviço *Banco*, é ilustrado pela Figura 6.3. Os demais programas XPEWS podem ser vistos no Apêndice A.3.

```
1  <envelope xmlns="http://aquarius.inf.ufpr.br" xmlns:xsi="http://www.
      w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
      aquarius.inf.ufpr.br pews.xsd">
2
3  <behaviour name="Banco" xmlns:banco="http://localhost:8080/axis/
      samples/loja/banco.wsdl">
4   <operations>
5    <operation name="autorizar" portType="banco" refersTo="
        banco:autorizar"/>
6    <operation name="enviarConta" portType="banco" refersTo="
        banco:enviarConta"/>
7    <operation name="emitirRecibo" portType="banco" refersTo="
        banco:emitirRecibo"/>
8    <operation name="enviarPagamento" portType="banco" refersTo="
        banco:enviarPagamento"/>
9   </operations>
10  <varDef name="tpgto">
11   <minus>
12    <libFunction name="now" unit="hours"/>
13    <pewsCounter opname="enviarConta" name="term" component="time"
         unit="hours"/>
14   </minus>
15  </varDef>
16  <pathExp>
17  <seq>
18   <operation name="autorizar"></operation>
19   <operation name="enviarConta"></operation>
20    <choice>
21     <pred>
22      <leq>
23       <var name="tpgto"/>
24       <const value="48"/>
25      </leq>
26       <operation name="enviarPagamento"></operation>
27      </pred>
28     <pred>
29     <gt>
30      <var name="tpgto"/>
31      <const value="48"/>
32     </gt>
33     <operation name="abortarOperação"></operation>
34    </pred>
35   </choice>
36   </seq>
37  </pathExp>
38 </behaviour>
39 </envelope>
```

Figura 6.3: Programa XPEWS referenta à tradução do serviço Banco.

Na linha 1, a tag `<envelope>` é utilizada para declarar alguns *namespaces*. Por exemplo, o *namespace* `xmlns:xsi` especifica que o documento XPEWS está associado a um esquema de validação, em particular, XML Schema. Na linha 3, a tag `behaviour` é usada para descrever os *namespaces* dos serviços usados pelo documento XPEWS. Neste caso, o documento XPEWS irá usar apenas operações do serviço *Banco*, descrito pelo *namespace* `xmlns:banco`, o qual indica a localização do documento WSDL que descreve as operações. Na linha 4, a tag `operations` agrupa as operações usadas na composição. Esta tag possui um atributo chamado *refersTo*, que associa o *namespace* utilizado pela operação, declarado pela tag `behaviour`. A linha 10 mostra a definição de uma variável, cujo valor é dado pela função now() (hora atual). Na linha 16, a tag `pathExpr` contém as tags que descrevem a ordem de execução das operações, isto é, as *path expressions*.

Após a codificação dos serviços Web, das descrições das interfaces e dos programas XPEWS, o *back-end* inicia a etapa de geração de código. O módulo `Compositor` irá usar os documentos XPEWS e WSDL para gerar o código intermediário (esqueleto) que implementa o serviço. Um documento XPEWS é uma árvore sintática abstrata que representa os construtores da linguagem PEWS e, por conseguinte, define a ordem de execução das operações, as quais são definidas pelo padrão WSDL. Primeiramente, os documentos XPEWS são processados juntamente com os respectivos documentos WSDL. O resultado desse processamento é gerar um esqueleto de classes Java, através da tradução dirigida pela sintaxe, de cada árvore sintática representada pelos programas XPEWS. Para cada programa XPEWS, será criado um processo JCSP para cada operação e um processo para cada serviço. A comunicação e a sincronização entre processos será feita por troca de mensagens através dos canais de comunicação, como ilustra a Figura 6.4.

No passo seguinte, o programador deverá adicionar código ao esqueleto de classes, a fim de gerar o sistema de tempo de execução (Pews *runtime*). Para este cenário, foi preciso adicionar o conteúdo das mensagens a serem enviadas pelos canais de comunicação. Por exemplo, para a operação autorizar do serviço *Loja*, a Figura 6.5 ilustra um documento WSDL que especifica qual o conteúdo da mensagem a ser enviada.

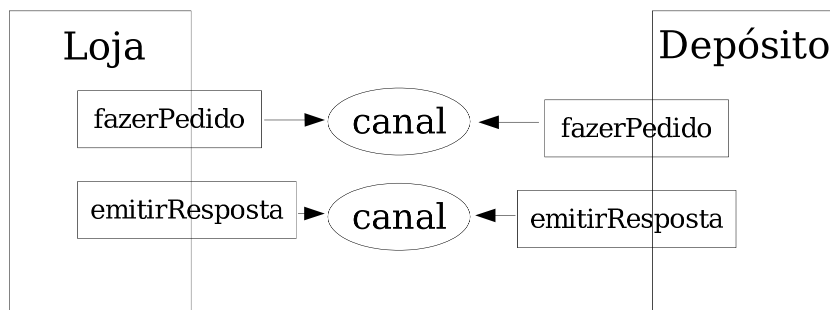


Figura 6.4: Comunicação entre processos

```
1  <wsdl:message name="pedidoLojaOut">
2     <wsdl:part name="pedidoLojaOut" type="xsd:int"/>
3  </wsdl:message>
4  <wsdl:message name="pedidoLojaIn">
5     <wsdl:part name="pedidoLojaIn" type="xsd:int"/>
6  </wsdl:message>
7  <wsdl:portType name="Loja">
8     <wsdl:operation name="fazerPedido">
9        <wsdl:output message="impl:pedidoLojaOut" name="pedidoLojaOut"/
            >
10       <wsdl:input message="impl:pedidoLojaIn" name="pedidoLojaIn"/>
11    </wsdl:operation>
12 </wsdl:portType>
```

Figura 6.5: Documento WSDL referente ao serviço Loja

Na linha 2, pode-se verificar que a mensagem a ser enviada possui dois argumentos: um do tipo inteiro (`xsd:int`) e outro do tipo real (`xsd:real`). Portanto, o programador deverá gerar esses dados, neste caso, o código do cliente e o total do pedido, para que o banco possa receber o dado e verificar se o cliente está autorizado para realizar a compra. Por fim, o Pews *runtime* será registrado no servidor Axis, ficando disponível como um novo serviço. Os códigos fontes utilizados para implementar o estudo de caso, bem como instruções de instalação e de execução estão disponíveis no site: http://www.est.ufpr.br/˜marcos/pews.

Tomando-se como base o esqueleto de classes, fazendo uma análise do número de linhas de código geradas pelo compilador, têm-se um total de 515 linhas, com um total de 15 classes geradas. Por outro lado, após a adição de código ao esqueleto de classes, têm-se um total 620 linhas, uma média de 7 linhas alteradas para cada classe gerada. Além destas alterações, criou-se também uma nova classe que contém o arquivo principal, responsável por criar as chamadas das classes geradas. Verificando-se o conteúdo gerado

pelo esqueleto de classes, percebe-se que as classes possuem um padrão de código bem definido, em grande parte refletido pelo uso da biblioteca JCSP e da API do Axis, fazendo com que o usuário que já tenha alguma experiência com programação consiga alterar o código para outras situações.

Ao adicionar código ao esqueleto de classes Java, alguns problemas podem ocorrer. Um possível problema pode ser criado no momento em que forem gerados os dados a serem enviados pelos canais de comunicação. Pode-se enviar dados incompatíveis com os esperados pelos serviços Web, fazendo com que sua execução ocasione uma exceção. Por isso, o Axis possibilita tratar exceções dos serviços Web codificados em Java, fazendo com que o retorno SOAP desta chamada contenha o conteúdo gerado pela exceção. Outro problema mais difícil de detectar é quando são inseridos novos canais de comunicação ao esqueleto de classes. Se um novo canal ligando dois processos for definido, é preciso verificar se está corretamente codificado a leitura e escrita pelos processos correspondentes.

