EDINARDO POTRICH

PEWS EDITOR, UM FRONT-END 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

Agradecimentos

Em primeiro lugar a Deus, pela benção de ter forças de continuar a buscar o novo.

Ao Professor Martin A. Musicante, pela oportunidade que mudou minha vida e pela sua preciosa paciência, orientação, apoio e participação durante o desenvolvimento deste trabalho.

Aos membros da banca, por terem aceitado avaliar este trabalho e pelos expressivos e frutíferos comentários feitos.

À minha família, em especial aos meus pais, que sempre me apoiaram, pelo incentivo e pelo companheirismo, em todos os momentos.

À Universidade Federal do Paraná, seus funcionários e professores, por todo esforço em manter a educação, pesquisa e desenvolvimento de qualidade em nosso país.

Aos colegas e amigos, que de certa forma contribuíram para a realização deste trabalho.

À minha amada esposa Marciana, que nunca deixou de me apoiar, me cuidar, me amar e, principalmente, compreender, pelo tempo que precisei dedicar ao trabalho e não pude estar com ela.

Sumário

Lista de Figuras

Resumo

PEWS é uma linguagem de composição de serviços web. Composições PEWS podem ser utilizadas para a descrição de serviços web simples tanto quanto compostos. Serviços web simples são construídos a partir de programas em Java[1] enquanto que serviços web compostos são construídos a partir da composição de serviços web já existentes. PEWS possui uma versão XML chamada XPEWS, permitindo que a linguagem possa ser utilizada também no nível de interface.

Com o objetivo de facilitar a utilização de PEWS, é apresentado neste trabalho o desenvolvimento de um *front-end* na forma de um *plug-in* para a plataforma Eclipse, permitindo uma maior integração com outras ferramentas como editores XML, WSDL e Java. O uso do *plug-in* pode ajudar na redução do tempo de desenvolvimento das composições, permitindo verificação de erros de codificação e geração de código XPEWS, aumentando assim a produtividade do desenvolvedor.

Finalmente, um estudo de caso é elaborado, analisando a linguagem PEWS do ponto de vista da sua expressividade, mediante a avaliação da linguagem, com base em um *framework* composto por padrões para *workflow*. Este estudo de caso nos permite apresentar uma comparação com outras linguagens de composição de serviços web, baseada no mesmo *framework*.

[1] Java e todas as marcas baseadas em Java são marcas da Sun Microsystems, Inc. nos Estados Unidos e/ou em outros países.

Abstract

PEWS is a language for web services composition. PEWS programs can be used for the description of simple web services as well as for the description of composite web services. Simple web services are constructed from Java programs whereas composites web services are constructed from the combination of existing web services. PEWS possesss a XML version called XPEWS, allowing the language to be used at the interface level.

The goals of this work are to present the developement of a programming environment for PEWS and to analyze the expressiveness of the language. The first goal is achieved by the the implementation of an Eclipse plug-in, allowing a bigger integration with other tools as XML, WSDL and Java editors. The use of plug-in can help reducing the time for development of the compositions, allowing verification of codification errors and generation XPEWS code, thus increasing the productivity of the developer.

The second goal of this work is achieved by the elaboration of a case study, by evaluating the language in relation to a framework composed by workflow patterns. This case study allows us to present a comparison with other languages for web service composition, based on the same framework.

Capítulo 1

Introdução

O desenvolvimento dos serviços web, como uma tecnologia modular que permite uma interoperabilidade para prover serviços de forma global, é uma peça fundamental para a evolução da área dos processos de negócios [BENNETT et al., 2004]. Esta tecnologia tem como base os serviços web, implementados através de funções e processos descritos em documentos XML [SANT'ANNA, 2005]. Esta tecnologia visa a normatização da descrição, execução, e supervisão dos processos de negócios, proporcionando aos usuário um elevado grau de liberdade no desenvolvimento de aplicações que irão utilizar estes serviços.

A tecnologia de processos de negócios baseada em XML, utiliza componentes como:

- SOAP, para a troca de mensagens [W3C, 2003a];
- WSDL, para a descrição dos serviços [W3C, 2004];
- UDDI, para publicação e descoberta de serviços [OASIS, 2004].

Podem-se, também, utilizar outros protocolos para providenciar características adicionais nas interações, como validação, coordenação e monitoramento. Características estas, providas por linguagens de composição como BPEL4WS [MALEK; MILANOVIC, 2004], WSCI [BROGI et al., 2004], WSTL [PIRES et al., 2002] e PEWS [BA et al., 2005].

A proposta deste trabalho é de implementar o *front-end* para a linguagem de composição PEWS. Uma implementação permitirá uma melhor análise do uso prático da

linguagem, levando a possíveis novas propostas como a implementação do *back-end* [CARRERO; MUSICANTE, 2006] ou adaptações na linguagem.

Para tratar o problema, em uma primeira abordagem, pensou-se em desenvolver uma ferramenta autônoma para edição de documentos PEWS com posterior tradução destes para XPEWS, mas o mesmo não teria uma integração transparente com outra ferramentas como editores XML ou WSDL.

Devido a isso, optou-se por uma nova abordagem, que consiste em produzir um *front-end* através de um *plug-in* Eclipse, o qual possui suporte à construção de novas ferramentas e disponibilidade de integração com outras ferramentas já disponíveis [GLOZIC; MELHEM, 2003].

O *plug-in* é desenvolvido na linguagem Java, e terá como premissa auxiliar a edição de composições PEWS, realizando análise léxica, sintática e semântica e por fim, geração de código XPEWS.

1.1 Objetivos

Este trabalho consiste na definição e implementação de um ambiente de programação (*front-end*) para a linguagem de composição PEWS, a qual não possui ainda suporte computacional, e no desenvolvimento de um estudo de caso que servirá como base para verificar a expressividade da linguagem PEWS, bem como verificar a usabilidade do *front-end*. O *front-end* tem como foco usuários com conhecimento mais técnico, e auxiliará na:

- composição de serviços web através de PEWS;
- tradução de PEWS para XPEWS;
- interação com outras ferramentas como editores WSDL e XML

Este trabalho tem a finalidade de:

- auxiliar na composição de serviços web;
- prover melhor análise do uso prático da linguagem PEWS, permitindo posteriores contribuições à mesma;

- realizar comparações com outras linguagens propostas.

1.2 Estrutura desta dissertação

O trabalho está organizado da seguinte maneira: o Capítulo 2 apresenta a linguagem XML, para especificação de dados semi-estruturados. O Capítulo 3 fornece uma visão sobre serviços web, tecnologias utilizadas na sustentação dos mesmos e uma visão geral das linguagens de composição de serviços web como BPEL4WS, WSCI e PEWS. o Capítulo 4 fornece uma visão sobre a plataforma Eclipse, sua arquitetura e capacidade de oferecer sustentação à construção de novas ferramentas e a descrição do desenvolvimento do *plug-in* **PEWS Editor**. O Capítulo 5 define o estudo de caso, mediante a avaliação sistemática da linguagem PEWS, com base em um framework composto por padrões para *workflow*. As conclusões obtidas durante todo o estudo realizado na elaboração deste trabalho juntamente com as contribuições e planos futuros estão especificados no Capítulo 6, que encerra o trabalho.

Capítulo 2

XML

XML, *eXtensible Markup Language*, é uma linguagem padrão para se representar dados semi-estruturados, proposta pelo W3C (World Wide Web Consortium) com a finalidade de atender às necessidades de comunicação entre sistemas, fornecendo uma identificação flexível para qualquer tipo de informação, formatação e especificação de documentos [ABITEBOUL et al., 2003].

Dentro da comunicação de sistemas, XML pode ser utilizado para troca de dados, apresentando características como por exemplo: informações sobre o destino dos dados (ex.: tabela destino), significado dos dados, suas relações e tipos; contemplados através de elementos e atributos de XML, podendo ainda ser acrescido de estruturas para validação do documento [SANT'ANNA, 2005].

XML é usando em nosso trabalho para publicar o comportamento dos serviços (composição) através da linguagem XPEWS que é apresentada na seção 3.2.1.3.

2.1 Estrutura dos documentos XML

Um documento XML é constituído de *marcações* e *texto*. As marcações definem elementos de dados e o texto fornece o dado em si, como pode ser visto no documento apresentado no Exemplo 1.

Exemplo 1 *O seguinte exemplo apresenta a marcação* <nome> *que inicia e define o elemento como* nome*, e o texto entre as marcações é o dado contido neste elemento. A*

marcação </nome> *delimita o fim do elemento.*

```
<nome>Edinardo Potrich</nome>
```

Elementos podem estar contidos dentro de outros elementos de forma à representar informações mais complexas como pode ser visto no Exemplo 2.

Exemplo 2 *O seguinte exemplo define o agrupamento entre os elementos nome, endereço e cidade para formar o elemento aluno. A marcação* <cidade/> *define o elemento* cidade *como vazio.*

```
<aluno>
   <nome>Edinardo Potrich</nome>
   <endereco>Rua Pedro Locatelli Jr, 94</endereco>
   <cidade/>
</aluno>
```

Os elementos são a base de um documento XML, mas eles não são os únicos itens de informação que estão presentes em um documento. Um documento XML é constituído por [YERGEAU, 2004]:

- Declarações XML;
- Instruções de processamento;
- Comentários;
- Elementos;
- Atributos.

Um documento XML contendo os itens acima citados é apresentado no Exemplo 3.

Exemplo 3 *O seguinte exemplo apresenta um documento XML formado pelo seus diversos itens.*

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="alunos.xsl"?>
<!-- Alunos da UFPR -->
<alunos>
   <aluno>
      <nome idade="24">Edinardo Potrich</nome>
      <endereco>Rua Pedro Locatelli Jr, 94</endereco>
      <cidade/>
   </aluno>
</alunos>
```

Cada item presente no Exemplo 3 é explicado a seguir.

2.1.1 Declarações XML

A declaração XML no inicio de um documento XML não é obrigatória, mas é a maneira de informar a um processador que o arquivo trata-se de um documento XML. A declaração inicia com <?xml e é seguido pelo único item obrigatório da declaração chamado *version* que deve conter o valor da versão atual "1.0". Pode conter ainda o item *standalone* que especifica se outro arquivo é necessário para formatar ou validar o documento, e o item *encoding* que especifica o conjunto de caracteres em uso no documento. Os valores mais comuns para o item *encoding* são:

- ISO-8859-1 - Alfabeto latino 1. Utilizado pela maioria das linguagens do oeste europeu. Similar ao ASCII de 8 bits. Permite o uso de acentuação.
- UTF-8 - Unicode de 8 bits utilizado para conjuntos de caracteres internacionais.
- UTF-16 - Unicode de 16 bits utilizado para conjuntos de caracteres internacionais.

A declaração termina com ?> como é apresentado no Exemplo 4.

Exemplo 4 *O seguinte exemplo apresenta uma declaração XML com seus itens.*

```
<?xml version="1.0" standalone="yes" encoding="UTF-8"?>
```

2.1.2 Instruções de processamento

Instruções de processamento são facultativas em um documento XML. Quando presentes, iniciam com <? e terminam com ?>, usando uma sintaxe semelhante à declaração XML.

Instruções de processamento permitem aos documentos conter instruções adicionais para aplicações externas [SYSTINET, 2000]. Essas instruções são utilizadas, por exemplo, para instruir um aplicativo sobre como manipular os dados em um documento XML, como é apresentando no Exemplo 5.

Exemplo 5 *O seguinte exemplo apresenta o uso de uma instrução de processamento para formatar a exibição de um documento XML, aplicando uma folha de estilo.*

```
<?xml-stylesheet type="text/xsl" href="alunos.xsl"?>
```

2.1.3 Comentários

Comentários podem ser adicionados em qualquer parte do documento XML, exceto dentro de uma marcação. Comentários iniciam com <!-- e terminam com --> podendo conter várias linhas. A única restrição é que não pode haver -- dentro de um comentário.

2.1.4 Elementos

Representam os dados em um documento XML. O conteúdo dos elementos pode ser vazio. Quando o elemento possui conteúdo não vazio, o conteúdo deve estar entre a marcação de abertura e a marcação de fechamento, como é apresentado no Exemplo 6.

Exemplo 6 *O seguinte exemplo apresenta um elemento com conteúdo não vazio.*

```
<nome>Edinardo Potrich</nome>
```

Elementos vazios são aqueles que não possuem conteúdo ou elementos filhos, como é apresentado no Exemplo 7.

Exemplo 7 *O seguinte exemplo apresenta um elemento com conteúdo vazio.*

```
<cidade></cidade>
```

Ou em sua forma abreviada com a barra antes do fechamento da marcação, indicando que a marcação representa ao mesmo tempo o inicio e o termino do elemento, como é apresentado no Exemplo 8

Exemplo 8 *O seguinte exemplo apresenta um elemento com conteúdo vazio de forma abreviada.*

```
<cidade/>
```

Dados em um documento XML são semi-estruturados. Sendo assim, elementos podem conter outros elementos, chamados de elementos filhos, como é apresentado no Exemplo 9.

Exemplo 9 *O seguinte exemplo apresenta o elemento nome que está contido dentro do elemento aluno.*

```
<aluno>
   <nome>Edinardo Potrich</nome>
</aluno>
```

Documentos XML podem conter um número ilimitado de elementos aninhados.

2.1.5 Atributos

São itens de dados associados a uma marcação, na forma atributo=“valor”. Aparecem somente na marcação de abertura e seus valores devem ser sempre cercados por aspas (simples ou duplas), como é apresentado no Exemplo 10.

Exemplo 10 *O seguinte exemplo exibe como os atributos podem ser apresentados.*

```
<nome idade="24">Edinardo Potrich</nome>
```

Se o valor do atributo contiver aspas duplas internamente devem-se utilizar aspas simples para representar a informação ou vice-versa, como é apresentado no Exemplo 11.

Exemplo 11 *O seguinte exemplo apresenta as formas de como pode ser tratado valores que contém aspas.*