Durante a implementação do estudo de caso, verificou-se algumas restrições apresentadas por esta versão do *back-end* da linguagem. Não foi implementado um mecanismo de transação, ou seja, se durante a execução da composição houver um erro, as operações que foram executadas não serão desfeitas. Outra restrição deve-se ao uso do provedor de serviços Axis. Na versão utilizada, o Axis não suporta que o retorno a uma chamada de operação via SOAP receba ou retorne todos os tipos de dados suportados pela linguagem Java. Isto significa que os serviços Web implementados em Java devem seguir as restrições impostas pela versão utilizada. Por fim, é preciso rever a codificação da escolha não-determinística que contém predicados. A gramática de atributos e as ações correspondentes funcionam de maneira correta no caso da escolha não-determinística contendo predicados binários. Nos casos onde houverem predicados complexos, é preciso criar uma generalização das ações que para o tratamento correto destes casos.

CAPÍTULO 7

CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho apresentou uma implementação para o sistema de tempo de execução da linguagem PEWS, seguindo a especificação dada no artigo estendido de PEWS [9]. Através do uso das descrições das interfaces das operações, descritas em WSDL, e da ordenação da execução das operações, descritas em XPEWS, a arquitetura do *back-end* gera um esqueleto de classes Java que implementam o serviço. A fim de gerar o sistema de tempo de execução (PEWS *runtime*), o programador deverá adicionar código de modo a complementar o serviço descrito pelo esqueleto de classes. Este código final é então registrado em um provedor de serviços Web, de modo que o PEWS *runtime* fique disponível para execução como um novo serviço Web.

No modelo utilizado para a comunicação entre os serviços, ilustrado pela Figura 4.2, foi visto que os serviços são tratados como processos, executam-se concorrentemente e comunicam-se através de troca de mensagens. Visto que o *back-end* foi codificado em Java, uma implementação Java da CSP (JCSP), cujos construtores são semanticamente equivalentes aos de PEWS, foi utilizada para implementar os construtores de PEWS, a comunicação e a sincronização entre processos. O uso dos canais de comunicação implementados pela biblioteca JCSP, mostra uma maneira natural de controle de concorrência usando-se *threads*. Dessa maneira, evitam-se os problemas encontrados na programação de *threads* em Java [59].

O comportamento dos serviços Web pode também ser definido por padrões de *workflow*, os quais definem a gerência de fluxo de execução de processos. Uma comparação entre a linguagem PEWS e as linguagens mais populares para descrição de serviços são apresentadas em [45]. Esse trabalho desenvolveu um estudo sobre as funcionalidades de PEWS, através do uso de 20 padrões de processos de *workflow*, apresentados em [66]. Foi verificado que 19 dos 20 padrões de *workflow* descritos podem ser implementados em

PEWS. Pode-se verificar que, comparando-se o estudo de padrões *workflow* de PEWS, com o estudo de padrões de *workflow* de outras linguagens, apresentadas em [74], PEWS é linguagem que expressa mais padrões de *workflow*. Isto é uma indicação de que as *path expressions* descrevem de maneira adequada a composição de serviços Web.

Em uma comparação de XPEWS com BPEL, por exemplo, é possível verificar que, para certos padrões, elas são muito semelhantes. Porém, XPEWS é desenvolvida a partir de programas PEWS, que descrevem o comportamento dos serviços Web em nível abstrato mais expressivo do que BPEL, que é puramente descrito em XML. Por exemplo, a Figura 7.1 mostra um diagrama de *workflow*, onde a execução da operação D será inicializada somente depois que A for concluída (isto é indicado pela flecha pontilhada). O código BPEL para esta situação também é mostrado nessa figura. Um programa PEWS e sua versão XPEWS equivalente ao programa mostrado pela Figura 7.1 é dado a seguir:

```
<pathExp>
 <par>
  <seq>
   <operation name="A"/>
   <operation name="B"/>
  </seq>
  <seq>
   <operation name="C"/>
    <pred>
     <gt>
      <pewsCounter component="val"
             name="term"
             opname="B"
             unit="miliseconds"/>
      <pewsCounter component="val"
             name="term"
             opname="D"
             unit="miliseconds"/>
     </gt>
      <operation name="D"/>
     </pred>
  </seq>
 </par>
</pathExp>
```

```
( A . B ) || ( C . [term(B).val > term(D).val]) D
```

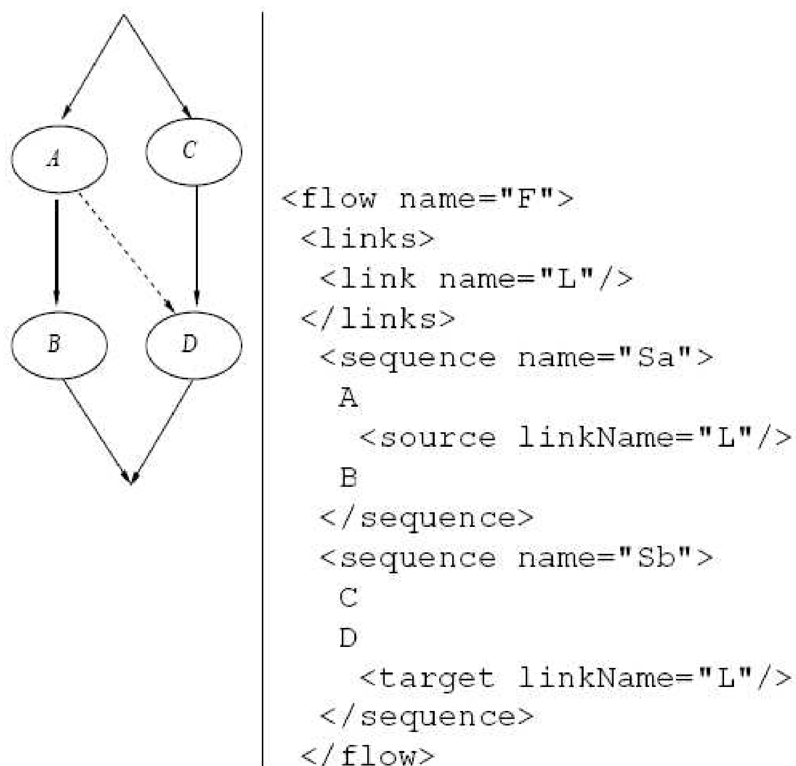


Figura 7.1: *Workflow* não-estruturado

A maior dificuldade encontrada no desenvolvimento do *back-end* foi modelar a comunicação entre processos, visto que o padrão WSDL define apenas a interface do serviço. A partir do modelo proposto pela Figura 4.2, foi possível utilizar os conceitos de programação concorrente para construir o sistema de tempo de execução da linguagem. Um problema encontrado foi com relação à codificação das chamadas de execução das operações. A melhor maneira para isto seria utilizar a ferramenta `WSDL2Java`, a fim de gerar automaticamente as chamadas de execução das operações dos serviços, através da leitura da interface WSDL dos serviços. Porém, na prática, viu-se que este código gerado era incompleto, pois era necessário adicionar código para complementar as chamadas. Visto que o esqueleto de classes também necessita de adição de código, optou-se por usar diretamente a API do Axis para gerar as chamadas de execução das operações, a fim de minimizar a intervenção do programador na geração do código final.

Como objeto de pesquisa, este trabalho proporcionou um estudo interdisciplinar, pois envolveu a pesquisa bibliográfica das seguinte áreas:

Banco de Dados: Estudo da linguagem XML e de seus esquemas;

Programação Concorrente: primitivas de comunicação e sincronização entre processos;

Linguagens de Programação: JCSP, CSP, BPEL entre outras

Como trabalhos futuros, identificou-se algumas melhorias nas funcionalidades do *back-end*, as quais são ressaltadas a seguir:

- Integração do *front-end* com o *back-end.* No estágio atual, o desenvolvimento foi destes dois trabalhos foram feitos paralelamente, porém eles não estão integrados. Numa versão que possa ser utilizada na prática, é preciso integrar o ambiente como um todo.

- Executar mais testes com o *back-end* de PEWS em diferentes cenários de composição, a fim de validar suas funcionalidades. Pode-se comparar as funcionalidades, por exemplo, com o *back-end* de BPEL, denominado *BPELWS4J.*

- Criar documentação, página Web e disponibilizar este projeto em um ambiente colaborativo, onde o trabalho possa ser divulgado e utilizado pela comunidade, por exemplo, no *sourceforge.*

BIBLIOGRAFIA

[1] Manish Agarwal. Synthesizing autonomic compositions in grid environment. Dissertação de Mestrado, Rutgers, The State University of New Jersey, 2003.

[2] A. V. Aho, R. Sethi, e J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1988.

[3] Selim Aissi, Pallavi Malu, e Krishnamurthy Srinivasan. E-business process modeling: The next big step. *Computer*, 35(5):55–62, 2002.

[4] Sten Andler. Predicate path expressions. *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, páginas 226–236. ACM Press, 1979.

[5] Gregory R. Andrews e Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Comput. Surv.*, 15(1):3–43, 1983.

[6] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, e S. Weerawarana. Specification: Business process execution language for web services version 1.1. 2003. http://www-128.ibm.com/developerworks/library/specification/ws-bpel/, 2003.

[7] Assaf Arkin, Sid Askary, Scott Fordin, Wolfgang Jekeli, Kohsuke Kawaguchi, David Orchard, Stefano Pogliani, Karsten Riemer, Susan Struble, Pal Takacsi-Nagy, Ivana Trickovic, e Sinisa Zimek. Web service choreography interface. http://www.w3.org/TR/wsci/, 2002.

[8] Daniel Austin, Abbie Barbir, Ed Peters, e Steve Ross-Talbot. Web services choreography requirements. http://www.w3.org/TR/2004/WD-ws-chor-reqs-20040311/, março de 2004. W3C Working Draft.

[9] C. Ba, M. Carrero, M. H. Ferrari, e M. Musicante. Pews: A new language for building web service interfaces. *Journal of Universal Computer Science (JUCS)*, 11(7):1215–1233, 2005.

[10] Cheikh Ba, Mírian Halfeld Ferrari, e Martin Alejandro Musicante. Building web service interfaces using predicate path expressions. *Proceedings of the 9th Brazilian Symposium on Programming Languages*, 2005.

[11] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, e M. Mecella. Automatic composition of e-services. Relatório Técnico 22-2003, Dipartimento di Informatica e Sistemistica, Universita di Roma La Sapienza, Roma, Itália, 2003.

[12] David Bodoff, Patrick C.K. Hung, e Mordechai Ben-Menachem. Web metadata standards: Observations and prescriptions. *Software*, 22(1):78–85, 2005.

[13] Antonio Brogi, Carlos Canal, Ernesto Pimentel, e Antonio Vallecillo. Formalizing web service choreographies. *Electronic Notes in Theoretical Computer Science*, 105:73–94, dezembro de 2004.

[14] James Clark e Makoto Murata. Relax ng specification. http://www.relaxng.org/spec-20011203.html, dezembro de 2001. Committee Specification.

[15] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, e S. Weerawarana. Unraveling the web services web: An introduction to soap, wsdl, and uddi. *IEEE Internet Computing*, 6(2):86–93, 2002.

[16] Francisco Curbera, Rania Khalaf, Nirmal Mukhi, Stefan Tai, e Sanjiva Weerawarana. The next step in web services. *Communications of the ACM*, 46(10):29–34, 2003.

[17] L. A. H. da Silva Maciel e E. T. Yano. Uma linguagem de workflow para composição de web services - lcws. *In 19th Brazilian Symposium on Software Engineering*, 2005.

[18] Seema Degwekar, Stanley Y. W. Su, e Herman Lam. Constraint specification and processing in web services publication and discovery. *Proceedings of the IEEE In-*

ternational Conference on Web Services, páginas 210–217. IEEE Computer Society, 2004.

[19] The eclipse foundation. The eclipse project. http://www.eclipse.org.

[20] Fatih Emekci, Ozgur D. Sahin, Divyakant Agrawal, e Amr El Abbadi. A peer-to-peer framework for web service discovery with ranking. *Proceedings of the IEEE International Conference on Web Services*, páginas 192–199. IEEE Computer Society, 2004.

[21] The Apache Software Foundation. http://www.apache.org.

[22] The Apache Software Foundation. Axis architecture guide. http://ws.apache.org/axis/java/architecture-guide.html.

[23] The Apache Software Foundation. Web services project @ apache. http://ws.apache.org/.

[24] Xiang Fu, Tevfik Bultan, e Jianwen Su. Analysis of interacting bpel web services. *Proceedings of the 13th international conference on World Wide Web*, páginas 621–630. ACM Press, 2004.

[25] Madhusudhan Govindaraju, Aleksander Slominski, Kenneth Chiu, Pu Liu, Robert van Engelen, e Michael J. Lewis. Toward characterizing the performance of soap toolkits. *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, páginas 365–372. IEEE Computer Society, 2004.

[26] Network Working Group. *RFC 1831: Remote Procedure Call Protocol Specification Version 2.* 1957.

[27] Network Working Group. *RFC 959: File Transfer Protocol.* 1985.

[28] Network Working Group. *RFC 2068: Hypertext Transfer Protocol – HTTP/1.1.* 1997.

[29] W3C Web Services Description Working Group. Web services description language (wsdl) 1.1. http://www.w3.org/TR/wsdl, 2001.

[30] Per Brinch Hansen. Java's insecure parallelism. *SIGPLAN Not.*, 34(4):38–45, 1999.

[31] Roberto André Hexsel. Núcleo multiprocessador para aplicações de tempo-real. Dissertação de Mestrado, Instituto de Matemática, Estatística e Ciência da Computação, Universidade Estadual de Campinas, 1988.

[32] G. H. Hilderink. The communicating threads for java (ctj) home page. http://www.ce.utwente.nl/javapp/.

[33] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[34] Richard Hull, Michael Benedikt, Vassilis Christophides, e Jianwen Su. E-services: a look behind the curtain. *Proceedings of the 22nd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, páginas 1–14. ACM Press, 2003.

[35] Intalio e BPMI.org. Bussiness process modeling language. http://www.bpmi.org/bpmi-downloads/BPML-SPEC-1.0.zip, 2002.

[36] International Organization for Standardization. *ISO 8879:1986: Information processing — Text and office systems — Standard Generalized Markup Language (SGML).* 1986.

[37] Komkamol Jamroendararasame, Tetsuya Suzuki, e Takehiro Tokuda. A visual approach to development of web services providers/requestors. *Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments*, páginas 251–253. IEEE Computer Society, 2003.

[38] Bas Jansen. Service profiling in business to business web services. Dissertação de Mestrado, The University of Twente, The Netherlands, dezembro de 2003.

[39] Bahman Kalali, Paulo Alencar, e Don Cowan. A service-oriented monitoring registry. *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, páginas 107–121. IBM Press, 2003.