```
<aluno nome='John "Maddog" Hall'/>
<aluno nome="John 'Maddog' Hall"/>
```

Não existe uma regra que especifique quando usar atributos e quando usar elementos filhos, apenas devemos levar em conta algumas considerações ao se utilizar atributos:

- Atributos não podem conter mais de um valor;
- Atributos não podem descrever estruturas aninhadas por não contemplar níveis.

2.2 Documentos bem formados e documentos válidos

A especificação XML define que a estrutura básica de um documento bem formado consiste de um cabeçalho e de um elemento raiz [YERGEAU, 2004]. O cabeçalho contém informações a respeito do documento e é composto da declaração XML, das instruções de processamento e das definições referentes ao uso de estruturas para a validação do documento XML. Já o elemento raiz consiste de um único elemento que contém todos os outros elementos e atributos do documento.

Para um documento XML ser validado, este deve satisfazer uma estrutura pré-definida [FOURER LEO LOPES, 2004]. Entre as várias notações criadas com o propósito de descrever a estrutura a ser satisfeita, podemos citar o *XML Schema*, que descreve a estrutura, o conteúdo e as restrições de um documento XML [RAMANUJAN; ROY, 2001]. *XML Schema* possui um sistema de tipos bem definido, incluindo tipos básicos (como inteiros e strings, entre outros) e construtores para definir tipos compostos.

2.2.1 DOM

DOM (*Document Object Model*) é uma especificação definida pelo W3C para representar documentos XML em um modelo orientado a objetos [W3C, 2000]. DOM tem a função de definir o conjunto de interfaces, que irá descrever a estrutura de um documento XML em

um formato de árvore. Estas interfaces serão a base para se obter e manipular os dados desta árvore.

Hoje em dia, podemos encontrar diversas APIs (*Application Programming Interfaces*) que implementam a especificação DOM. Uma delas é o JAXP (*Java API for XML Processing*) [SUN MICROSYSTEMS, INC, 2006] que foi escolhida para este trabalho e será utilizada para a criação de documentos XML, utilizado em conjunto com *XML Schema* para a validação dos documentos XML.

Capítulo 3

Serviços web

Os serviços web em seus vários usos, são vistos como uma importante e emergente tecnologia, que está em constante mudança e amadurecimento. Os serviços web visam automatizar os processos de negócios através da integração de aplicações distribuídas e autônomas, possibilitando acesso a recursos e reusabilidade de software na Internet [BENNETT et al., 2004]. Apesar de não haver ainda um consenso entre os autores, podemos citar a definição abaixo como a mais completa.

"Serviço web é um conjunto de padrões projetados para suportar a interoperabilidade entre aplicações sobre uma rede, independente da plataforma de hardware e de software. Os serviços web têm uma relação descrita em um determinado formato (especificamente WSDL). Outros serviços interagem com o serviço web de maneira especificada em sua descrição, usando mensagens SOAP, que utilizam tipicamente o HTTP com uma serialização XML em conjunto com outros padrões web relacionados" [W3C, 2004].

A infra-estrutura disponível para serviço web é composta por três entidades: *service provider*, *service requester* e *service broker* [BODOFF et al., 2005], como ilustra a Figura 3.1. Nesta figura também são ilustradas as ligações, que definem operações de interação entre os serviços web, as quais são: publicação (*publish*), busca (*find*) e ligação (*bind*).

As operações ocorrem na seguinte ordem:

1. O *service provider* registra seus serviços (metadados) em um repositório que é ge-

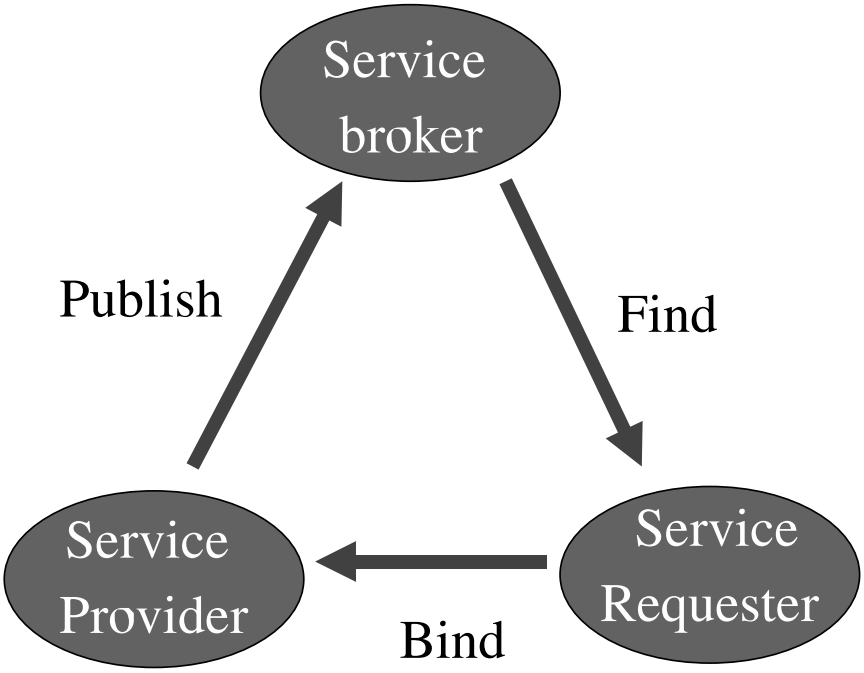


Figura 3.1: Infra-estrutura para serviço web [CAO et al., 2003]

renciado pelo *service broker*;

2. O *service broker* disponibiliza um conjunto de interfaces para utilização dos serviços web nele registrados;

3. O *service requester* solicita ao *service broker* o serviço que deseja, bem como instruções de como utilizar o mesmo;

4. Uma vez que o *service requester* encontra o serviço desejado, o *service broker* retorna os detalhes de utilização do serviço;

5. O *service requester* executa a chamada ao serviço provido pelo *service provider*;

6. O *service provider* retorna o resultado da execução ao *service requester*, finalizando a operação.

3.1 Tecnologias para construção de serviços web

As tecnologias necessárias para o desenvolvimento dos serviços web dividem-se em três áreas: comunicação, descrição e reconhecimento de serviços, onde as especificações comu-

mente utilizadas para cada uma delas são:

- **SOAP** (*Simple Object Access Protocol*), protocolo que utiliza o padrão XML para a representação de mensagens entre os serviços web;
- **WSDL** (*Web Services Description Language*), linguagem baseada no padrão XML que provê uma descrição das operações providas pelo serviço web;
- **UDDI** (*Universal Description, Discovery, and Integration*), registro que contém a descrição dos serviços web, responsável pela divulgação de serviços disponíveis em um provedor.

Cada uma das tecnologias acima mencionadas é explicada a seguir:

3.1.1 SOAP

O **SOAP** é uma especificação para troca de mensagens entre serviços web [W3C, 2003a], na forma de documento XML, independente de plataforma e transmitido sobre protocolos como HTTP, FTP ou SMTP.[ANDREWS et al., 2003]

No serviço de SOAP podem ser utilizados desde a troca de mensagens simples do tipo *request-response*, executando uma função e recebendo uma resposta, até chamadas **RCP** (*Remote Procedure Call*), encapsulando as chamadas a métodos e procedimentos segundo o padrão SOAP [W3C, 2003b; CUNHA, 2002].

A Figura 3.2 ilustra a estrutura básica das mensagens do protocolo **SOAP**, contendo os seguintes elementos [KEIDL; KEMPER, 2004]:

- **Envelope** (*envelope*), elemento obrigatório que identifica o documento XML como uma mensagem SOAP;
- **Header** (*cabeçalho*), elemento opcional que contém meta-informações de cabeçalho, tais como autenticação, assinatura, transações e etc;
- **Body** (*corpo*), elemento obrigatório que contém informações de chamada e resposta aos serviços web.

Figura 3.2: Estrutura do protocolo SOAP [KEIDL; KEMPER, 2004]

Um exemplo para a estrutura apresentada na Figura 3.2 é apresentado a seguir.

Exemplo 12 *O seguinte fragmento ilustra o uso de SOAP para executar um serviço web disponibilizado por um provedor de serviços. Neste exemplo, o provedor disponibilizou um serviço para calcular o fatorial de um número. O serviço recebe como entrada um número inteiro positivo, e devolve como resultado o seu fatorial. Para realizar a operação, o serviço possui um procedimento chamado* fat.

```
<?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:fat
    soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:ns1="fat">
     <arg1xsi:type="xsd:int">6</ arg1>
  </ns1:fat>
  </soapenv:Body>
</soapenv:Envelope>
```

O Exemplo 12 mostra como é descrita uma chamada de procedimento usando SOAP. O procedimento a ser executado, chamado `fat`, é descrito pelo elemento `ns1:fat`. Esse

elemento possui um único sub-elemento chamado `arg1`, definindo 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. Neste exemplo, deseja-se calcular o fatorial para o argumento de valor 6.

3.1.2 WSDL

O **WSDL** é uma linguagem XML que descreve as funcionalidades providas por um serviço web [W3C, 2004].

Para definição dos serviços web providos, o WSDL é dividido em dois grupos, compostos pelos seguintes elementos [GUNZER, 2002; W3C, 2004]:

- Abstratos
  - **Types**, representa os tipos de dados usados pelo serviço web: integer, string, etc;
  - **Message**, é o elemento que define o formato ou tipo de dados que uma mensagem pode utilizar na comunicação;
  - **Operation**, é uma descrição de uma ação que pode ser executada pelo serviço, definindo estruturas de entrada, saída e tratamento de exceções;
  - **PortType**, é um conjunto de operações sustentadas pelo serviço;
- Concretos
  - **Binding**, realiza dois mapeamentos com as mensagens e operações definidas pelo **PortType**. O primeiro mapeamento define qual protocolo será utilizado na comunicação (HTTP, SMTP, ou outro protocolo disponível). O segundo define o formato de requisição (RPC ou documento);
  - **Port**, um ponto de acesso que é único, definido pela combinação de um **Binding** e um endereço real de rede;
  - **Service**, é o conjunto de **Ports** relacionadas.

O seguinte exemplo mostra a descrição abstrata de um serviço utilizando-se WSDL.

Exemplo 13 *O seguinte fragmento mostra definições dos tipos de dados com que o serviço web pode operar (como os tipos inteiro, string e real, definidos a partir de XML Schema), e um elemento* portType *que contempla as operações* order *e* bill*, que serão providas pelo serviço web.*

```
<message name="productOrderIn">
  <part name="prodCode"
    type="xsd:string"/>
  <part name="quantity"
    type="xsd:integer"/>
</message>
<message name="productOrderOut">
  <part name="price"
    type="xsd:real"/>
</message>
<message name="sendBill">
  <part name="prodCode"
    type="xsd:string"/>
  <part name="total"
    type="xsd:real"/>
</message>
<message name="ackBill">
  <part name="prodCode"
    type="xsd:string"/>
</message>
...
<portType name="WarehousePortType">
  <operation name="order">
    <input message="productOrderIn"/>
    <output message="productOrderOut"/>
  </operation>
  <operation name="bill">
    <output message="sendBill"/>
    <input message="ackBill"/>
  </operation>
</portType>
```

No Exemplo 13, **order** é uma operação de requisição com resposta, uma vez que ela primeiro recebe como entrada a mensagem **productOrderIn**, com os parâmetros **prodCode** (tipo string) e **quantity** (tipo inteiro) e depois retorna como resposta a mensagem **productOrderOut**, contendo o parâmetro **price** (tipo real). Já a operação **bill** opera de forma contrária, enviando primeiramente a mensagem de solicitação **sendBill**, com

os parâmetros **prodCode** (tipo string) e **total** (tipo real) e aguarda o recebimento da mensagem de resposta **ackBill**, com o parâmetro **prodCode** (tipo string).

3.1.3 UDDI

O **UDDI** é um protocolo que define padrões para organização, registro, publicação e descoberta de serviços web. O UDDI descreve um registro (meta-dados) de serviços web e interfaces, oferecendo mecanismos para classificar, catalogar e gerenciar serviços web, fazendo com que eles possam ser descobertos e utilizados por outros aplicativos, independente de linguagem ou localização [OASIS, 2004].

A Figura 3.3 ilustra a organização de um registro **UDDI** e seus principais elementos [OASIS, 2002]:

- O **businessEntity** equivale as "páginas brancas"(white pages) descrevendo a companhia: nome, endereço, contatos, etc;
- O **businessService** equivale as "páginas amarelas"(yellow pages) incluindo as informações sobre os serviços oferecidos;
- O **bindingTemplate** equivale as "páginas verdes"(green pages) descrevendo a interface para utilizar o serviço, em um nível de detalhe suficiente para que o mesmo possa ser executado;
- O **tModel** descreve a forma como os serviços são definidos no documento UDDI, através de atributos como transporte, assinatura digital, etc. Em muitos casos, o *tModel* contém uma ligação para um documento WSDL que descreve a interface SOAP do serviço.

3.2 Composição de serviços web

Com a maturidade e padronização na criação de serviços web, aumenta a oferta dos mesmos, bem como surge a necessidade de utilizar um ou mais serviços para obter a solução de problemas específicos.

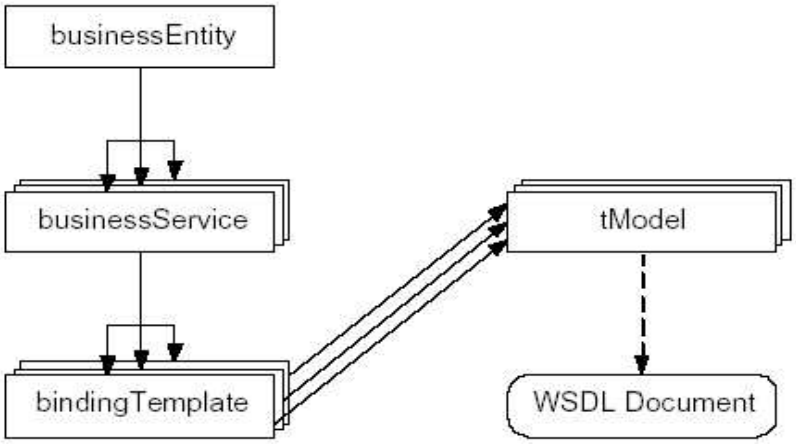


Figura 3.3: Organização do registro UDDI [KEIDL; KEMPER, 2004]

Para satisfazer esta necessidade, utilizam-se serviços compostos, os quais combinam serviços mais simples seguindo um certo padrão [AGARWAL, 2003] para alcançar um objetivo de negócio, resolver um problema complexo ou ainda prover novas funcionalidades a serviços web existentes ou criar novos serviços.

A composição de serviços web pode ser usada para:

- **Coordenação**: controle das execuções dos serviços componentes e gerenciamento do fluxo de dados entre os mesmos;
- **Monitoramento**: coleta das informações produzidas pelos serviços componentes;
- **Validação**: garantia da integridade da composição, verificação de parâmetros, imposição de restrições e aplicação das regras de negócio;
- **Qualidade de serviço**: levantamento, agregação e avaliação de qualidade de serviço (QoS) de cada componente para produzir uma composição que inclui aspectos como escalabilidade, desempenho, autenticação, privacidade, custo, segurança, integridade e disponibilidade [HU, 2004].

Para a composição de serviços web, dois paradigmas norteiam as composições: a coreografia (*choreography*) e a orquestração (*orchestration*) [BORDEAUX et al., 2004]. A orquestração se preocupa com o fluxo de interações entre os serviços web (lógica de negócio e ordem de execução dos serviços, como um fluxo de trabalho). Normalmente

a orquestração necessita de uma entidade que coordene a composição, conhecida como motor de orquestração, enquanto a coreografia trabalha em como cada serviço deve se comportar de forma a colaborar harmoniosamente com os demais, definindo os protocolos para as trocas de mensagens entre os mesmos. Qualquer solução intermediária a estes paradigmas é possível , uma vez que a coreografia e a orquestração são formas extremas de realizar composições.

3.2.1 Linguagens para composição

Um serviço web publicado comumente apresenta apenas uma lista das declarações de mensagens disponíveis (operações) e seus tipos, na forma de um documento WSDL, não apresentando informações a respeito das seqüências de mensagens que o serviço deve enviar e receber, nem os detalhes de sua implementação. Na prática, é interessante conhecer não apenas a interface de cada serviço, mas outras informações de comportamento do mesmo. Para descrever o comportamento, bem como as seqüências dos serviços web, é necessário utilizar outras linguagens para compor os serviços [BULTAN et al., 2004; DER, 2003] tais como BPEL4WS [ANDREWS et al., 2003], WSCI [W3C, 2002], WSTL [PIRES et al., 2002] e PEWS [BA et al., 2005], as quais serão descritas a seguir.

3.2.1.1 BPEL4WS

BPEL4WS (*Business Process Execution Language for Web Services*) também conhecida como BPEL foi desenvolvida através da combinação das linguagens XLANG [THATTE, 2001] da Microsoft e WSFL (*Web Services Flow Language*) [LEYMANN, 2001] da IBM.

BPEL é essencialmente uma camada sobre WSDL, baseada em XML, que compõe serviços web através da descrição da lógica de controle requerida para coordenar os processos de negócios [MALEK; MILANOVIC, 2004]. Um processo pode ser definido de duas maneiras [CURBERA et al., 2004]:

- **Executáveis**: Abordagem de orquestração que provê uma definição completa sobre o modelo do processo de como ele será interpretado;

- **Abstratos**: Abordagem de coreografia que descreve somente o comportamento do serviço, podendo ocultar informações não relevantes da interação para outros serviços.

Em BPEL, o resultado da composição é chamado de processo (*process*), que é composto por um conjunto de atividades (*activity*) de trocas de mensagens entre os serviços participantes (*partners*), através de uma interface WSDL.

Um documento BPEL é constituído por atividades divididas nas seguintes primitivas [KOCHUT; YI, 2004]:

- Básicas
  - `<invoke>`: chamada de uma operação;
  - `<receive>`: recepção de uma mensagem;
  - `<reply>`: envio de um dado de resposta;
  - `<assign>`: modificar/atribuir o valor de uma variável;
  - `<wait>`: pausar o processo;
  - `<terminate>`: parar o processo;
  - `<empty>`: executar uma operação nula (sincronização);
  - `<throw>`: levantar uma exceção para execução de uma tarefa;
- Estruturadas
  - `<sequence>`: seqüência de ações;
  - `<flow>`: execução de atividades em paralelo;
  - `<switch>`: apresentar condições para determinada execução;
  - `<pick>`: resposta a um evento de atividade;
  - `<while>`: operação de interação.

O seguinte exemplo mostra a composição de um serviço em BPEL4WS utilizando as operações providas no Exemplo 13.

Exemplo 14 *O seguinte fragmento mostra a composição de um serviço web, com o processo identificado de* PurchaseProcess. *É apresentada a declaração do parceiro* purchase *o qual é utilizado na execução das duas atividades em seqüência,* ReceiveOrder *e* SendBill.

```
<process name="PurchaseProcess"
         targetNamespace="http://acme.com/simplepurchase"
         xmlns="http://schemas.xmlsoap.org/ws/2002/07/business-
            process/"
         xmlns:tns="http://puchase.org/wsdl/purchase"
  <partners>
     <partner name="purchase"
                  serviceLinkType="tns:Warehouse"
                  myRole="purchase"/>
...
   </partners
...
  <sequence>
     <invoke     partner="purchase"
                 portType="tns:WarehousePortType"
                 operation="order">
     </invoke>
     <invoque    partner="purchase"
          portType="tns:WarehousePortType"
          operation="bill">
     </invoque>
  </sequence>
</process>
```

No Exemplo 14, o elemento **process** contém as definições de um processo em BPEL. Já o elemento **partners**, declara os "parceiros" (provedores de serviços web) que irão interagir no processo. Após a declaração dos parceiros, a primitiva estruturada **sequence** que define uma seqüência de duas ações, **order** e **bill** que são invocadas pela primitiva básica **invoque**, informando o parceiro, o **PortType** e a operação provida pelo parceiro.

3.2.1.2 WSCI

WSCI (*Web Services Choreography Interface*) é uma linguagem que utiliza o padrão XML para descrever a coreografia, ou seja, o fluxo das trocas de mensagens para colaboração de serviços web [BROGI et al., 2004]. Somente o comportamento visível é descrito no documento WSCI, não tratando da definição de processos executáveis como no BPEL

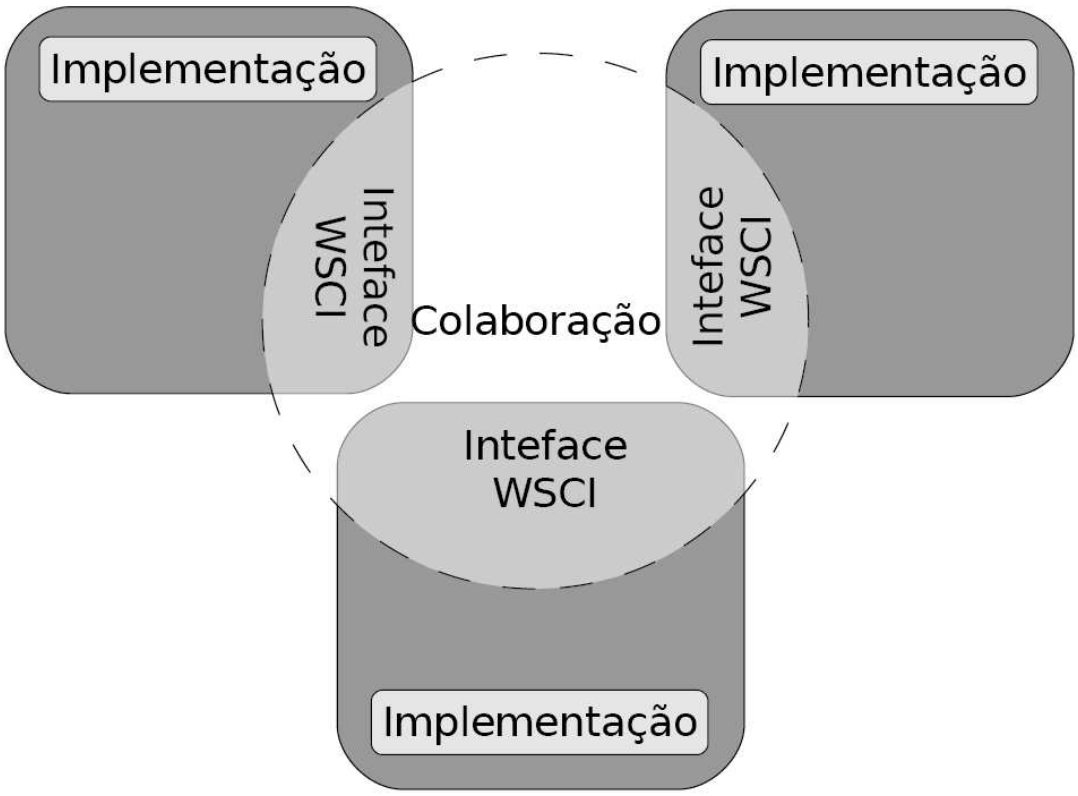


Figura 3.4: Web Services Choreography Interface (WSCI) [W3C, 2002]

e também não há um processo que gerencie globalmente as interações (ex.: motor de orquestração no BPEL). WSCI pode ser utilizada em conjunto com qualquer linguagem para descrição de serviços web, como WSDL, estendendo sua interface e relacionando várias operações através da troca de mensagens.

WSCI descreve somente a participação de um parceiro na troca de mensagens gerando uma interface composta por várias ações como ilustra a Figura 3.4. Cada ação está conectada a uma operação definida em um documento WSDL. As ações realizam as operações de um serviço web, tais como enviar e receber mensagens ou pausar a execução do serviço por um tempo determinado. Ações mais complexas podem ainda possuir características como execução paralela, condicional, iteração e tratamento de exceções. O conjunto de ações é chamado de processo.

O seguinte exemplo mostra a composição de um serviço em WSCI utilizando as operações providas no Exemplo 13.

Exemplo 15 *O seguinte fragmento mostra a composição de um serviço web com a in-*

terface identificada de PurchaseInterface, contendo o processo Purchase *que executa duas atividades em seqüência,* ReceiveOrder *e* SendBill.

```
<interface name="PurchaseInterface">
  <process name="Purchase">
    <sequence>
      <action name="ReceiveOrder"
            operation="tns:WarehousePortType/order">
      </action>
      <action name="SendBill"
            operation="tns:WarehousePortType/bill">
      </action>
    </sequence>
  </process>
</interface>
```

O Exemplo 15 define uma interface de serviço web (composição) através do elemento **interface**, que contém o conjunto de ações definido pelo elemento **process**. Este conjunto de ações é uma seqüência de execução de duas operações (**order** e **bill**) definida pela primitiva **sequence**. Cada operação é definida pelo elemento **action** contendo o atributo **name** e principalmente, o atributo **operation** que define o **portType** e a operação a ser executada.

3.2.1.3 PEWS

PEWS (*Predicate path-Expression for Web Services*) é uma linguagem utilizada para especificar a ordem em que as operações de serviços web são executadas. A linguagem pode ser utilizada não apenas na especificação de serviços web simples ou compostos, mas também como uma forma sintática na qual é possível compreender as propriedades dos serviços web [BA et al., 2005].

Path-Expressions são construções utilizadas para restringir as seqüências permitidas das operações em um objeto [CAMPBELL; HABERMANN, 1974]. Como exemplo, suponha que existam as operações x, y e z. A *path expression* $x^*.(y||z)$ expressa que a execução paralela das operações y e z deve ser precedida por zero ou mais execuções de x. Já *Predicate Path Expressions* são extensões das *Path-Expressions* com a adição de predicados e variáveis. [ANDLER, 1979].

Em PEWS, as *Predicate Path Expressions* são utilizadas para descrever o comportamento das interfaces do serviço web composto. Uma *Path-Expression* combina os nomes das operações do serviço web através dos operadores de seqüência (.), escolha não determinística (|), paralelismo (||), repetição (*), repetição paralela (...) e prefixo de predicado ([...]...).

A gramática da linguagem PEWS é apresentada na Tabela 3.1, sendo uma adaptação da gramática apresentada em [BA et al., 2005], já desprovida de recursões a esquerda (requisito da ferramenta utilizada na construção do compilador) e contendo as definições das operações aritméticas e booleanas e declaração dos *namespaces*, operações e variáveis. Os símbolos terminais estão com os respectivos lexemas entre parênteses, exceto:

- O terminal "int_constant" que aceita números decimais, octais, hexadecimais ou binários;
- O terminal "string_constant" que aceita constantes do tipo *string* como "abcd bcda";
- O terminal "ident" que trabalha com identificadores que iniciam com uma letra seguido de zero ou mais letras ou dígitos.

Um programa PEWS é definido como uma seqüência de *namespaces* ns, seguida de uma seqüência de declaração de operações alias, uma seqüência de definições def e finalizada com uma *path expression*.

Cada *namespace* declara onde estarão disponibilizados os documentos WSDL que descrevem as operações, assim podendo utilizar diferentes provedores em uma única composição. Já as declarações de operações, definem os "apelidos" das operações que serão usadas, informando a origem das operações (*portType* e *namespace*). Esta característica permite que operações com nomes iguais em diferentes provedores possam ser usadas na mesma composição, apenas definindo apelidos diferentes. Cada definição declara variáveis inteiras que serão usadas em predicados. O valor das variáveis pode ser definido a partir de números inteiros ou ser obtido pela avaliação de expressões aritméticas envolvendo contadores pré-definidos e bibliotecas de funções.

Os contadores são compostos pelos componentes *val* e *time*. O componente *val* representa um contador, enquanto que o componente *time* indica o momento em que o contador foi modificado pela última vez em milisegundos. Para cada operação *opname* de um serviço web, é possível o uso de um dos seguintes contadores:

act(opname): O componente *val* descreve o número de vezes que a operação *opname* começou a executar. O componente *time* indica o momento da última execução da operação.

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

req(opname): O componente *val* descreve o número de vezes que a operação *opname* foi requisitada. O componente *time* indica o momento da última requisição da operação (esta última não existente na versão original de PEWS proposta em [BA et al., 2005]).

Em PEWS também foram propostas operação pré-definidas (chamadas de funções) como `abortOperation` que segundo [BA et al., 2005] aborta todo o processo, e durante a elaboração deste trabalho, foram propostas novas operações como `escape`, que aborta apenas uma ramificação (para uso com paralelismo por exemplo) e `nop` que seria uma operação "nula", ou seja, que não executa operação alguma (não está ligada e nenhum `alias`).

O Exemplo 16 mostra a composição de um serviço web em PEWS utilizando as operações providas no Exemplo 13.

Exemplo 16 *O seguinte fragmento mostra a composição de um serviço web contendo duas atividades em seqüência,* order *e* bill *sendo executadas 0 ou mais vezes.*

```
ns warehouse = "http:\\url_of_wsdl_file.wsdl"
alias order = WarehousePortType/order in warehouse
alias bill = WarehousePortType/bill in warehouse
(order.bill)*
```

O Exemplo 16 define um *namespace* chamado **warehouse** indicando a origem do documento WSDL. Em seguida define as operações **order** e **bill**, informando seus respectivos *portType*, operação e *namespace*. Por fim é descrita a *path expression*.

O modelo formal definido pelas *path expressions* torna PEWS uma linguagem que descreve o comportamento de serviços web de forma abstrata. Mas para se detalhar uma composição no nível de interface, é necessária uma versão da linguagem PEWS que faça uma representação concreta e padronizada. Com este intuito, criou-se uma versão XML de PEWS, chamada XPEWS. XPEWS é uma linguagem que faz uma descrição detalhada das interfaces de um documento WSDL com base no comportamento definido na forma abstrata.

O elemento raiz em um documento XPEWS é o `<envelope>`. Ele pode conter as definições dos possíveis comportamentos de um serviço web e possui também a declaração do *XML Schema* que define a estrutura do documento. O elemento `<behaviour>` especifica o modo em que cada cliente pode interagir com o serviço web. O elemento `<behaviour>` é composto por dois atributos:

- `name`: identifica a composição;
- `xmlns`: declara um *namespace* que referencia o documento WSDL.

O elemento `operations` declara as operações que serão usadas na composição. Este elemento possui o atributo `name`, o qual declara um nome para a operação, que será usado na composição, o atributo `portType` informa qual *portType* contém a operação e o atributo `refersTo` associa a operação a um *namespace* declarado no elemento `behavior`.

Exemplo 17 mostra a tradução do Exemplo 16 em PEWS, para a composição de serviço web em XPEWS utilizando as operações providas no Exemplo 13.

Exemplo 17 *O seguinte fragmento mostra a composição do serviço web identificado de* Purchase *contendo a execução de duas operações em seqüência,* order *e* bill.