[40] Markus Keidl e Alfons Kemper. Towards context-aware adaptable web services. *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, páginas 55–65. ACM Press, 2004.

[41] F. Leyman. Web services flow language (wsfl) 1.0, 2001.

[42] Sisi Liu, Rania Khalaf, e Francisco Curbera. From daml-s processes to bpel4ws. *Proceedings of the 14th International Workshop on Research Issues on Data Engineering: Web Services for e-Commerce and e-Government Applications*, páginas 77–84. IEEE Computer Society, março de 2004.

[43] Nikola Milanovic e Miroslaw Malek. Current solutions for web service composition. *IEEE Internet Computing*, 8(6):51–59, 2004.

[44] Peter D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[45] Martin A. Musicante e Edinardo Potrich. Expressing workflow patterns forweb services: The case of pews. *10th Brazilian Symposium on Programming Languages*, 2006. submitted.

[46] Chris Nevison. Teaching distributed and parallel computing with java and csp. *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, páginas 484–491, Washington, DC, USA, 2001. IEEE Computer Society.

[47] W3C Working Group Note. Web services architecture, w3c working group note 11 february 2004. http://www.w3.org/TR/ws-arch/.

[48] L. G. Novak e S. D. Kuznetsov. Canonical forms of xml schemas. *Programming and Computer Software*, 29(5):283–293, setembro de 2003.

[49] OAIS. Introduction to uddi: important features and functional concepts. http://uddi.org/pubs/uddi-tech-wp.pdf, 2004. UDDI Technical White Paper.

[50] OASIS. Uddi v2.03 data structure specification. http://uddi.org/pubs/DataStructure-V2.03-Published-20020719.pdf, 2002.

[51] Nicola Onose e Jerome Simeon. Xquery at your web service. *Proceedings of the 13th international conference on World Wide Web.*, páginas 603–611. ACM Press, 2004.

[52] Chris Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, outubro de 2003.

[53] Paulo F. Pires, Mario R. F. Benevides, e Marta Mattoso. Building reliable web services compositions. *Web, Web-Services, and Database Systems*, páginas 59–72, 2002.

[54] Edinardo Portrich. Pews editor, um front-end para a linguagem pews. Dissertação de Mestrado, Departamento de Informática, Universidade Federal do Paraná, 2006. Orientador: Martin A. Musicante.

[55] Jonathan B. Postel. Simple mail transfer protocol. http://www.ietf.org/rfc/rfc0821.txt, 1982. RFC 821.

[56] Ana Maria PRICE e Simão Sirineo TOSCANI. *Implementação de linguagens de programação : Compiladores*. Editora Sagra-Luzzato, 2nd edition, 2001. Instituto de Informática da UFRGS - Série Livros Didáticos.

[57] John Hamilton Reppy. *High-Order Concurrency*. Tese de Doutorado, Cornell University, Ithaca, New York, junho de 1982. TR92-1285.

[58] G. Salaün, L. Bordeaux, e M. Schaerf. Describing and reasoning on web services using process algebra. *Proceedings of the IEEE International Conference on Web Services*, páginas 43–50, julho de 2004.

[59] Nan C. Schaller, Gerald H. Hilderink, e Peter H. Welch. Using Java for Parallel Computing - JCSP versus CTJ. Peter H. Welch e Andrè W. P. Bakkers, editors, *Communicating Process Architectures 2000*, páginas 205–226, 2000.

[60] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn Bacon, 1986.

[61] IBM Web Services Architecture team. Web services architecture overview. http://www-106.ibm.com/developerworks/webservices/library/w-ovr/.

[62] S. Thatte. XLANG: Web services for business process design. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm, 2001.

[63] Henry S. Thompson, David Beech, Murray Maloney, e Noah Mendelsohn. Xml schema part 1: Structures second edition. http://www.w3.org/TR/xmlschema-1/, outubro de 2004. W3C Recommendation.

[64] Wei Tsek Tsai, Ray Paul, Z. Cao, , e B. Xiao. Verification of web services using an enhanced uddi server. *Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003).*, páginas 131–138, janeiro de 2003.

[65] Mark Turner, Fujun Zhu, Ioannis Kotsiopoulos, Michelle Russell, David Budgen, Keith Bennett, Pearl Brereton, John Keane, Paul Layzell, e Michael Rigby. Using web service technologies to create an information broker: An experience report. *Proceedings of the 26th International Conference on Software Engineering*, páginas 552–561. IEEE Computer Society, 2004.

[66] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, e A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[67] W3C. Soap version 1.2 part 1: Messaging framework. http://www.w3.org/TR/2003/REC-soap12-part1-20030624, 2003.

[68] W3C. Soap version 1.2 part 2: Adjuncts. http://www.w3.org/TR/2003/REC-soap12-part2-20030624, 2003.

[69] W3C. Extensible markup language (xml) 1.0 (third edition). http://www.w3.org/TR/2004/REC-xml-20040204/, fevereiro de 2004. W3C Recommendation.

[70] David A. Watt. *Programming Language Syntax and Semantics*. Prentice Hall International Series in Computer Science. Prentice Hall, 1991.

[71] P. H. Welch. Java threads in light of occam/csp. *Architectures, Languages and Patterns, WoTUG*, 21:259–284, 1998.

[72] Peter H. Welch e P. D. Austin. The communicating sequential processes for java (jcsp) home page. http://www.cs.kent.ac.uk/projects/ofa/jcsp/.

[73] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing Series. MIT Press, 1993.

[74] Petia Wohed, Wil M.P. van der Aalst, Marlon Dumas, e Arthur H.M. ter Hofstede. Analysis of web services composition languages: The case of bpel4ws. *Proceedings of the 22nd International Conference on Conceptual Modelling (ER)*, páginas 200–215. Springer-Verlag Heidelberg, 2003.

[75] Xiaochuan Yi e Krys J. Kochut. Cpnet model for bpel4ws workflow. Relatório técnico, Department of Computer Science, The University of Georgia, novembro de 2004.

[76] A. Zisman. An overview of xml. *Computing & Control Engineering Journal*, 11(4):165–167, agosto de 2000.

APÊNDICE A

A.1 XPEWS Schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:wsdl="
   http://schemas.xmlsoap.org/wsdl/" xmlns:pews="http://aquarius.inf.
   ufpr.br" targetNamespace="http://aquarius.inf.ufpr.br"
   elementFormDefault="qualified">

<xs:element name="envelope" type="pews:envelope"/>
<xs:complexType name="envelope">
 <xs:annotation>
  <xs:documentation>
    This is the root element of XPEWS.
    </xs:documentation>
 </xs:annotation>
 <xs:sequence>
  <xs:element name="behaviour" type="pews:behavior"/>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="behavior">
 <xs:annotation>
  <xs:documentation>
  This element specifies the way in which the client interact with
     the service.
  </xs:documentation>
 </xs:annotation>
 <xs:sequence>
  <xs:element name="import" type="pews:import" minOccurs="0"
     maxOccurs="unbounded"/>
  <xs:element name="operations" type="pews:operations" minOccurs="1"
     maxOccurs="1"/>
  <xs:element name="varDef" type="pews:varDef" minOccurs="0"
     maxOccurs="unbounded"/>
  <xs:element name="pathExp" type="pews:pathExp" minOccurs="1"
     maxOccurs="1"/>
 </xs:sequence>
 <xs:attribute name="name" type="xs:NCName" use="required"/>
</xs:complexType>

<xs:complexType name="import">
 <xs:attribute name="namespace" type="xs:anyURI" use="required"/>
 <xs:attribute name="location" type="xs:anyURI" use="required"/>
</xs:complexType>
```