```
<envelope xmlns="http://aquarius.inf.ufpr.br"
          xsi:schemaLocation="http://aquarius.inf.ufpr.br pews.xsd">
    <behaviour name="Purchase"
```

```
            xmlns:warehouse="http:\\url_of_wsdl_file.wsdl">
        <operations>
            <operation name="order" portType="WarehousePortType"
                refersTo="warehouse:order"/>
            <operation name="bill" portType="WarehousePortType"
                refersTo="warehouse:bill"/>
        </operations>
        <pathExp>
            <star>
                <seq>
                    <operation name="order"/>
                    <operation name="bill"/>
                </seq>
            </star>
        </pathExp>
    </behaviour>
</envelope>
```

Mais exemplos de uso da linguagens de composição PEWS serão apresentados no Capítulo 5.

program	::=	(ns)+ (alias)+ (def)* path_expr “EOF”
ns	::=	“ns” namespace “=” file
alias	::=	“alias” opname “=” portType “/” operation “in” namespace
portType	::=	“ident”
operation	::=	“ident”
namespace	::=	(“ident”)
file	::=	“string_constant”
def	::=	“def” var “=” arith_expr
var	::=	“ident”
pred_expr	::=	pred_term (pred_expr2)?
pred_expr2	::=	“or” pred_term
pred_term	::=	pred_factor (pred_term2)?
pred_term2	::=	“and” pred_factor
pred_factor	::=	(“not”)? bool_expr
bool_expr	::=	“true”
	|	“false”
	|	arith_expr (“<” | “>” | “<=” | “>=” | “==” | “! =”) arith_expr
	|	“(” pred_expr “)”
path_expr	::=	parallel (“||” parallel)*
parallel	::=	choice (“|” choice)*
choice	::=	sequence (“.” sequence)*
sequence	::=	“{” path_expr “}”
	|	unarypath
unarypath	::=	path ((“*” | “+”))?
path	::=	opname
	|	“[” pred_expr “]” path
	|	“(” path_expr “)”
	|	“abortOperation”
	|	“escape”
arith_expr	::=	term (arith_expr2)?
arith_expr2	::=	“+” term
	|	“−” term
term	::=	unaryexpr (term2)?
term2	::=	“*” unaryexpr
	|	“/” unaryexpr
unaryexpr	::=	(“−”)? factor
factor	::=	“now” “(” “)”
	|	“act” “(” opname “)” “.” “val”
	|	“act” “(” opname “)” “.” “time”
	|	“term” “(” opname “)” “.” “val”
	|	“term” “(” opname “)” “.” “time”
	|	“req” “(” opname “)” “.” “val”
	|	“req” “(” opname “)” “.” “time”
	|	var
	|	“int_constant”
	|	“(” arith_expr “)”
opname	::=	(“ident”)

Tabela 3.1: Gramática da linguagem de composição PEWS

Capítulo 4

Construção do *front-end* para a linguagem de composição PEWS

4.1 Plataforma Eclipse

Eclipse é um ambiente multi-plataforma de código aberto [IBM CORPORATION, 2001] para o desenvolvimento e utilização de ferramentas de software. O Eclipse foi construído visando o desenvolvimento de aplicações em todos os domínios da computação; Sendo assim, ele é uma ferramenta que agrega funcionalidades, permitindo que o processo produtivo de software possa ser feito em um mesmo ambiente.

Uma das características principais da arquitetura do Eclipse é sua capacidade de extensão, uma vez que o *runtime* da plataforma é responsável pela inicialização da mesma bem como descobrir, registrar e carregar seus componentes, também conhecidos como *plug-ins*, responsáveis pelas funcionalidades apresentadas ao usuário [IBM CORPORATION, 2003].

Este mecanismo de extensão permite que a plataforma possa servir como ambiente de desenvolvimento a praticamente qualquer linguagem ou paradigma, sendo necessário o uso dos *plug-ins* que agregam a funcionalidade desejada [CLAYBERG; RUBEL, 2004].

Como exemplo desta capacidade de extensão, podemos citar *plug-ins* como **TeXlipse**[1]

[1] http://texlipse.sourceforge.net/

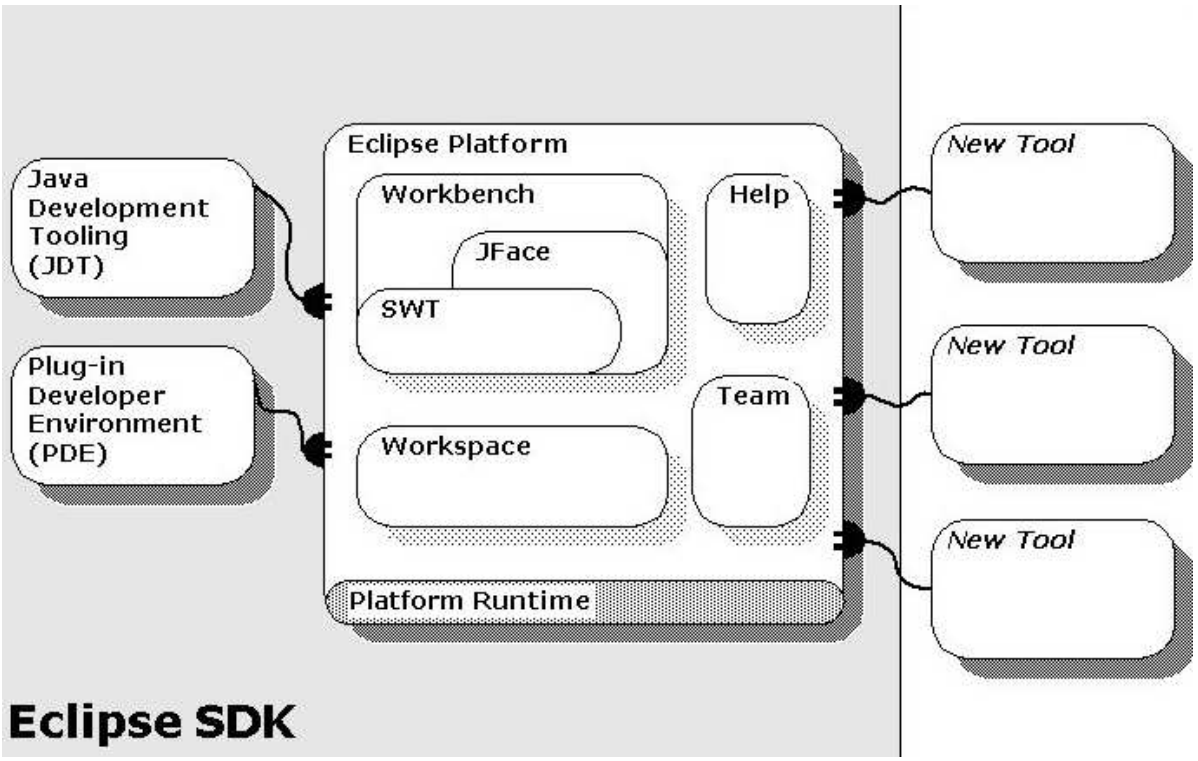


Figura 4.1: Arquitetura do Eclipse [IBM CORPORATION, 2004]

que provê suporte para LaTeX[2] (para produção de textos técnicos e científicos) e **JavaCC Eclipse Plug-in**[3] que provê suporte para JavaCC[4] (para criação de *parsers* em linguagem Java), permitindo que todo o processo de produção, desde a escrita da dissertação até a produção do protótipo, fosse realizado em um único ambiente de trabalho.

Os *plug-ins* que são carregados pela ferramenta são desenvolvidos na linguagem Java, e consistem de um arquivo de manifesto (que contém as informações sobre o *plug-in* como versão, provedor e classes que compõem o mesmo) e das classes Java reunidas em um ou mais arquivos JAR [BOLOUR, 2003]. A nova função que o *plug-in* implementa pode ser apresentada na forma de bibliotecas de código (classes Java com API pública), extensões de plataforma a outros *plug-ins* ou até mesmo documentação. Os *plug-ins* podem também definir seus pontos de extensão, que são locais bem definidos em que outros *plug-ins* podem agregar novas funcionalidades e características não previstas no mesmo [BOLOUR, 2003].

A Figura 4.1 ilustra a arquitetura básica do Eclipse SDK, que possui os seguintes

[2]http://www.latex-project.org/
[3]http://perso.wanadoo.fr/eclipse_javacc/
[4]https://javacc.dev.java.net/

componentes, além do *runtime* e dos *plug-ins* [IBM CORPORATION, 2004]:

- **Workspace**: responsável pelo gerenciamento dos recursos disponíveis tanto para os *plug-ins* da ferramenta quanto para aqueles que são adicionados pelo usuário. Ele implementa bibliotecas que são usadas por outros *plug-ins*, visando a gerência e modificação de projetos, pastas e arquivos. Os pontos de extensão do **workspace** permitem que os *plug-ins* definam seus próprios tipos de recursos, ou seja, como eles tratarão os arquivos e outros itens dos projetos.

- **Workbench**: responsável pela interface com o usuário, definindo os pontos de extensão que permitem que outros *plug-ins* contribuam com as ações da barra de ferramentas e do menu, operações de arrastar e soltar, diálogo, assistentes, exibições e editores personalizados.

- Bibliotecas **SWT** (*Standard Widget Toolkit*) e **JFace**: definem estruturas úteis para o desenvolvimento da interface com o usuário. Estas estruturas foram utilizadas para desenvolver o próprio workbench. A utilização dessas estruturas assegura que os *plug-ins* tenham aparência e comportamento comum, e um nível consistente de integração com o **workbench**.

- *Plug-ins* **JDT** (*Java Development Tools*): estendem o *workbench* da plataforma, fornecendo recursos para edição, visualização, compilação, depuração e execução do código Java.

- *Plug-ins* **PDE** (*Plug-in Development Environment*): fornecem ferramentas que automatizam a criação, manipulação, depuração e implementação de *plug-ins* através de uma grande quantidade de assistentes (*wizards*) e bibliotecas, o que auxilia na produção de novas ferramentas (*plug-ins*), ou adição de novas funcionalidades a *plug-ins* existentes através de seus pontos de extensão [GLOZIC; MELHEM, 2003].

- **Help**: responsável pela integração de ajuda pelo Eclipse entre o documento e o servidor Web (interno). Ele define pontos de extensão que os *plug-ins* podem utilizar

para contribuir com ajuda ou outro tipo de documentação de *plug-in*, como manuais navegáveis.

- *Plug-ins* **Team**: permitem que *plug-ins* definam e registrem implementações para programação em equipe, acesso a repositórios de código e controle de versão (CVS).
- *Plug-ins* **Debug**: permitem que outros *plug-ins* implementem depuradores de programas específicos da linguagem que darão suporte.

4.1.1 Escolha da plataforma Eclipse

Após análise técnica da plataforma Eclipse, algumas características foram decisivas para sua escolha como ambiente de desenvolvimento e utilização do *front-end* para a linguagem de composição PEWS na forma de um *plug-in*, das quais podemos citar:

- grande capacidade de integração com outras ferramentas indispensáveis para o desenvolvimento de serviços web, tais como editores XML, WSDL e Java;
- ausência do custo de aquisição da plataforma base e de outras ferramentas utilizadas para a produção e utilização do *plug-in*;
- código fonte aberto, permitindo extensão e alteração da plataforma para adequar-se às necessidades almejadas;
- multi-plataforma, uma vez que o Eclipse foi desenvolvido em Java, atingindo assim um número maior de usuários que poderão utilizar a ferramenta (os testes do *plug-in* foram realizados em Windows e GNU/Linux, ambos sobre arquitetura x86);
- simplicidade no desenvolvimento da parte estética da ferramenta, como por exemplo sintaxe colorida e marcação de erros, podendo assim dedicar maior tempo para detalhes da implementação da linguagem;
- possibilidade de execução automática do compilador durante a edição;
- forte adesão e apoio do mercado quanto a utilização da plataforma Eclipse.

4.2 O *plug-in* PEWS Editor

Como citado no seção 4.1, a plataforma Eclipse pode ser estendida para prover novas funcionalidade através de novos *plug-ins*. Estes novos *plug-ins* podem utilizar pontos de extensão de *plug-ins* pré-existente na plataforma, agregando novas funcionalidades e características, como o suporte a linguagem de composição PEWS apresentada na seção 3.2.1.3. Para a construção do *plug-in* que será utilizado como *front-end* da linguagem PEWS, foram utilizados os seguintes pontos de extensão:

- `org.eclipse.ui.editors` provê suporte à adição de novos editores ao *workbench*, ou à adicionar novas características a editores existentes [IBM CORPORATION, 2004];
- `org.eclipse.ui.popupMenus` provê suporte à adição de menus de contexto e ações aos documentos suportados pelo novo *plug-in* [IBM CORPORATION, 2004];
- `org.eclipse.core.resources.builders` provê suporte à utilização de construtores (ou um compilador no caso) ligados a uma natureza [ARTHORNE, 2003];
- `org.eclipse.core.resources.natures` provê suporte à adição de naturezas aos projetos. A natureza é responsável por definir o comportamento que o *plug-in* deverá ter com os documentos pertencentes a um projeto [ARTHORNE, 2003];
- `org.eclipse.core.resources.markers` provê suporte à adição de marcadores ao editor (para exibição de erros e demais informações necessárias) [IBM CORPORATION, 2004];
- `org.eclipse.ui.newWizards` provê suporte à adição de assistentes para a criação de novos projetos e arquivos [IBM CORPORATION, 2004].

A implementação do *plug-in* se dá a partir da classe núcleo do *plug-in* `PEWSPlugin`, que somada ao arquivo de manifesto `plugin.xml`, permitem a interação do novo *plug-in* com a plataforma Eclipse.

Os demais componentes que constituem o *plug-in* serão descritos a seguir.

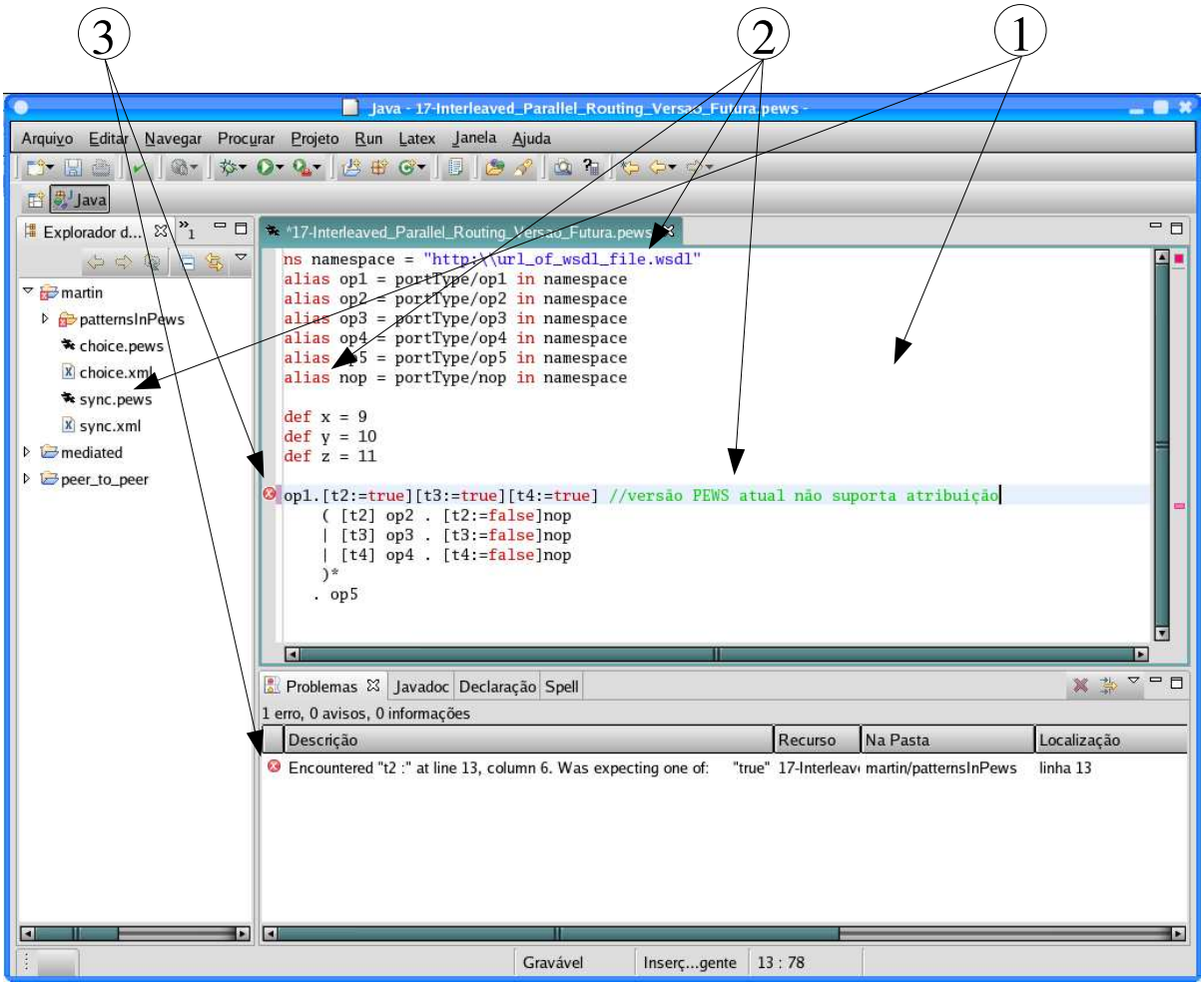


Figura 4.2: *Screenshot* do Eclipse com o *plug-in* PEWS Editor ativo

4.2.1 Editor

Principal artefato no nível de usuário, uma vez que é nele que se dará a escrita das composições em linguagem PEWS. Ele apresenta recursos como sintaxe colorida para diferenciar palavras reservadas da linguagem, *strings* e comentários adicionados à composição, como pode ser visto no item 2 da Figura 4.2. Também é nele que se apresenta a marcação de erros encontrados durante a análise da composição, para facilitar a localização pelo usuário (item 3 da Figura 4.2). As escolhas de cores, disposição de menus, mensagens de erros e demais pontos de interação com o usuário foram desenvolvidas usando engenharia semiótica [SOUZA, 2005].

4.2.2 Menu de contexto

Foi criado um menu de contexto para as composições PEWS. Ele é exibido ao clicar com o botão direito do mouse sobre uma composição PEWS dentro de um projeto (Explorador de pacotes) ou sobre a área do editor, desde que o mesmo esteja editando uma composição PEWS. Estes pontos estão indicados pelo item 1 da Figura 4.2.

Este menu de contexto chamado de "Generate XPEWS" executa o compilador que verifica a composição em questão, e caso a mesma contenha erros, é apresentada uma caixa de diálogo informando o erro. Caso contrário é criado um documento XML contendo o código XPEWS referente a composição PEWS verificada.

4.2.3 Construtor

Os projetos que contêm composições PEWS, possuem um construtor que é executado a cada vez que uma composição é alterada e salva. Este construtor, por sua vez, executa o compilador a procura de erros. Caso eles existam, são informados no editor e no visualizador "Problemas" como pode ser observado no item 3 da Figura 4.2.

4.2.4 Natureza

A natureza faz a ligação do construtor a um determinado projeto. Esta ligação ocorre quando é criada uma nova composição PEWS via assistente.

A composição PEWS é um arquivo com extensão ".pews" e pode ser criada em qualquer projeto pré-existente. Esta escolha é motivada pelo fato de que composições de serviços web podem ser utilizadas por projetos Java, projetos de sites dinâmicos ou projetos simples, ficando a critério do usuário.

4.2.5 Assistente

O assistente é responsável por auxiliar na criação de uma nova composição PEWS, trazendo uma estrutura mínima funcional de código como exemplo e fazer a ligação da natureza com o projeto que irá conter a composição.

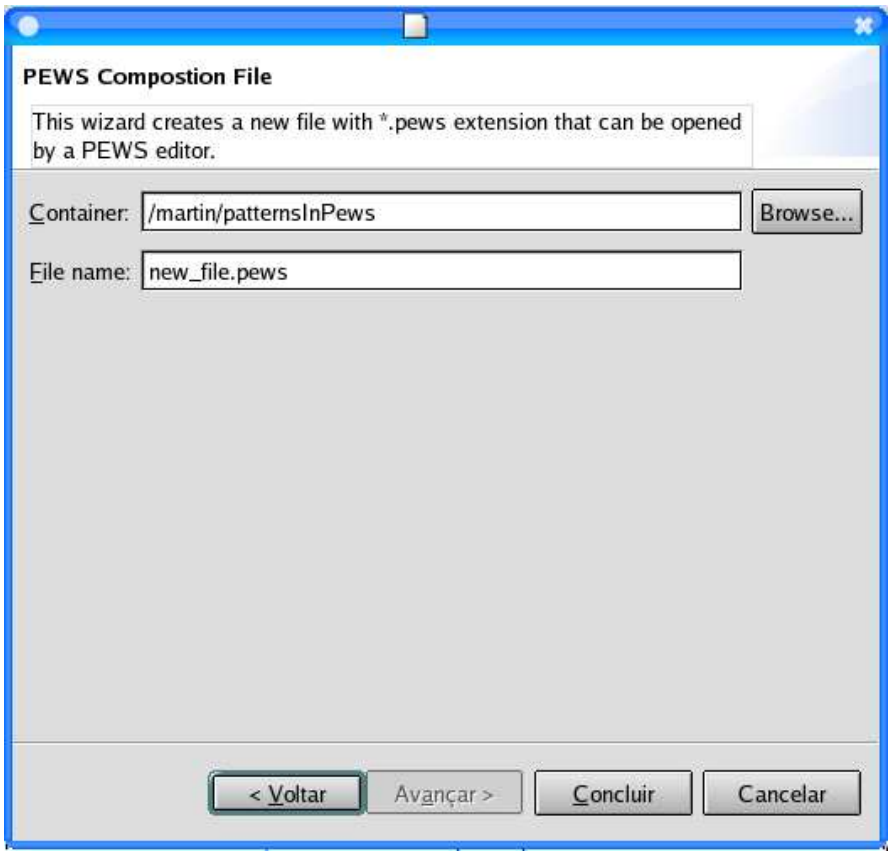


Figura 4.3: Assistente para criação de uma nova composição PEWS

O assistente é chamado via o menu "Arquivo → Novo → Outros...". Dentro do grupo "PEWS" existe a opção "PEWS Composition" que exibe o assistente como pode ser visto na Figura 4.3. Neste assistente é possível definir o projeto que irá conter a composição, a possível pasta dentro do projeto e o nome do arquivo que contém a composição.

4.2.6 Compilador

O compilador é a parte mais importante do *plug-in*, responsável pela analise léxica, sintática, semântica e por fim, tradução de código.

Para a construção do compilador, foi utilizada a ferramenta JavaCC [SUN MICROSYSTEMS, INC, 2005]. Este programa é utilizado para gerar a base para compiladores, mais especificamente o analisador léxico e sintático [DELAMARO, 2004]. Ele utiliza como entrada um arquivo (extensão ".jj"), contendo uma gramática, como por exemplo a gramática da linguagem de composição PEWS apresentada na Tabela 3.1 acrescida do conjunto de símbolos terminais, e a transforma em um programa Java com a capacidade de analisar

determinado arquivo e informar se o mesmo satisfaz a regras especificadas pela gramática.

Este programa gerado realiza análise sintática descendente (*top-down*) ou seja, constrói a árvore de derivação a partir da raiz em direção às folhas (símbolos terminais). Esta característica de árvore a partir da raiz se mostrou bastante útil. Uma vez que as construções para o JavaCC suportam operações semânticas, pode-se gerar o código XPEWS durante a própria análise, criando uma árvore com DOM [W3C, 2000]. Para isso foi utilizado o conceito de tradução dirigida pela sintaxe [AHO et al., 1986].

Através de operações semânticas, foi definida também uma tabela de símbolos, que recebe as declarações de *namespaces*, operações e variáveis que serão utilizadas na composição PEWS.

Quando é encontrado algum erro léxico, sintático ou semântico, é gerada uma exceção Java, que é tratada pelo *plug-in*, o qual gera uma marcação de erro no Editor e uma descrição do mesmo no visualizador "Problemas", como pode ser observado na Figura 4.2.

A gramática apresentada na Tabela 3.1 é uma versão adaptada da tabela gerada pelo utilitário jjdoc que lê uma especificação JavaCC e gera um arquivo HTML com toda a gramática da linguagem na notação E-BNF. O jjdoc é utilizado para documentação e verificação da gramática descrita no arquivo JavaCC.

Capítulo 5

Estudo de caso: Padrões de processo de *workflow* em PEWS

A composição de serviços web e os padrões de processo de *workflow* estão relacionados no sentido que ambos estão focados na gerência do fluxo de execução de processos. Neste capítulo, consideramos os 20 padrões de processo de *workflow* apresentados em [AALST et al., 2003a], discutindo como estes padrões podem ser implementados em PEWS.

Este estudo de caso serve como base para verificar a expressividade da linguagem PEWS, bem como apresentar o uso dos novos operadores e funções adicionados a linguagem durante a elaboração deste projeto. Este estudo também auxiliará no teste do *front-end*, verificando sua usabilidade e se o mesmo faz a correta análise do código PEWS e por fim a correta tradução para código XPEWS.

Nas seções seguintes serão apresentados alguns exemplos de programas em PEWS descrevendo sua estrutura a cada exemplo.

5.1 Padrões de controle básico

Nesta seção são apresentados padrões que contemplam aspectos básicos no controle de processos de *worflow*.

WP1 Sequence: Uma operação só é executada depois de completada a execução da operação anterior no mesmo processo.

Implementação do Padrão WP1: O operador PEWS "." define uma seqüência onde inicialmente é executada a primeira operação encontrada na seqüência, e ao término da mesma, é executado a próxima operação encontrada posterior ao operador PEWS. A execução continua até executar a última operação encontrada em uma seqüência.

Exemplo 18 *A operação "op2" é executada após o término da execução da operação "op1".*

```
ns namespace = "http:\\url_of_wsdl_file.wsdl"

alias op1 = portType/opA in namespace
alias op2 = portType/opB in namespace

op1 . op2
```

WP2 Parallel split: Em um determinado ponto no programa, múltiplos fluxos de operação são criados. Estes fluxos de operações são executados em diferentes *threads*, assim permitindo que as operações sejam executadas simultaneamente (paralelamente) ou fora de uma ordem cronológica.

Implementação do padrão WP2: O operador de paralelismo PEWS "||" divide as operações em *threads* independentes, podendo executá-las de forma paralela ou sem uma ordem pré-definida.

Exemplo 19 *As operações "op1", "op2" e "op3" são executadas em diferentes threads, o que permite paralelismo.*

```
ns namespace = "http:\\url_of_wsdl_file.wsdl"
alias op1 = portType/opA in namespace
alias op2 = portType/opB in namespace
alias op3 = portType/opC in namespace

op1 || op2 || op3
```

WP3 Synchronization: Um ponto no programa onde múltiplos fluxos de execução (*threads*) convergem para um único fluxo, sincronizando assim as operações. A sincronização permite que o fluxo de execução só prossiga após o término de todas a *threads* paralelas que estão em execução.

Implementação do padrão WP3: A implementação assemelha-se com o **parallel split** tendo as operações paralelas delimitadas por parênteses, seguido pelo operador PEWS desejado, sendo este último executado apenas após a conclusão de todas operações em execução (*threads*).

Exemplo 20 *As operações "op1", "op2" e "op3" são executadas em diferentes threads e somente ao término de todas as anteriores, a "op4" será executada.*

```
ns namespace = "http:\\url_of_wsdl_file.wsdl"
alias op1 = portType/opA in namespace
alias op2 = portType/opB in namespace
alias op3 = portType/opC in namespace
alias op4 = portType/opD in namespace

(op1 || op2 || op3) . op4
```

É possível observar que somente as versões estruturadas dos padrões WP2 e WP3 são contemplados por PEWS. A semântica para fluxo de execução não estruturado para o padrão *parallel split* e *synchronization* pode ser obtida com o uso de construções mais elaboradas em PEWS (ou com a repetição do nome da operação, caso necessário). Um exemplo de fluxo de execução não estruturado é apresentado na Figura 5.1. Por exemplo, a operação D deve ser executada somente após a operação A, mas pode ser executada em paralelo com a operação B. A operação D deve ter sido executada antes que a operação C possa ser executada.

A ordem de execução das operações é definida em PEWS com o uso dos **predicados** e **contadores** apresentados na seção 3.2.1.3.

O comportamento do fluxo de execução apresentado na Figura 5.1 pode ser expresso em PEWS usando a composição descrita no Exemplo 21.

Exemplo 21 *Fragmento de código PEWS expressando o exemplo da Figura 5.1*

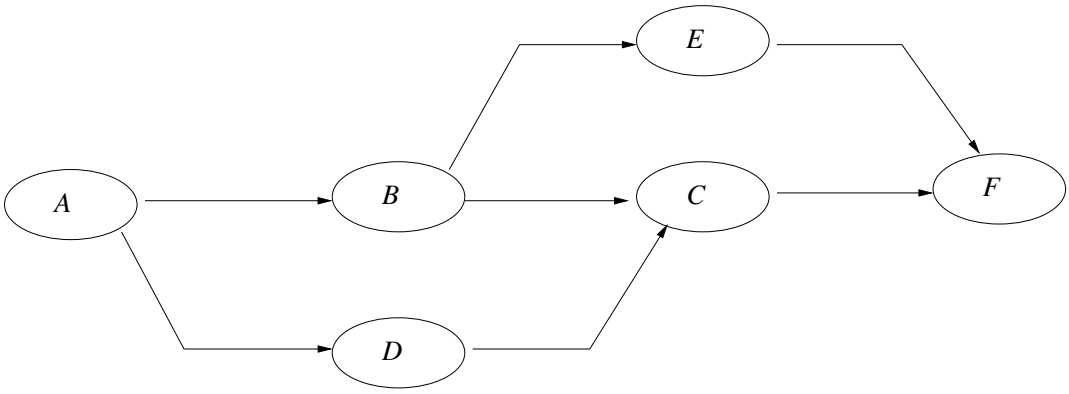


Figura 5.1: Fluxo de execução não estruturado [MUSICANTE; POTRICH, 2006]

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

É possível observar no fragmento apresentado no Exemplo 21 que a operação `A` é a primeira a ser executada. Depois que `A` é concluída, três ramificações em paralelo são executadas. A primeira ramificação representa a linha principal do fluxo de execução da Figura 5.1, onde a operação `B` pode ser executada em paralelo com a operação `D`. A execução das operações `C`, `F` e `E` não é iniciada até que os predicados que precedem estas operações se tornem verdadeiros. Por exemplo, a execução da operação `E` será adiada até que o número de conclusões da operação `B` seja maior do que o número de conclusões da operação `E`.

WP4 Exclusive choice: Um ponto no programa onde uma entre várias ramificações é escolhida, embasando-se em um critério de decisão.

WP5 Simple merge: Em um ponto no processo de *workflow*, duas ou mais ramificações se juntam em uma única, sem ocorrer uma sincronização. Este é um padrão simples de junção, assumindo que as operações das ramificações não tenham sido executadas em paralelo (caso este contemplado pelos padrões **multi-merge** e **discriminator**).
Implementações dos padrões WP4 e WP5: Estes padrões são implementados em

PEWS usando o operador para escolha não determinística (escolha exclusiva) "|". Este comportamento é descrito pela composição no Exemplo 22.

Exemplo 22 *WP4 Exclusive Choice e WP5 Simple Merge*

```
ns namespace = "http:\\url_of_wsdl_file.wsdl"
alias op1 = portType/opA in namespace
alias op2 = portType/opB in namespace
alias op3 = portType/opC in namespace
alias op4 = portType/opD in namespace

(op1 | op2 | op3) . op4
```

O padrão WP4 corresponde à uma escolha entre as operações apresentadas na última linha do Exemplo 22. Quando executado, o operador "|" selecionará uma ramificação que possui sua execução permitida. Uma ramificação tem sua execução permitida se a mesma tiver uma operação disponível para ser executada ou a operação desta ramificação é precedida de um predicado, e esse predicado é avaliado como verdadeiro [BA et al., 2005].

O padrão WP5 também é apresentada no Exemplo 22 uma vez que a operação `op4` será executada somente depois que a ramificação escolhida terminar sua execução.

5.2 Padrões de sincronização e ramificação avançada

Esta seção descreve padrões que são usados para sincronização e ramificação avançada de processos de *workflow*.

WP6 Multi-choice: Um ponto no processo de *workflow*, onde uma ou mais ramificações são escolhidas baseado um em critério de decisão, podendo ser executadas em paralelo.

WP7 Synchronizing Merge: Um ponto no processo de *workflow* onde múltiplas ramificações convergem em um único processo. Se mais de uma ramificação teve seu processo iniciado, a sincronização se torna necessária. Caso apenas uma ramificação teve seu processo iniciado, o processo de *workflow* deve seguir sem sincronização. Este padrão

assume que uma ramificação, após ter o seu processo executado, não pode ser executada novamente enquanto aguarda a sincronização com as outras ramificações [AALST et al., 2003b].

Implementações dos padrões WP6 e WP7: As composições PEWS podem conter os predicados, que podem agir como monitores em processos paralelos. Estes predicados são definidos com o valor dos contadores ou de outras variáveis declaradas. Os padrões WP6 e WP7 podem ser implementados em PEWS usando uma combinação dos operadores de "paralelismo" e "escolha não determinística". Um exemplo que contempla estes padrões é apresentado no Exemplo 23. Neste exemplo, o primeiro operador de "seqüência" define o comportamento previsto no padrão WP6, onde a execução da operação `op2` é condicionada ao valor verdade do predicado `[x < v1]`. O predicado será avaliado após o término da execução da operação `op1`. Caso este predicado for avaliado como falso, a primitiva de operação **nula**, representada pelo operador `nop` é executada (e o mesmo raciocínio se aplica à ramificação contendo a operação `op3`).

O comportamento do padrão WP7 é alcançado pela composição apresentada no Exemplo 23, desde que a operação `op4` seja executada uma vez, e somente uma das operações `op2` e `op3` tenha sido executada.

Exemplo 23 *WP6 Multi-choice e WP7 Synchronizing merge*