```
<xs:complexType name="operations">
 <xs:annotation>
  <xs:documentation>
   This element specifies the operations which are defined by the WSDL
       portType.
  </xs:documentation>
 </xs:annotation>
 <xs:sequence>
  <xs:element name="operation" type="pews:operation" minOccurs="1"
      maxOccurs="unbounded"/>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="operation">
 <!-- The attribute name defines a local name to the operation -->
 <xs:attribute name="name" type="xs:NCName" use="required"/>
 <!-- The attribute portType is the name of a portType defined in a
     WSDL document -->
 <xs:attribute name="portType" type="xs:QName" use="optional"/>
 <!-- The attribute refersTo is the name of a operation defined in a
     WSDL portType -->
 <xs:attribute name="refersTo" type="xs:QName" use="required"/>
</xs:complexType>

<xs:complexType name="varDef">
 <xs:annotation>
  <xs:documentation>
    This element specifies the declaration of variables.
  </xs:documentation>
 </xs:annotation>
 <xs:group ref="pews:arith-expr" minOccurs="1"/>
 <xs:attribute name="name" type="xs:NCName" use="required"/>
</xs:complexType>

<xs:group name="arith-expr">
 <xs:annotation>
  <xs:documentation>
    This group specifies that the value of variables is obtained by
        the evaluation of an arithmetic expression.
  </xs:documentation>
 </xs:annotation>
 <xs:choice>
  <xs:element name="operation" type="pews:opname" minOccurs="1"
      maxOccurs="1"/>
  <xs:element name="var" minOccurs="1" maxOccurs="1" type="pews:var"/
      >
  <xs:element name="const" minOccurs="1" maxOccurs="1" type="
      pews:const"/>
  <xs:element name="libFunction" minOccurs="1" maxOccurs="1" type="
      pews:libFunc"/>
```

```
  <xs:element name="pewsCounter" minOccurs="1" maxOccurs="1" type="
     pews:pewsCounter"/>
  <xs:group ref="pews:arithOp"/>
 </xs:choice>
</xs:group>

<xs:group name="arithOp">
 <xs:annotation>
  <xs:documentation>
    This group specifies the arithmetic operations.
  </xs:documentation>
 </xs:annotation>
 <xs:choice>
  <xs:element name="minusUn" minOccurs="1" maxOccurs="1" type="
     pews:unaryOp"/>
  <xs:element name="minus" minOccurs="1" maxOccurs="1" type="
     pews:binaryOp"/>
  <xs:element name="sum" minOccurs="1" maxOccurs="1" type="
     pews:binaryOp"/>
  <xs:element name="mult" minOccurs="1" maxOccurs="1" type="
     pews:binaryOp"/>
  <xs:element name="div" minOccurs="1" maxOccurs="1" type="
     pews:binaryOp"/>
 </xs:choice>
</xs:group>

<xs:complexType name="binaryOp">
 <xs:annotation>
  <xs:documentation>
    Specifies the construction of the binary arithmetic operations.
  </xs:documentation>
 </xs:annotation>
 <xs:sequence>
  <xs:group ref="pews:arith-expr" minOccurs="1" maxOccurs="1"/>
  <xs:group ref="pews:arith-expr" minOccurs="1" maxOccurs="1"/>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="unaryOp">
 <xs:annotation>
  <xs:documentation>
    Specifies the construction of the unary arithmetic operations.
  </xs:documentation>
 </xs:annotation>
 <xs:sequence>
  <xs:group ref="pews:arith-expr" minOccurs="1" maxOccurs="1"/>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="libFunc">
 <xs:annotation>
```

```
  <xs:documentation>
    Specifies the libFunction attributes.
  </xs:documentation>
 </xs:annotation>
 <xs:attribute name="name" type="xs:NCName" use="required"/>
 <xs:attribute name="unit" type="xs:NCName" use="required"/>
</xs:complexType>

<xs:complexType name="pewsCounter">
 <xs:annotation>
  <xs:documentation>
    Specifies the pewsCounter attributes.
  </xs:documentation>
 </xs:annotation>
 <xs:attribute name="opname" type="xs:NCName" use="required"/>
 <xs:attribute name="name" type="xs:NCName" use="required"/>
 <xs:attribute name="component" type="xs:NCName" use="required"/>
 <xs:attribute name="unit" type="xs:NCName" use="required"/>
</xs:complexType>

<xs:complexType name="pathExp">
 <xs:sequence>
  <xs:group ref="pews:PE" minOccurs="1" maxOccurs="1"/>
 </xs:sequence>
</xs:complexType>

<xs:group name="PE">
 <xs:annotation>
  <xs:documentation>
    This group specifies the path operators.
  </xs:documentation>
 </xs:annotation>
 <xs:choice>
  <xs:element name="operation" minOccurs="0" maxOccurs="unbounded"
     type="pews:opname"/>
  <xs:element name="parRep" minOccurs="1" maxOccurs="1" type="
     pews:binaryPE"/>
  <xs:element name="star" minOccurs="1" maxOccurs="1" type="
     pews:unaryPE"/>
   <xs:element name="seq" minOccurs="1" maxOccurs="1" type="
      pews:binaryPE"/>
  <xs:element name="choice" minOccurs="1" maxOccurs="1" type="
     pews:binaryPE"/>
  <xs:element name="par" minOccurs="1" maxOccurs="1" type="
     pews:binaryPE"/>
  <xs:element name="pred" minOccurs="1" maxOccurs="1" type="pews:pred
     "/>
 </xs:choice>
</xs:group>

<xs:complexType name="unaryPE">
```

```
 <xs:annotation>
  <xs:documentation>
    Specifies the construction of the unary path expression.
  </xs:documentation>
 </xs:annotation>
 <xs:sequence>
  <xs:group ref="pews:PE"/>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="binaryPE">
 <xs:annotation>
  <xs:documentation>
    Specifies the construction of the binary path expression.
  </xs:documentation>
 </xs:annotation>
 <xs:sequence>
  <xs:group ref="pews:PE"/>
  <xs:group ref="pews:PE"/>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="pred">
 <xs:annotation>
  <xs:documentation>
    Specifies the construction of predicates.
  </xs:documentation>
 </xs:annotation>
  <xs:sequence>
  <xs:choice>
   <xs:element name="bool" minOccurs="0" type="pews:simplePred"/>
   <xs:group ref="pews:boolOp"/>
   <xs:group ref="pews:relOp"/>
  </xs:choice>
  <xs:group ref="pews:PE"/>
 </xs:sequence>
</xs:complexType>

<xs:group name="pred">
 <xs:annotation>
  <xs:documentation>
    This group specifies the predicate rules.
  </xs:documentation>
 </xs:annotation>
<xs:sequence>
 <xs:choice>
  <xs:element name="bool" minOccurs="0" type="pews:simplePred"/>
  <xs:group ref="pews:boolOp"/>
  <xs:group ref="pews:relOp"/>
 </xs:choice>
 <xs:group ref="pews:PE"/>
```

```
</xs:sequence>
</xs:group>

<xs:complexType name="simplePred">
 <xs:annotation>
  <xs:documentation>
    Specifies the construction of a simple predicate.
  </xs:documentation>
 </xs:annotation>
 <xs:attribute name="value" type="xs:boolean"/>
</xs:complexType>

<xs:group name="relOp">
 <xs:annotation>
  <xs:documentation>
    This group specifies the relational operators.
  </xs:documentation>
 </xs:annotation>
 <xs:choice>
  <xs:element name="leq" minOccurs="0" maxOccurs="1" type="
     pews:binRelOp"/>
  <xs:element name="gt" minOccurs="0" maxOccurs="1" type="
     pews:binRelOp"/>
 </xs:choice>
</xs:group>