```
ns namespace = "http:\\url_of_wsdl_file.wsdl"
alias op1 = portType/opA in namespace
alias op2 = portType/opB in namespace
alias op3 = portType/opC in namespace
alias op4 = portType/opD in namespace

def x = ...uma expressão...
def y = ...outra expressão...
def v1 = ...um valor...
def v2 = ...outro valor...

op1.(   ([x < v1]op2 | [x >= v1]nop)
    || ([y > v2]op3 | [y <= v2]nop)
    )
   .
    ( [term(op2).time > term(op1).time
       or term(op3).time > term(op1).time] op4
```

```
    | [term(op2).time <= term(op1).time
       and term(op3).time <= term(op1).time] nop
    )
```

WP8 Multi-Merge: Este padrão consiste em um ponto no processo de *workflow* onde duas ou mais ramificações convergem em um único processo sem realizar sincronização. Se mais de uma ramificação estiver em execução, possivelmente em paralelo, a operação seguinte ao ponto onde as ramificações convergem, será executada para cada ramificação. Segundo [AALST et al., 2003b], BPEL4WS não possui base para implementar este padrão.

Implementação do padrão WP8: A composição apresentada no Exemplo 24 usa os contadores de PEWS para executar a operação `op3` quantas vezes se faça necessário.

Exemplo 24 *WP8 Multi-merge*

```
ns namespace = "http:\\url_of_wsdl_file.wsdl"
alias op1 = portType/opA in namespace
alias op2 = portType/opB in namespace
alias op3 = portType/opC in namespace

def x = ...uma expressão...
def y = ...outra expressão...
def v1 = ...um valor...
def v2 = ...outro valor...

(   ([x < v1]op1 | [x >= v1]nop)
|| ([y > v2]op2 | [y <= v2]nop)
|| ([term(op1).val + term(op2).val
     > 2*term(op3).val] op3)*
)
```

WP9 Discriminator: Este padrão descreve uma situação onde várias ramificações, possivelmente em paralelo, são ativadas. Em um determinado ponto no processo de *workflow*, o término de uma das ramificações é aguardado, antes de executar a operação subseqüente. A seguir é aguardado pelo término das ramificações restantes e as ignora (não executando novamente a operação subseqüente). Este padrão pode ser generalizado

em "aguardar k dentre n ramificações ativadas". De acordo com [AALST et al., 2003b], BPEL4WS é incapaz de implementar este padrão diretamente.

Implementação do padrão WP9: A composição apresentada no Exemplo 25 usa contadores PEWS para executar a operação `op3` após a execução da operação `op1` ou da operação `op2`.

Exemplo 25 *WP9 Discriminator*

```
ns namespace = "http:\\url_of_wsdl_file.wsdl"
alias op1 = portType/opA in namespace
alias op2 = portType/opB in namespace
alias op3 = portType/opC in namespace

def x = ...uma expressão...
def y = ...outra expressão...
def v1 = ...um valor...
def v2 = ...outro valor...

(   ([x < v1]op1 | [x >= v1]nop)
|| ([y > v2]op2 | [y <= v2]nop)
|| ([term(op1).val + term(op2).val
     >= 2*term(op3).val + 1] op3)
)
```

5.3 Padrões estruturais

Esta seção focaliza padrões de processos de *workflow* que apresentam estruturas de repetição ou desvios (semelhante a *while* e *goto* em linguagens de programação).

WP10 Arbitrary cycles: Em determinados pontos no processo de *workflow*, uma parte do processo (indiferente de sua complexidade) pode ter sua execução repetida, sem restrições quanto ao número e localização destes pontos (Semelhante ao comando `perform` da linguagem Cobol). Este padrão não pode ser implementado por BPEL4WS e PEWS devido a estas linguagens disporem somente de repetições estruturadas (provida pela operação "*" em PEWS).

WP11 Implicit Termination: Uma ramificação deve ser terminada quando não há mais nada a ser executado, ou seja, não há operações em execução no processo de *workflow* e não há outra operação disponível para ser executada.

Implementação do padrão WP11: Em PEWS, não há a necessidade de tornar explicito o término de uma operação.

5.4 Padrões para múltiplas instâncias

Os seguintes quatro padrões descrevem como criar múltiplas instâncias ou cópias de determinadas operações.

WP12 Multiple Instances without Synchronization: A partir de um ponto no processo de *workflow* uma operação pode criar várias instâncias de si, em diferentes *threads*. Cada uma destas instâncias é independente e não há necessidade de sincronizar com as outras operações em execução para seguir com o processo de *workflow*.

Implementação do padrão WP12: A implementação em PEWS se dá delimitando as operações que se deseja ter múltiplas instâncias, com o operador de "repetição paralela" PEWS "{...}".

WP13-WP15 Multiple Instances with Synchronization: Estes três padrões correspondem a pontos em um processo de *workflow* onde múltiplas instâncias de uma operação são iniciadas, em diferentes *threads*. Estas instâncias são posteriormente sincronizadas. No padrão WP13, o número de instâncias a serem iniciadas e sincronizadas já é conhecido quando a composição é escrita. No padrão WP14, o número de instâncias é conhecido em algum momento em tempo de execução, mas antes de iniciar a execução das instâncias. Já no padrão WP15 o número de instâncias a serem criadas não é de conhecimento prévio em momento algum do processo, sendo assim, novas instâncias são criadas sob demanda, até não necessitar mais de novas instâncias.

Implementações dos padrões WP13 a WP15: Combinações entre o operador de "repetição paralela" com a avaliação de predicados em PEWS podem ser usados para

implementar estes padrões. No caso do padrão WP13, a condição para se determinar o número de novas instâncias é definido de forma estática. O padrão WP14 é implementado através de uma variável que define o número de novas instâncias, variável esta que tem o seu valor alterado em tempo de execução. Já o padrão WP15 não possui uma condição que defina o número de instâncias (sua implementação é similar à WP12).

5.5 Padrões com base em estados

Esta seção descreve padrões que definem como o comportamento de um processo de *workflow* pode às vezes ser afetado por fatores externos ao processo.

WP16 Deferred Choice: Este padrão descreve um ponto no processo de *workflow* onde uma entre diversas ramificações é escolhida. O critério que será utilizado na escolha não está necessariamente definido quando este ponto for alcançado. Como apenas uma ramificação será escolhida, após a escolha, as demais são descartadas. Este padrão difere do WP4, uma vez que a escolha pode ser adiada até a ocorrência de algum evento.
Implementação do padrão WP16: A implementação deste padrão se assemelha ao padrão WP4 (apresentado no Exemplo 22), a única diferença deste caso é que os predicados que contenham as condições permitindo a execução, devem ser definidos em cada ramificação. De acordo com [BA et al., 2005], o processo irá aguardar até que uma das condições retorne como verdade, escolhendo assim a ramificação associada a ela.

WP17 Interleaved Parallel Routing: Neste padrão, um conjunto de operações é executado em uma ordem arbitrária. Esta ordem é decidida em tempo de execução, as operação são executadas apenas uma vez e não podem ser executadas ao mesmo tempo (não podem ser executadas em paralelo).
Implementação do padrão WP17: PEWS não é capaz de implementar este padrão diretamente. Isso pode ser simulado por uma estrutura de seleção, onde cada ramificação contém uma réplica das operações envolvidas no processo, ordenadas de forma arbitrária em cada ramificação. Outra maneira de implementar este padrão é através de um pro-

grama externo à composição, que defina a ordem de execução entre as diferentes operações disponíveis no conjunto.

Versões futuras de PEWS poderão implementar este padrão, através da introdução de operações com dados à linguagem.

WP18 Milestone: Este padrão descreve uma situação onde uma operação pode ser executada até alcançar um certo ponto ("marco"), ou satisfazer um pré-requisito. Um "marco" é um ponto no processo de *workflow* onde uma operação A terminou sua execução e uma atividade subseqüente B não foi iniciada ainda.

Implementação do padrão WP18: O Exemplo 26 demonstra este padrão. A execução das operações `op2` e `op3` é dependente de uma mensagem entrante. É possível observar que depois de requisitada a execução da operação `op2`, a operação `op3` não estará mais disponível para execução.

Exemplo 26 *WP18 Milestone*

```
ns namespace = "http:\\url_of_wsdl_file.wsdl"
alias op1 = portType/opA in namespace
alias op2 = portType/opB in namespace
alias op3 = portType/opC in namespace

( op1 . op2
|| ( [term(op1).val > req(op2).val] op3
   | [term(op1).val <= req(op2).val] nop
   )
)
```

5.6 Padrões de cancelamento

Este dois últimos padrões apresentam como podem ser canceladas operações ou um grupo de operações (uma ramificação por exemplo) em um processo de *workflow*.

WP19 Cancel Activity e WP20 Cancel Case: O padrão WP19 refere-se ao término de uma instância em execução. Já o padrão WP20 é definido como o término

de todo o processo de *workflow*. Respectivamente as operações pré-definidas `escape` e `abortOperation` permitem que PEWS de a sustentação necessária para estes padrões.

Capítulo 6

Conclusões

Esta dissertação traz contribuições ao trabalho desenvolvido por [BA et al., 2005] que deixou em aberto, entre outros, a implementação no nível abstrato. Esta implementação se dá através de uma ferramenta (*front-end*) que auxilie na composição utilizando a linguagem PEWS. A ferramenta deve também traduzir entre a sintaxe de PEWS para XPEWS (que corresponde ao nível de interface).

Em um primeiro momento, podemos citar as contribuições ligadas à especificação formal da sintaxe de PEWS, obtidas a partir da conclusão das regras não definidas na gramática apresentada por [BA et al., 2005] como operações booleanas, relacionais e aritméticas e definição de variáveis e operações (operações estas, oriundas de documentos WSDL). Também foram adicionadas funções como `escape` e `nop` e o contador `req`. A gramática também sofreu alterações na sua forma de ser expressa, devido à impossibilidade da ferramenta JavaCC de utilizar produções com recursão à esquerda (código fonte apresentado no Apêndice A). Esta implementação, tem como resultado o compilador que faz a análise léxica, sintática e semântica das composições PEWS e por fim, gera o código XPEWS.

Outra contribuição é o *front-end* desenvolvido na forma de um *plug-in* para a plataforma Eclipse, contendo além do editor, o compilador PEWS. O *plug-in* está disponível no endereço http://www.unoescxxe.edu.br/~edinardo/index.php?secao=pews junto com sua documentação e código fonte. A premissa deste *plug-in* é auxiliar na escrita de com-

posições PEWS, através de facilidades como:

- Assistente para criação de novas composições, trazendo uma estrutura mínima para a nova composição;
- Sintaxe colorida, facilitando a identificação de palavras reservadas;
- Análise automática do código da composição, realizada a cada vez que a composição é salva;
- Identificação dos erros encontrados a partir da análise do código, identificação esta, apresentada tanto no editor (diretamente na linha que contém o erro), quanto no visualizador de problemas, o que facilita a busca de erros até em outras composições PEWS (o visualizador informa os erros de composições que não estão abertas no momento, permitindo que com o duplo-clique, esta composição seja aberta, e o cursor seja posicionado na linha que contém o erro);
- Facilidade na geração de código XPEWS, através de um menu de contexto;
- Integração com outras ferramentas disponibilizadas pela plataforma Eclipse, uma vez que o *plug-in* permite realizar uma composição em qualquer projeto existente (Java por exemplo).

O desenvolvimento do *plug-in* foi uma das partes mais complexas no desenvolvimento do trabalho, devido principalmente à escassez de uma documentação clara e padronizada sobre o desenvolvimento de *plug-ins*. Uma grande parcela da documentação e exemplos encontrados, referiam-se a versões antigas do Eclipse (a base do nosso desenvolvimento foi a versão 3.1), já não contempladas na versão corrente.

Como última contribuição, podemos citar o estudo de caso apresentado no Capítulo 5 que demonstra a implementação em PEWS da maioria dos padrões para testes de processos de *workflow* definidos por [AALST et al., 2003a]. Com o estudo de caso é possível observar que PEWS é capaz de implementar todos os padrões propostos, exceto o padrão WP10 (*Arbitrary cycles*), devido a PEWS ser capaz de expressar apenas ciclos estruturados e o padrão WP17 (*Interleaved Parallel Routing*) de forma direta, sendo necessário

explicitar as combinações possíveis utilizar-se de implementações externas como por exemplo, o uso da linguagem Java .

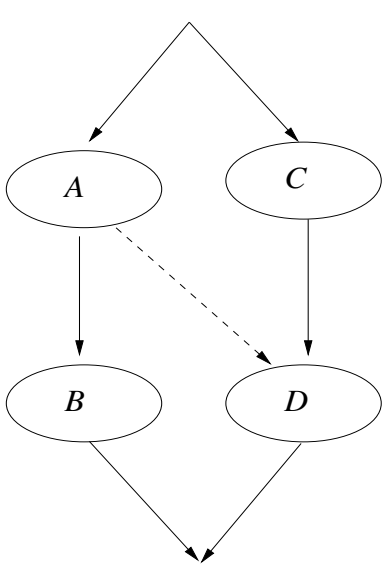
Padrão	**Produto/Padrão**					
	BPEL	XLANG	WSFL	BPML	WSCI	PEWS
Sequence	+	+	+	+	+	+
Parallel Split	+	+	+	+	+	+
Synchronization	+	+	+	+	+	+
Exclusive Choice	+	+	+	+	+	+
Simple Merge	+	+	+	+	+	+
Multi Choice	+	-	+	-	-	+
Synchronizing Merge	+	-	+	-	-	+
Multi-Merge	-	-	-	+/-	+/-	+
Discriminator	-	-	-	-	-	+
Arbitrary Cycles	-	-	-	-	-	-
Implicit Termination	+	-	+	+	+	+
MI without Synchronization	+	+	+	+	+	+
MI with a Priori Design Time Knowledge	+	+	+	+	+	+
MI with a Priori Runtime Knowledge	-	-	-	-	-	+
MI without a Priori Runtime Knowledge	-	-	-	-	-	+
Deferred Choice	+	+	-	+	+	+
Interleaved Parallel Routing	+/-	-	-	-	-	+/-
Milestone	-	-	-	-	-	+
Cancel Activity	+	+	+	+	+	+
Cancel Case	+	+	+	+	+	+

Tabela 6.1: Linguagens de composição e sua respectiva capacidade de implementação de cada padrão [AALST, 2003]

Mais linguagens de composição foram testadas por outros autores, utilizando estes padrões de teste. Os resultados obtidos de algumas destas linguagens são apresentados por [AALST, 2003; AALST et al., 2003b]. As cinco primeiras colunas de resultados apresentadas na Tabela 6.1 foram extraídas destes artigos. A última coluna da tabela corresponde ao nosso estudo. Cada linha da tabela representa um padrão para teste de processo de *workflow*. O símbolo "+" indica que a linguagem em questão é capaz de implementar o padrão. O símbolo "-" indica a impossibilidade da linguagem implementar o referido padrão. Já uma capacidade parcial de implementação é marcada com "+/-".

É possível observar que PEWS é capaz de implementar mais padrões do que as linguagens restantes. Esta é uma indicação da adequação das *path-expression* em expressar composições de serviços web.

Podemos afirmar que composições PEWS são, em geral, mais sucintas do que composições equivalentes em outras linguagens de composição de serviços web. Isto se deve não somente devido à existência de uma versão abstrata da linguagem, mas por causa do




```
<flow name="F">
 <links>
  <link name="L"/>
 </links>
  <sequence name="Sa">
   A
    <source linkName="L"/>
   B
  </sequence>
  <sequence name="Sb">
   C
   D
    <target linkName="L"/>
  </sequence>
</flow>
```

Figura 6.1: Processo de *workflow* não estruturado [AALST et al., 2003b].

uso das condições (definidas através de predicados) na execução de operações. Por exemplo, na Figura 6.1 é apresentado um diagrama de processo de *workflow*, onde a execução da operação `D` é iniciada somente após a conclusão da execução da operação `A` (indicada pela seta pontilhada). O código da composição em BPEL4WS que corresponde a este processo de *workflow* é apresentado na mesma figura.

Uma composição equivalente à apresentada na Figura 6.1 tem sua implementação em PEWS da forma apresentada pelo Exemplo 27, com sua respectiva tradução para a linguagem XPEWS apresentada no Exemplo 28

Exemplo 27 *O seguinte fragmento define que a operação* `D` *terá permissão de ser executada somente após que a operação* `A` *terminar sua execução.*

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

Exemplo 28 *O seguinte fragmento é a tradução da composição PEWS do Exemplo 27.*

```
<pathExp>
 <par>
  <seq>
   <operation name="A"/>
```

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

6.1 Trabalhos futuros

Algumas característica ficaram em aberto no desenvolvimento do *front-end*, as quais podemos citar:

- Verificação de tipos no compilador;
- Descoberta automática de operações provenientes de documentos WSDL;
- Ajuda on-line para o *front-end* tendo como conteúdo o uso da ferramenta e a descrição formal da linguagem;
- Modo de composição gráfica, podendo construir as composições, simplesmente arrastando as operações e operadores PEWS;
- Adicionar característica de robustez ao compilador, permitindo que o mesmo detecte todos os erros existentes na composição de uma única vez;
- Assistente de código, auxiliando na escrita da composição, exibindo as operações declaradas e palavras reservadas;
- Construção de um fragmento do *plug-in* contendo uma versão do mesmo em língua portuguesa.

Estas características foram detectadas ao longo do desenvolvimento do *front-end* e servem como sugestões para trabalhos futuros.

Ainda como trabalho futuro, seria interessante também a inserção de operações com dados nas próximas versões de PEWS, bem como substituir a entrada "portType/operation" na declaração das operações por uma expressão XPath e comparação com outras linguagens de composição como WSTL.

Referências Bibliográficas

AALST, W. M. P. V. D. et al. Workflow patterns. *Distrib. Parallel Databases*, Kluwer Academic Publishers, Hingham, MA, USA, v. 14, n. 1, p. 5–51, 2003. ISSN 0926-8782.

AALST, W. M. van der et al. *Analysis of Web Services Composition Languages: The Case of BPEL4WS*. 2003. Disponível em: <http://citeseer.ist.psu.edu/659670.html>.

AALST, W. van der. *Don't go with the flow: Web services composition standards exposed*. 2003. Disponível em: <http://citeseer.ist.psu.edu/vanderaalst03dont.html>.

ABITEBOUL, S. et al. Dynamic XML documents with distribution and replication. In: *Proc. of the ACM SIGMOD Conf.* [s.n.], 2003. Disponível em: <http://citeseer.ist.psu-.edu/abiteboul03dynamic.html>. Acesso em: 13 mai. 2005.

AGARWAL, M. *Synthesizing autonomic compositions in grid environment*. Dissertação (Mestrado) — Rutgers, The State University of New Jersey, 2003. Disponível em: <http://www.caip.rutgers.edu/TASSL/Thesis/manishag_thesis.pdf>. Acesso em: 04 mai. 2005.

AHO, A. V.; SETHI, R.; ULLMAN, J. D. *Compilers, Principles, Techniques, and Tools*. [S.l.]: Addison-Wesley, 1986.

ANDLER, S. Predicate path expressions. In: *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. [S.l.]: ACM Press, 1979. p. 226–236.

ANDREWS, T. et al. *Specification: Business Process Execution Language for Web Services Version 1.1*. 2003. Disponível em: <http://www-128.ibm.com/developerworks-/library/specification/ws-bpel/>. Acesso em: 11 mai. 2005.

ARTHORNE, J. *Project Builders and Natures*. 2003. Disponível em: <http://www-.eclipse.org/articles/Article-Builders/builders.html>. Acesso em: 27 jan. 2006.

BA, C.; FERRARI, M. H.; MUSICANTE, M. Building web services interfaces using predicate path expressions. In: BRASILIAN COMPUTER SCIENCE SOCIETY. *Proceedings of SBLP 2005. IX Brazilian Symposium on Programming Languages*. Recife - Brazil: University of Pernambuco, 2005. p. 147–160.

BENNETT, K. H. et al. Using web service technologies to create an information broker: An experience report. In: *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*. [S.l.]: IEEE Computer Society, 2004. p. 552–561.

BODOFF, D.; HUNG, P. C.; BEN-MENACHEM, M. Web metadata standards: Observations and prescriptions. *Software*, IEEE Computer Society Press, v. 22, n. 1, p. 78–85, 2005.

BOLOUR, A. Notes on the eclipse plug-in architecture. 3 jul. 2003. Disponível em: <http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture-.html>. Acesso em: 04 abr. 2005.

BORDEAUX, L.; SALAUN, G.; SCHAERF, M. Describing and reasoning on web services using process algebra. In: *Proceedings of the IEEE International Conference on Web Services*. [S.l.: s.n.], 2004. p. 43–50.

BROGI, A. et al. Formalizing web service choreographies. In: *Electronic Notes in Theoretical Computer Science*. [S.l.: s.n.], 2004. v. 105, p. 73–94.

BULTAN, T.; FU, X.; SU, J. Analysis of interacting bpel web services. In: *Proceedings of the 13th international conference on World Wide Web*. [S.l.]: ACM Press, 2004. p. 621–630.

CAMPBELL, R. H.; HABERMANN, A. N. The specification of process synchronization by path expressions. In: *Symposium on Operating Systems*. [S.l.: s.n.], 1974. p. 89–102.

CAO, Z. et al. Verification of web services using an enhanced uddi server. In: *Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003)*. [S.l.: s.n.], 2003. p. 131–138.

CARRERO, M. A.; MUSICANTE, M. A. *Um Sistema de tempo de Execução para a linguagem PEWS*. Dissertação (Mestrado) — Departamento de Informática, Universidade Federal do Paraná, 2006. Em andamento.

CLAYBERG, E.; RUBEL, D. *eclipse*: Building commercial-quality plug-ins. Boston, MA: Addison-Wesley, 2004. 821 p. (The Eclipse Series). ISBN 0-321-122847-2.

CUNHA, D. Web services, soap e aplicações web. 10 dez. 2002. Disponível em: <http://devedge-temp.mozilla.org/viewsource/2002/soap-overview/index_pt_br.html>. Acesso em: 11 mai. 2005.

CURBERA, F.; KHALAF, R.; LIU, S. From daml-s processes to bpel4ws. In: *Proceedings of the 14th International Workshop on Research Issues on Data Engineering: Web Services for e-Commerce and e-Government Applications*. [S.l.]: IEEE Computer Society, 2004. p. 77–84.

DELAMARO, M. E. *Como construir um compilador*: Utilizando ferramentas java. São Paulo, SP: Novatec, 2004. 308 p. ISBN 85-7522-055-1.

DER, W. V. *Web Service Composition Languages: Old Wine in New Bottles?* 2003. Disponível em: <http://citeseer.ist.psu.edu/635006.html>. Acesso em: 13 mai. 2005.

FOURER LEO LOPES, K. M. R. *LPFML: A W3C XML Schema for Linear Programming*. [S.l.], 2004.

GLOZIC, D.; MELHEM, W. Pde does plug-ins. set. 2003. Disponível em: <http://-/www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html>. Acesso em: 15 set. 2004.

GUNZER, H. Introduction to web services. Mar 2002. Disponível em: <http://-/thecoadletter.com/article/images/28818/webservices.pdf>. Acesso em: 04 mai. 2005.

HU, M. *Web Services Composition, Partition, and Quality of Service in Distributed System Integration and Re-engineering*. [S.l.], 5 mai. 2004.

IBM CORPORATION. Common public license - v 1.0. 2001. Disponível em: <http://www.eclipse.org/legal/cpl-v10.html>. Acesso em: 28 set. 2004.

IBM CORPORATION. Eclipse platform technical overview. fev. 2003. Disponível em: <http://www.eclipse.org/whitepapers/eclipse-overview.pdf>. Acesso em: 28 set. 2004.

IBM CORPORATION. *Plataform Plug-in Developer Guide*: Plataform architecture. [S.l.], 2004.

KEIDL, M.; KEMPER, A. *Towards Context-Aware Adaptable Web Services*. 2004. Disponível em: <http://citeseer.ist.psu.edu/keidl04towards.html>. Acesso em: 25 jan. 2006.

KOCHUT, K. J.; YI, X. *CPNet Model for BPEL4WS Workflow*. [S.l.], Nov. 2004.

LEYMANN, F. *Web services flow language*. 2001. Disponível em: <http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf>. Acesso em: 11 mai. 2005.

MALEK, M.; MILANOVIC, N. Current solutions for web service composition. *IEEE Internet Computing*, v. 8, n. 6, p. 51–59, Nov.–Dec. 2004.

MUSICANTE, M. A.; POTRICH, E. Expressing workflow patterns for web services: The case of pews. Submetido para SBLP. 2006.

OASIS. *UDDI V2.03 data structure specification*. 2002. Disponível em: <http://-/uddi.org/pubs/DataStructure-V2.03-Published-20020719.pdf>. Acesso em: 04 mai. 2005.

OASIS. *Introduction to UDDI: important Features and Functional Concepts*. 2004. UDDI Technical White Paper. Disponível em: <http://uddi.org/pubs/uddi-tech-wp.pdf>. Acesso em: 03 mai. 2005.

PIRES, P. F.; BENEVIDES, M. R. F.; MATTOSO, M. Building reliable web services compositions. In: *Web, Web-Services, and Database Systems*. [S.l.: s.n.], 2002. p. 59–72.

RAMANUJAN, A.; ROY, J. Xml schema language: taking xml to the next level. *IT Professional*, v. 3, n. 2, p. 37–40, Mar./Apr. 2001.

SANT'ANNA, M. O que é xml? 2005. Disponível em: <http://www.microsoft-.com/brasil/msdn/CincoEstrelas/downloads/Default.mspx>. Acesso em: 04 abr. 2005.

SOUZA, C. S. de. *The Semiotic Engineering of Human-Computer Interaction*. [S.l.]: The MIT Press, 2005. ISBN 0262042207.

SUN MICROSYSTEMS, INC. *Java Compiler Compiler™*. 2005. Disponível em: <https://javacc.dev.java.net/>. Acesso em: 28 abr. 2005.

SUN MICROSYSTEMS, INC. *Java API for XML processing*. 2006. Disponível em: <http://java.sun.com/xml/jaxp/>. Acesso em: 30 jan. 2006.

SYSTINET. Tutorial xml. 2000. Disponível em: <http://www.zvon.org/xxl-/XMLTutorial/General_or/book.html>. Acesso em: 04 abr. 2005.

THATTE, S. *XLANG, Web Services for Business Process Design*. [S.l.], 2001.

W3C. Document object model (dom) level 2 core specification. 13 nov. 2000. Disponível em: <http://www.w3.org/TR/DOM-Level-2-Core/>. Acesso em: 30 jan. 2006.

W3C. *Web Services Choreography Interface 1.0*. 2002. Disponível em: <http://www.w3-.org/TR/wsci>. Acesso em: 11 mai. 2005.

W3C. *SOAP Version 1.2 Part 1: Messaging Framework*. 2003. Disponível em: <http://www.w3.org/TR/2003/REC-soap12-part1-20030624>. Acesso em: 03 mai. 2005.

W3C. *SOAP Version 1.2 Part 2: Adjuncts*. 2003. Disponível em: <http://www.w3.org-/TR/2003/REC-soap12-part2-20030624>. Acesso em: 03 mai. 2005.

W3C. Web services architecture. 11 fev. 2004. Disponível em: <http://www.w3.org/TR-/ws-arch/>. Acesso em: 18 abr. 2006.

YERGEAU, F. Extensible markup language (xml) 1.0 (third edition). 4 fev. 2004. Disponível em: <http://www.w3.org/TR/2004/REC-xml-20040204/>. Acesso em: 30 abr. 2005.

Apêndice A

Código fonte JavaCC

```
/**
 * Compilador PEWS
 **/

options {
  STATIC = false;
  DEBUG_LOOKAHEAD = true;
  JAVA_UNICODE_ESCAPE = true;
  JDK_VERSION = "1.5";
}

PARSER_BEGIN(pews)
package br.ufpr.pews.parser;

import java.util.Hashtable;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class pews {
  private Hashtable symTab = new Hashtable();
  public static int lineParseError = 0;
  public static DocumentBuilderFactory dbfac = DocumentBuilderFactory
      .newInstance();
  public static DocumentBuilder docBuilder;
  public static Document doc;
  public static Element root;
  public static Element behaviour;
  public static Element operations;
```

```
  public static Element pathExp;

  public static Document createDomDocument() {
        try {
            docBuilder = dbfac.newDocumentBuilder();
            doc = docBuilder.newDocument();
            return doc;
        } catch (ParserConfigurationException e) {
          System.err.println(e);
        }
        return null;
    }
}

PARSER_END(pews)

TOKEN_MGR_DECLS :
{
int countLexError = 0;

public int foundLexError()
{
   return countLexError;
}

}


/* Espaços a serem desprezados no início de cada token */

SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}

/* Ignorar comentário */

SKIP :
{
  "//" : singlelinecomment
}
<singlelinecomment> SKIP:
{
  <["\n","\r"]> : DEFAULT
  | <~[]>
}
```

```
/* Palavras reservadas */

TOKEN :
{
  < NOW: "now" >
| < ACT: "act" >
| < TERM: "term" >
| < REQ: "req" >
| < ABORT: "abortOperation" >
| < VAL: "val" >
| < TIME: "time" >
| < DEF: "def" >
| < TRUE: "true" >
| < FALSE: "false" >
| < AND: "and" >
| < OR: "or" >
| < NOT: "not">
| < ALIAS: "alias">
| < IN: "in">
| < NS: "ns">
}

/* constantes */

TOKEN :
{
  < int_constant:(  // números decimais, octais, hexadecimais ou
      binários
                  (["0"-"9"] (["0"-"9"])* ) |
                  (["0"-"7"] (["0"-"7"])* ["o", "O"] ) |
                  (["0"-"9"] (["0"-"7","A"-"F","a"-"f"])* ["h", "H"
                     ] ) |
                  (["0"-"1"] (["0"-"1"])* ["b", "B"])
                ) >
|
  < string_constant:    // constante string como "abcd bcda"
      "\""( ~["\"","\n","\r"])* "\"" >
|
  < null_constant: "null" > // constante null
}