<xs:complexType name="binRelOp">
 <xs:annotation>
  <xs:documentation>
    Specifies the construction of the binary relational operators.
  </xs:documentation>
 </xs:annotation>
 <xs:sequence>
  <xs:group ref="pews:arith-expr"/>
  <xs:group ref="pews:arith-expr"/>
 </xs:sequence>
</xs:complexType>

<xs:group name="boolOp">
 <xs:annotation>
  <xs:documentation>
    This group specifies the boolean operators.
  </xs:documentation>
 </xs:annotation>
 <xs:choice>
  <xs:element name="and" minOccurs="1" maxOccurs="1" type="
     pews:binBoolOp"/>
  <xs:element name="or"  minOccurs="1" maxOccurs="1" type="
     pews:binBoolOp"/>
  <xs:element name="not" minOccurs="1" maxOccurs="1" type="
     pews:binBoolOp"/>
```

```
 </xs:choice>
</xs:group>

<xs:complexType name="binBoolOp">
 <xs:annotation>
  <xs:documentation>
    Specifies the construction of the binary bool operators.
  </xs:documentation>
 </xs:annotation>
 <xs:sequence>
  <xs:group ref="pews:pred"/>
  <xs:group ref="pews:pred"/>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="opname">
 <xs:annotation>
  <xs:documentation>
    Specifies the attributes of the element operation.
  </xs:documentation>
 </xs:annotation>
 <xs:attribute name="name" type="xs:NCName" use="required"/>
</xs:complexType>

<xs:complexType name="var">
 <xs:annotation>
  <xs:documentation>
    Specifies the attributes of the element var.
  </xs:documentation>
 </xs:annotation>
 <xs:attribute name="name" type="xs:NCName" use="required"/>
</xs:complexType>

<xs:complexType name="const">
 <xs:annotation>
  <xs:documentation>
    Specifies the attributes of the element const.
  </xs:documentation>
 </xs:annotation>
 <xs:attribute name="value" type="xs:float" use="required"/>
</xs:complexType>

</xs:schema>
```

APÊNDICE B

A.2 Documentos WSDL referentes ao estudo de caso

documento WSDL referente ao serviço Cliente

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="clienteService" targetNamespace="http://
   localhost:8080/axis/samples/warehouse/wsdl/cliente.wsdl"
   xmlns:impl="http://localhost:8080/axis/samples/warehouse/wsdl/
   cliente.wsdl" xmlns:intf="urn:cliente" xmlns:apachesoap="http://
   xml.apache.org/xml-soap" xmlns:wsdlsoap="http://schemas.xmlsoap.
   org/wsdl/soap/" xmlns:soapenc="http://schemas.xmlsoap.org/soap/
   encoding/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:wsdl
   ="http://schemas.xmlsoap.org/wsdl/" xmlns="http://schemas.xmlsoap.
   org/wsdl/">

   <wsdl:message name="comprarOut">
      <wsdl:part name="in0" type="soapenc:Array"/>
   </wsdl:message>

   <wsdl:message name="levarIn">
      <wsdl:part name="in0" type="soapenc:Array"/>
   </wsdl:message>

   <wsdl:portType name="Cliente">
      <wsdl:operation name="comprar" parameterOrder="in0">
         <wsdl:output message="impl:comprarOut" name="comprarOut"/>
      </wsdl:operation>

      <wsdl:operation name="levar" parameterOrder="in0">
         <wsdl:input message="impl:levarIn" name="levarIn"/>
      </wsdl:operation>
 </wsdl:portType>

   <wsdl:binding name="lojaSoapBinding" type="impl:cliente">
      <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap
          .org/soap/http"/>

      <wsdl:operation name="comprar">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:output name="comprar">
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/
               soap/encoding/" namespace="urn:cliente" use="encoded"/
               >
         </wsdl:output>
      </wsdl:operation>
```

```
      <wsdl:operation name="levar">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="levar">
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/
               soap/encoding/" namespace="urn:cliente" use="encoded"/
               >
         </wsdl:input>
      </wsdl:operation>
 </wsdl:binding>

   <wsdl:service name="clienteService">
      <wsdl:port binding="impl:clienteSoapBinding" name="cliente">
         <wsdlsoap:address location="http://localhost:8080/axis/
            services/Loja"/>
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

documento WSDL referente ao serviço Banco

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="bancoService" targetNamespace="http://
   localhost:8080/axis/samples/warehouse/wsdl/banco.wsdl" xmlns:impl=
   "http://localhost:8080/axis/samples/warehouse/wsdl/banco.wsdl"
   xmlns:intf="urn:banco" xmlns:apachesoap="http://xml.apache.org/xml
   -soap" xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:wsdl="http://
   schemas.xmlsoap.org/wsdl/" xmlns="http://schemas.xmlsoap.org/wsdl/
   ">

   <wsdl:message name="autorizarOut">
      <wsdl:part name="autorizarOut" type="xsd:string"/>
   </wsdl:message>

 <wsdl:message name="autorizarIn">
      <wsdl:part name="in0" type="soapenc:Array"/>
   </wsdl:message>

   <wsdl:message name="enviarContaOut">
      <wsdl:part name="enviarContaOut" type="xsd:string"/>
   </wsdl:message>

   <wsdl:message name="enviarContaIn">
      <wsdl:part name="in0" type="xsd:string"/>
      <wsdl:part name="in1" type="xsd:string"/>
      <wsdl:part name="in2" type="xsd:string"/>
      <wsdl:part name="in3" type="xsd:double"/>
   </wsdl:message>

   <wsdl:message name="enviarPagamentoOut">
      <wsdl:part name="enviarPagamentoOut" type="xsd:string"/>
```

```
</wsdl:message>

  <wsdl:message name="enviarPagamentoIn">
     <wsdl:part name="in0" type="xsd:string"/>
     <wsdl:part name="in1" type="xsd:string"/>
     <wsdl:part name="in2" type="xsd:double"/>
  </wsdl:message>

  <wsdl:portType name="Banco">
     <wsdl:operation name="enviarConta" parameterOrder="in0 in1 in2
        in3">
        <wsdl:input message="impl:enviarContaIn" name="enviarContaIn
           "/>
        <wsdl:output message="impl:enviarContaOut" name="
           enviarContaOut"/>
     </wsdl:operation>

     <wsdl:operation name="enviarPagamento" parameterOrder="in0 in1
        in2">
        <wsdl:input message="impl:enviarPagamentoIn" name="
           enviarPagamentoIn"/>
        <wsdl:output message="impl:enviarPagamentoOut" name="
           enviarPagamentoOut"/>
     </wsdl:operation>

     <wsdl:operation name="autorizar" parameterOrder="in0">
        <wsdl:input message="impl:autorizarIn" name="autorizarIn"/>
        <wsdl:output message="impl:autorizarOut" name="autorizarOut"
           />
     </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="bancoSoapBinding" type="impl:banco">
     <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap
        .org/soap/http"/>
     <wsdl:operation name="enviarConta">
        <wsdlsoap:operation soapAction=""/>
        <wsdl:input name="enviarContaIn">
           <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/
              soap/encoding/" namespace="urn:banco" use="encoded"/>
        </wsdl:input>
        <wsdl:output name="enviarContaOut">
           <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/
              soap/encoding/" namespace="urn:banco" use="encoded"/>
        </wsdl:output>
     </wsdl:operation>

     <wsdl:operation name="enviarPagamento">
        <wsdlsoap:operation soapAction=""/>
        <wsdl:input name="enviarPagamentoIn">
```

```
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/
               soap/encoding/" namespace="urn:banco" use="encoded"/>
         </wsdl:input>
         <wsdl:output name="enviarPagamentoOut">
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/
               soap/encoding/" namespace="urn:banco" use="encoded"/>
         </wsdl:output>
      </wsdl:operation>

      <wsdl:operation name="autorizar">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="autorizarIn">
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/
               soap/encoding/" namespace="urn:banco" use="encoded"/>
         </wsdl:input>
         <wsdl:output name="autorizarOut">
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/
               soap/encoding/" namespace="urn:banco" use="encoded"/>
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>

   <wsdl:service name="bancoService">
      <wsdl:port binding="impl:bancoSoapBinding" name="banco">
         <wsdlsoap:address location="http://localhost:8080/axis/
            services/Banco"/>
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

documento WSDL referente ao serviço Loja

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="lojaService" targetNamespace="http://
   localhost:8080/axis/samples/warehouse/wsdl/loja.wsdl" xmlns:impl="
   http://localhost:8080/axis/samples/warehouse/wsdl/loja.wsdl"
   xmlns:intf="urn:loja" xmlns:apachesoap="http://xml.apache.org/xml-
   soap" xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:wsdl="http://
   schemas.xmlsoap.org/wsdl/" xmlns="http://schemas.xmlsoap.org/wsdl/
   ">

 <wsdl:message name="comprarIn">
      <wsdl:part name="in0" type="soapenc:Array"/>
   </wsdl:message>

   <wsdl:message name="levarOut">
      <wsdl:part name="in0" type="soapenc:Array"/>
   </wsdl:message>

 <wsdl:message name="emitirReciboIn">
```

```
      <wsdl:part name="in0" type="xsd:string"/>
      <wsdl:part name="in1" type="xsd:double"/>
   </wsdl:message>

<wsdl:message name="fazerPedidoOut">
      <wsdl:part name="fazerPedidoOut" type="soapenc:Array"/>
   </wsdl:message>

<wsdl:message name="fazerPedidoIn">
      <wsdl:part name="in0" type="xsd:string"/>
   </wsdl:message>

   <wsdl:message name="autorizarOut">
      <wsdl:part name="autorizarOut" type="soapenc:Array"/>
   </wsdl:message>

<wsdl:message name="autorizarIn">
      <wsdl:part name="in0" type="xsd:string"/>
   </wsdl:message>

   <wsdl:portType name="Loja">
      <wsdl:operation name="comprar" parameterOrder="in0">
         <wsdl:input message="impl:comprarIn" name="comprarIn"/>
      </wsdl:operation>

      <wsdl:operation name="levar" parameterOrder="in0">
         <wsdl:output message="impl:levarOut" name="levarOut"/>
      </wsdl:operation>

      <wsdl:operation name="emitirRecibo" parameterOrder="in0 in1">
         <wsdl:input message="impl:emitirReciboIn" name="
            emitirReciboIn"/>
      </wsdl:operation>

 <wsdl:operation name="fazerPedido" parameterOrder="in0">
         <wsdl:output message="impl:fazerPedidoOut" name="
            fazerPedidoOut"/>
         <wsdl:input message="impl:fazerPedidoIn" name="fazerPedidoIn
            "/>
      </wsdl:operation>

 <wsdl:operation name="autorizar" parameterOrder="in0">
         <wsdl:output message="impl:autorizarOut" name="autorizarOut"
            />
         <wsdl:input message="impl:autorizarIn" name="autorizarIn"/>
      </wsdl:operation>
   </wsdl:portType>

   <wsdl:binding name="lojaSoapBinding" type="impl:loja">
      <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap
         .org/soap/http"/>
```

```
    <wsdl:operation name="comprar">
       <wsdlsoap:operation soapAction=""/>
       <wsdl:input name="comprar">
          <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/
             soap/encoding/" namespace="urn:loja" use="encoded"/>
       </wsdl:input>
    </wsdl:operation>

    <wsdl:operation name="levar">
       <wsdlsoap:operation soapAction=""/>
       <wsdl:output name="levar">
          <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/
             soap/encoding/" namespace="urn:loja" use="encoded"/>
       </wsdl:output>
       <wsdl:input name="levar">
          <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/
             soap/encoding/" namespace="urn:loja" use="encoded"/>
       </wsdl:input>
    </wsdl:operation>

    <wsdl:operation name="autorizar">
       <wsdlsoap:operation soapAction=""/>
       <wsdl:output name="autorizarOut">
          <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/
             soap/encoding/" namespace="urn:loja" use="encoded"/>
       </wsdl:output>
       <wsdl:input name="autorizarIn">
          <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/
             soap/encoding/" namespace="urn:loja" use="encoded"/>
       </wsdl:input>
    </wsdl:operation>

    <wsdl:operation name="emitirRecibo">
       <wsdlsoap:operation soapAction=""/>
       <wsdl:input name="emitirReciboIn">
          <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/
             soap/encoding/" namespace="urn:loja" use="encoded"/>
       </wsdl:input>
    </wsdl:operation>

 <wsdl:operation name="fazerPedido">
       <wsdlsoap:operation soapAction=""/>
       <wsdl:output name="fazerPedidoOut">
          <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/
             soap/encoding/" namespace="urn:loja" use="encoded"/>
       </wsdl:output>
       <wsdl:input name="fazerPedidoIn">
          <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/
             soap/encoding/" namespace="urn:loja" use="encoded"/>
       </wsdl:input>
```

```
      </wsdl:operation>
   </wsdl:binding>

   <wsdl:service name="lojaService">
      <wsdl:port binding="impl:lojaSoapBinding" name="loja">
         <wsdlsoap:address location="http://localhost:8080/axis/
            services/Loja"/>
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

documento WSDL referente ao serviço Depósito

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="depositoService" targetNamespace="http://
   localhost:8080/axis/samples/warehouse/wsdl/deposito.wsdl"
   xmlns:impl="http://localhost:8080/axis/samples/warehouse/wsdl/
   deposito.wsdl" xmlns:intf="urn:deposito" xmlns:apachesoap="http://
   xml.apache.org/xml-soap" xmlns:wsdlsoap="http://schemas.xmlsoap.
   org/wsdl/soap/" xmlns:soapenc="http://schemas.xmlsoap.org/soap/
   encoding/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:wsdl
   ="http://schemas.xmlsoap.org/wsdl/" xmlns="http://schemas.xmlsoap.
   org/wsdl/">

   <wsdl:message name="fazerPedidoIn">
      <wsdl:part name="in0" type="soapenc:Array"/>
   </wsdl:message>

   <wsdl:message name="fazerPedidoOut">
      <wsdl:part name="fazerPedidoOut" type="xsd:string"/>
   </wsdl:message>

   <wsdl:message name="emitirReciboOut">
      <wsdl:part name="emitirReciboOut" type="xsd:string"/>
   </wsdl:message>

   <wsdl:message name="enviarPagamentoOut">
      <wsdl:part name="enviarPagamentoOut" type="xsd:string"/>
   </wsdl:message>

   <wsdl:message name="enviarPagamentoIn">
      <wsdl:part name="in0" type="xsd:string"/>
      <wsdl:part name="in1" type="xsd:string"/>
      <wsdl:part name="in2" type="xsd:double"/>
   </wsdl:message>

   <wsdl:message name="enviarContaIn">
      <wsdl:part name="in0" type="xsd:string"/>
   </wsdl:message>

   <wsdl:message name="enviarContaOut">
      <wsdl:part name="in0" type="xsd:string"/>
```

```
    <wsdl:part name="in1" type="xsd:string"/>
    <wsdl:part name="in2" type="xsd:string"/>
    <wsdl:part name="in3" type="xsd:double"/>
  </wsdl:message>

  <wsdl:portType name="Deposito">
    <wsdl:operation name="enviarPagamento" parameterOrder="in0">
      <wsdl:output message="impl:enviarPagamentoOut" name="
         enviarPagamentoOut"/>
      <wsdl:input message="impl:enviarPagamentoIn" name="
         enviarPagamentoIn"/>
    </wsdl:operation>

 <wsdl:operation name="enviarConta" parameterOrder="in0">
      <wsdl:output message="impl:enviarContaOut" name="
         enviarContaOut"/>
      <wsdl:input message="impl:enviarContaIn" name="enviarContaIn
         "/>
    </wsdl:operation>

    <wsdl:operation name="fazerPedido" parameterOrder="in0">
      <wsdl:input message="impl:fazerPedidoIn" name="fazerPedidoIn
         "/>
      <wsdl:output message="impl:fazerPedidoOut" name="
         fazerPedidoOut"/>
    </wsdl:operation>

    <wsdl:operation name="emitirRecibo" parameterOrder="in0">
      <wsdl:output message="impl:emitirReciboOut" name="
         emitirReciboOut"/>
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="depositoSoapBinding" type="impl:deposito">
    <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap
       .org/soap/http"/>

    <wsdl:operation name="enviarPagamento">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:output name="enviarPagamentoOut">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/
           soap/encoding/" namespace="urn:deposito" use="encoded"
           />
      </wsdl:output>
      <wsdl:input name="enviarPagamentoIn">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/
           soap/encoding/" namespace="urn:deposito" use="encoded"
           />
      </wsdl:input>
    </wsdl:operation>
```

```
    <wsdl:operation name="fazerPedido">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="fazerPedidoIn">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/
          soap/encoding/" namespace="urn:deposito" use="encoded"
          />
      </wsdl:input>
      <wsdl:output name="fazerPedidoOut">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/
          soap/encoding/" namespace="urn:deposito" use="encoded"
          />
      </wsdl:output>
    </wsdl:operation>

    <wsdl:operation name="enviarConta">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:output name="enviarContaOut">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/
          soap/encoding/" namespace="urn:deposito" use="encoded"
          />
      </wsdl:output>
      <wsdl:input name="enviarContaIn">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/
          soap/encoding/" namespace="urn:deposito" use="encoded"
          />
      </wsdl:input>
    </wsdl:operation>

    <wsdl:operation name="emitirRecibo">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:output name="emitirReciboOut">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/
          soap/encoding/" namespace="urn:deposito" use="encoded"
          />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="depoisitoService">
    <wsdl:port binding="impl:depositoSoapBinding" name="deposito">
      <wsdlsoap:address location="http://localhost:8080/axis/
        services/Deposito"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

APÊNDICE C

A.3 Documentos XPEWS referentes ao estudo de caso

documento XPEWS referente ao serviço Cliente