/* Identificadores */

TOKEN :
{
  < IDENT: <LETTER> (<LETTER>|<DIGIT>)* >
|
  < #LETTER:["A"-"Z","a"-"z"] >
|
  < #DIGIT:["0"-"9"] >
}
```

```
/* Símbolos especiais */

TOKEN :
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < LBRACE: "{" >
| < RBRACE: "}" >
| < LBRACKET: "[" >
| < RBRACKET: "]" >
| < SEMICOLON: ";" >
| < COMMA: "," >
| < DOT: "." >
| < PARALLEL: "||" >
| < CHOICE: "|" >
}

/* Operadores */

TOKEN :
{
  < ASSIGN: "=" >
| < GT: ">" >
| < LT: "<" >
| < EQ: "==" >
| < LE: "<=" >
| < GE: ">=" >
| < NEQ: "!=" >
| < PLUS: "+" >
| < MINUS: "-" >
| < STAR: "*" >
| < SLASH: "/" >
}

/* Token retornado ParserException */

 < * > TOKEN :
{
< UNEXPECTED_CHAR : ~[] >
}

/**
 * A gramática de PEWS começa aqui
 **/

Document program(String name) :
{
}
```

```
{
  {
    createDomDocument();
    root = doc.createElement("envelope");
    root.setAttribute("xmlns","http://aquarius.inf.ufpr.br");
    root.setAttributeNS("http://www.w3.org/2001/XMLSchema-instance",
       "xsi:schemaLocation","http://aquarius.inf.ufpr.br pews.xsd");
    doc.appendChild(root);
    behaviour = doc.createElement("behaviour");
    root.appendChild(behaviour);
    operations = doc.createElement("operations");
    behaviour.appendChild(operations);
    behaviour.setAttribute("name",name);
    Element path_expr;
  }
          (ns())+ (alias())+ (def())* path_expr = path_expr() <EOF
             > {
            pathExp = doc.createElement("pathExp");
            behaviour.appendChild(pathExp);
            pathExp.appendChild(path_expr);
            return doc;
          }
}

void ns() :
{
  Token Tnamespace;
  Token Tfile;
}
{
  <NS> Tnamespace = namespace(true) <ASSIGN> Tfile = file()
  {
    behaviour.setAttribute("xmlns:" + Tnamespace.image, Tfile.image.
       substring(1,
              Tfile.image.length() - 1));
  }
}

void alias() :
{
  Token Topname;
  Token Toperation;
  Token Tnamespace;
  Token TportType;
}
{
  <ALIAS> Topname = opname(true) <ASSIGN> TportType = portType()<
     SLASH>Toperation = operation() <IN> Tnamespace = namespace(false
     )
  {
```

```
    Element operation = doc.createElement("operation");
    operation.setAttribute("name", Topname.image);
    operation.setAttribute("portType", TportType.image);
    operation.setAttribute("refersTo", Tnamespace.image + ":" +
       Toperation.image);
    operations.appendChild(operation);
  }
}

Token portType() :
{
  Token t;
}
{
  t = <IDENT> {
    return t;
  }
}

Token operation() :
{
  Token t;
}
{
  t = <IDENT> {
    return t;
  }
}

Token namespace(boolean add) :
{
  Token t;
}
{
  (t = <IDENT>
  {if (symTab.containsKey(t.image)) {
    if (add) {
      lineParseError = t.beginLine;
      throw new ParseException("Encountered \"" + t.image + "\" at
         line " + t.beginLine + ", column " + t.beginColumn + "." + "
          Name Space already declared");
    }
  } else {
    if (add) {
      symTab.put(t.image,new Double(0));
    } else {
      lineParseError = t.beginLine;
      throw new ParseException("Encountered \"" + t.image + "\" at
         line " + t.beginLine + ", column " + t.beginColumn + "." + "
          Name Space not declared");
    }
```

```
  }
  return t;
}
  )
}

Token file() :
{
  Token t;
}
{
  t = <string_constant>
  {
    return t;
  }
}

void def() :
{
  Element arith_expr;
  Token Tvar;
}
{
  <DEF> Tvar = var(true) <ASSIGN> arith_expr = arith_expr() {
    Element varDef = doc.createElement("varDef");
    varDef.setAttribute("name", Tvar.image);
    varDef.appendChild(arith_expr);
    behaviour.appendChild(varDef);
  }
}

Token var(boolean add) :
{
  Token t;
}
{
  t = <IDENT> {
    if (symTab.containsKey(t.image)) {
      if (add) {
        lineParseError = t.beginLine;
        throw new ParseException("Encountered \"" + t.image + "\" at
            line " + t.beginLine + ", column " + t.beginColumn + "."
             + " Variable already declared");
      }
    } else {
      if (add) {
        symTab.put(t.image,new Double(0));
      } else {
        lineParseError = t.beginLine;
        throw new ParseException("Encountered \"" + t.image + "\" at
            line " + t.beginLine + ", column " + t.beginColumn + "."
```

```
            + " Variable not declared");
      }
    }
    return t;
  }
}

Element pred_expr() :
{
  Element pred_term;
  Element pred_expr2 = null;
}
{
  pred_term = pred_term() [pred_expr2 = pred_expr2(pred_term)] {
    if (pred_expr2 != null)
      return pred_expr2;
    else
      return pred_term;
  }
}

Element pred_expr2(Element firs) :
{
  Element pred_term;
}
{
  <OR> pred_term = pred_term() {
    Element or = doc.createElement("or");
    or.appendChild(firs);
    or.appendChild(pred_term);
    return or;
  }
}

Element pred_term() :
{
  Element pred_factor;
  Element pred_term2 = null;
}
{
  pred_factor = pred_factor() [pred_term2 = pred_term2(pred_factor)
     ] {
    if (pred_term2 != null)
      return pred_term2;
    else
      return pred_factor;
  }
}

Element pred_term2(Element firs) :
{
```

```
  Element pred_factor;
}
{
  <AND> pred_factor = pred_factor() {
    Element and = doc.createElement("and");
    and.appendChild(firs);
    and.appendChild(pred_factor);
    return and;
  }
}

Element pred_factor() :
{
  Element bool_expr;
  Token Tnot = null;
}
{
  [Tnot = <NOT>] bool_expr = bool_expr() {
    if (Tnot != null) {
      Element not = doc.createElement("not");
      not.appendChild(bool_expr);
      return not;
    } else
      return bool_expr;
  }
}

Element bool_expr() :
{
  Element bool = null;
  Element pred_expr;
  Element arith_expr;
  Element arith_expr2;
}
{
  <TRUE> {
    bool = doc.createElement("bool");
    bool.setAttribute("value", "true");
    return bool;
  }
  | <FALSE> {
    bool = doc.createElement("bool");
    bool.setAttribute("value", "false");
    return bool;
  }
  | LOOKAHEAD(2) arith_expr = arith_expr() ( <LT> {bool = doc.
     createElement("LT");}
    | <GT> {bool = doc.createElement("GT");}
    | <LE> {bool = doc.createElement("LEQ");}
    | <GE> {bool = doc.createElement("GEQ");}
    | <EQ> {bool = doc.createElement("EQ");}
```

```
    | <NEQ> {bool = doc.createElement("NEQ");} ) arith_expr2 =
       arith_expr() {
      bool.appendChild(arith_expr);
      bool.appendChild(arith_expr2);
      return bool;
    }
  |  <LPAREN> pred_expr = pred_expr() <RPAREN> {
    return pred_expr;
  }


}

Element path_expr() :
{
  Element parallel0;
  Element parallel1;
  Element parallel2 = null;
}
{
  {
    parallel0 = doc.createElement("par");
  }
  parallel1 = parallel() {
    parallel0.appendChild(parallel1);
  } (<PARALLEL> parallel2 = parallel() {
    parallel0.appendChild(parallel2);
  } )* {
    if (parallel2 == null)
      return parallel1;
    else
      return parallel0;
  }
}

Element parallel() :
{
  Element choice0;
  Element choice1;
  Element choice2 = null;
}
{
  {
    choice0 = doc.createElement("choice");
  }
  choice1 = choice() {
    choice0.appendChild(choice1);
  } (<CHOICE> choice2 = choice() {
    choice0.appendChild(choice2);
  } )* {
    if (choice2 == null)
```

```
      return choice1;
    else
      return choice0;
  }
}

Element choice() :
{
  Element sequence0;
  Element sequence1;
  Element sequence2 = null;
}
{
  {
    sequence0 = doc.createElement("seq");
  }
  sequence1 = sequence() {
    sequence0.appendChild(sequence1);
  } (<DOT> sequence2 = sequence() {
    sequence0.appendChild(sequence2);
  } )* {
    if (sequence2 == null)
      return sequence1;
    else
      return sequence0;
  }

}

Element sequence() :
{
  Element unarypath;
  Element path_expr;
}
{

  <LBRACE> path_expr = path_expr() <RBRACE> {
    Element parRep = doc.createElement("parRep");
    parRep.appendChild(path_expr);
    return parRep;
  }
  | unarypath = unarypath() {
    return unarypath;
  }
}

Element unarypath() :
{
  Element path;
  Element star = null;
}
```

```
{
  path = path() [(<STAR> {
    star = doc.createElement("star");
    star.appendChild(path);
  }
  | <PLUS>)] {
    if (star == null)
      return path;
    else
      return star;
  }
}

Element path() :
{
  Token Topname;
  Element operation;
  Element path_expr;
  Element pred_expr;
  Element pred;
}
{
  Topname = opname(false) {
    operation = doc.createElement("operation");
    operation.setAttribute("name", Topname.image);
    return operation;
  }
  | <LBRACKET> pred_expr = pred_expr() <RBRACKET> path_expr = path()
      {
    pred = doc.createElement("pred");
    pred.appendChild(pred_expr);
    pred.appendChild(path_expr);
    return pred;
  }
  | <LPAREN> path_expr = path_expr() <RPAREN> {
    return path_expr;
  }
  | <ABORT> {
    operation = doc.createElement("operation");
    operation.setAttribute("name", "abortOperation");
    return operation;
  }
}

Element arith_expr() :
{
  Element term;
  Element arith_expr2 = null;
}
{
  term = term() [arith_expr2 = arith_expr2(term)] {
```

```
    if (arith_expr2 != null)
      return arith_expr2;
    else
      return term;
  }
}

Element arith_expr2(Element firs) :
{
  Element term;
}
{
  <PLUS> term = term() {
    Element sum = doc.createElement("sum");
    sum.appendChild(firs);
    sum.appendChild(term);
    return sum;
  }
  | <MINUS> term = term() {
    Element minus = doc.createElement("minus");
    minus.appendChild(firs);
    minus.appendChild(term);
    return minus;
  }
}

Element term():
{
  Element unaryexpr;
  Element term2 = null;
}
{
  unaryexpr = unaryexpr() [term2 = term2(unaryexpr)] {
    if (term2 != null)
      return term2;
    else
      return unaryexpr;
  }
}

Element term2(Element firs) :
{
  Element unaryexpr;
}
{
  <STAR> unaryexpr = unaryexpr() {
    Element mult = doc.createElement("mult");
    mult.appendChild(firs);
    mult.appendChild(unaryexpr);
    return mult;
  }
```

```
  | <SLASH> unaryexpr = unaryexpr() {
    Element div = doc.createElement("div");
    div.appendChild(firs);
    div.appendChild(unaryexpr);
    return div;
  }
}

Element unaryexpr() :
{
  Element factor;
  Token Tminus = null;
}
{
  [Tminus = <MINUS>] factor = factor() {
    if (Tminus != null) {
      Element minusUn = doc.createElement("minusUn");
      minusUn.appendChild(factor);
      return minusUn;
    } else
      return factor;
  }
}

Element factor() :
{
  Token Tint_constant;
  Token Topname;
  Token Tvar;
  Element arith_expr;
  Element pewsCounter;
}
{
  <NOW> <LPAREN> <RPAREN> {
    Element libFunction = doc.createElement("libFunction");
    libFunction.setAttribute("name","now");
    libFunction.setAttribute("unit","miliseconds");
    return libFunction;
  }
  | LOOKAHEAD(6) <ACT> <LPAREN> Topname = opname(false) <RPAREN> <DOT
    > <VAL> {
    pewsCounter = doc.createElement("pewsCounter");
    pewsCounter.setAttribute("opname",Topname.image);
    pewsCounter.setAttribute("name","act");
    pewsCounter.setAttribute("component","val");
    pewsCounter.setAttribute("unit","miliseconds");
    return pewsCounter;
  }
  | <ACT> <LPAREN> Topname = opname(false) <RPAREN> <DOT> <TIME> {
    pewsCounter = doc.createElement("pewsCounter");
    pewsCounter.setAttribute("opname",Topname.image);
```

```
    pewsCounter.setAttribute("name","act");
    pewsCounter.setAttribute("component","time");
    pewsCounter.setAttribute("unit","miliseconds");
    return pewsCounter;
  }
  | LOOKAHEAD(6) <TERM> <LPAREN> Topname = opname(false) <RPAREN> <
     DOT> <VAL> {
    pewsCounter = doc.createElement("pewsCounter");
    pewsCounter.setAttribute("opname",Topname.image);
    pewsCounter.setAttribute("name","term");
    pewsCounter.setAttribute("component","val");
    pewsCounter.setAttribute("unit","miliseconds");
    return pewsCounter;
  }
  | <TERM> <LPAREN> Topname = opname(false) <RPAREN> <DOT> <TIME> {
    pewsCounter = doc.createElement("pewsCounter");
    pewsCounter.setAttribute("opname",Topname.image);
    pewsCounter.setAttribute("name","term");
    pewsCounter.setAttribute("component","time");
    pewsCounter.setAttribute("unit","miliseconds");
    return pewsCounter;
  }
  | LOOKAHEAD(6) <REQ> <LPAREN> Topname = opname(false) <RPAREN> <DOT
     > <VAL> {
    pewsCounter = doc.createElement("pewsCounter");
    pewsCounter.setAttribute("opname",Topname.image);
    pewsCounter.setAttribute("name","req");
    pewsCounter.setAttribute("component","val");
    pewsCounter.setAttribute("unit","miliseconds");
    return pewsCounter;
  }
  | <REQ> <LPAREN> Topname = opname(false) <RPAREN> <DOT> <TIME> {
    pewsCounter = doc.createElement("pewsCounter");
    pewsCounter.setAttribute("opname",Topname.image);
    pewsCounter.setAttribute("name","req");
    pewsCounter.setAttribute("component","time");
    pewsCounter.setAttribute("unit","miliseconds");
    return pewsCounter;
  }



  | Tvar = var(false) {
    Element var = doc.createElement("var");
    var.setAttribute("name",Tvar.image);
    return var;
  }
  | Tint_constant = <int_constant> {
    Element Econst = doc.createElement("const");
    Econst.setAttribute("value",Tint_constant.image);
    return Econst;
```

```
  }
  | <LPAREN> arith_expr = arith_expr() <RPAREN> {
    return arith_expr;
  }
}

Token opname(boolean add) :
{
  Token t;
}
{
  (t = <IDENT>
  {if (symTab.containsKey(t.image)) {
    if (add) {
      lineParseError = t.beginLine;
      throw new ParseException("Encountered \"" + t.image + "\" at
         line " + t.beginLine + ", column " + t.beginColumn + "." + "
          Operation already declared");
    }
  } else {
    if (add) {
      symTab.put(t.image,new Double(0));
    } else {
      lineParseError = t.beginLine;
      throw new ParseException("Encountered \"" + t.image + "\" at
         line " + t.beginLine + ", column " + t.beginColumn + "." + "
          Operation not declared");
    }
  }
  return t;
}
  )
}
```