```
<envelope xmlns="http://aquarius.inf.ufpr.br" xmlns:xsi="http://www.
   w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
   aquarius.inf.ufpr.br pews.xsd">
    <behaviour name="Cliente" xmlns:cliente="http://localhost:8080/
       axis/samples/loja/wsdl/cliente.wsdl">
        <operations>
            <operation name="comprar" portType="Cliente" refersTo="
               cliente:comprar"/>
            <operation name="levar" portType="Cliente" refersTo="
               cliente:levar"/>
        </operations>
        <pathExp>
            <seq>
                <operation name="comprar"/>
                <operation name="levar"/>
            </seq>
        </pathExp>
    </behaviour>
</envelope>
```

documento XPEWS referente ao serviço Banco

```
<envelope xmlns="http://aquarius.inf.ufpr.br" xmlns:xsi="http://www.
   w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
   aquarius.inf.ufpr.br pews.xsd">
    <behaviour name="Banco" xmlns:banco="http://localhost:8080/axis/
       samples/loja/wsdl/banco.wsdl">
        <operations>
            <operation name="autorizar" portType="Banco" refersTo="
               banco:autorizar"/>
            <operation name="enviarConta" portType="Banco" refersTo="
               banco:enviarConta"/>
            <operation name="enviarPagamento" portType="Banco"
               refersTo="banco:enviarPagamento"/>
            <operation name="abortarOperacao" portType="Banco"
               refersTo="banco:abortarOperacao"/>
        </operations>
        <pathExp>
          <choice>
            <seq>
                <operation name="autorizar"/>
                <operation name="enviarConta"/>
                <operation name="enviarPagamento"/>
```

```
  </seq>
                <operation name="abortarOperacao"/>
          </choice>
       </pathExp>
    </behaviour>
</envelope>
```

documento XPEWS referente ao serviço Loja

```
<envelope xmlns="http://aquarius.inf.ufpr.br" xmlns:xsi="http://www.
   w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
   aquarius.inf.ufpr.br pews.xsd">
    <behaviour name="Loja" xmlns:loja="http://localhost:8080/axis/
       samples/loja/wsdl/loja.wsdl">
       <operations>
           <operation name="comprar" portType="Loja" refersTo="
              loja:comprar"/>
           <operation name="autorizar" portType="Loja" refersTo="
              loja:autorizar"/>
           <operation name="fazerPedido" portType="Loja" refersTo="
              loja:fazerPedido"/>
           <operation name="emitirRecibo" portType="Loja" refersTo="
              loja:emitirRecibo"/>
           <operation name="levar" portType="Loja" refersTo="
              loja:levar"/>
           <operation name="abortarOperacao" portType="Loja"
              refersTo="loja:abortarOperacao"/>
       </operations>
       <pathExp>
        <choice>
              <seq>
                  <operation name="comprar"/>
                  <operation name="autorizar"/>
                  <operation name="fazerPedido"/>
                  <operation name="emitirRecibo"/>
                  <operation name="levar"/>
              </seq>
              <operation name="abortarOperacao"/>
  </choice>
       </pathExp>
    </behaviour>
</envelope>
```

documento XPEWS referente ao serviço Depósito

```
<envelope xmlns="http://aquarius.inf.ufpr.br" xmlns:xsi="http://www.
   w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
   aquarius.inf.ufpr.br pews.xsd">
    <behaviour name="Deposito" xmlns:deposito="http://localhost:8080/
       axis/samples/loja/wsdl/deposito.wsdl">
       <operations>
```

```
        <operation name="fazerPedido" portType="Deposito"
           refersTo="deposito:fazerPedido"/>
        <operation name="enviarConta" portType="Deposito"
           refersTo="deposito:enviarConta"/>
        <operation name="enviarPagamento" portType="Deposito"
           refersTo="deposito:enviarPagamento"/>
        <operation name="emitirRecibo" portType="Deposito"
           refersTo="deposito:emitirRecibo"/>
        <operation name="abortarOperacao" portType="Deposito"
           refersTo="deposito:abortarOperacao"/>
    </operations>
    <varDef name="tpay">
        <minus>
            <libFunction name="now" unit="hours"/>
            <pewsCounter component="time" name="term"
                opname="enviarConta" unit="hours"/>
        </minus>
    </varDef>
    <pathExp>
        <seq>
            <operation name="fazerPedido"/>
            <operation name="enviarConta"/>
            <choice>
                <seq>
                    <pred>
                        <leq>
                            <var name="tpay"/>
                            <const value="12"/>
                        </leq>
                        <operation name="enviarPagamento"/>
                    </pred>
                    <operation name="emitirRecibo"/>
                </seq>
                <pred>
                    <gt>
                        <var name="tpay"/>
                        <const value="12"/>
                    </gt>
                    <operation name="abortarOperacao"/>
                </pred>
            </choice>
        </seq>
    </pathExp>
  </behaviour>
</envelope>
```