UNIVERSIDADE FEDERAL DE PERNAMBUCO
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
CENTRO DE INFORMÁTICA

ESTENDENDO O ACTIVEBPEL COM WS-POLICY

TRABALHO DE GRADUAÇÃO

Aluno: Bruno Leonardo Barros Silva (blbs@cin.ufpe.br)
Orientador: Nelson Souto Rosa (nsr@cin.ufpe.br)

DEZEMBRO DE 2008

“Pressure pushing down on me
Pressing down on you no man ask for”
Freddie Mercury
5/09/1946 – 24/11/1991
Compartilho as mesmas idéias do autor
de Under Pressure

Agradecimentos

- Primeiramente, a Deus; a gente entra na ciência e naturalmente Ele é deixado algumas vezes em segundo plano, no entanto sempre temos provas de Sua existência, não há como negar;

- A meus pais, minha base sólida nessa vida, que sempre me apoiaram nessa vinda para Recife, e sempre ficaram felizes com minhas conquistas (apesar de não entenderem exatamente o que era a conquista em si em algumas vezes, hehe) e me deram apoio nos momentos de dúvida;

- A minhas avós e minhas tias, especialmente Tia Noninha pelo carinho e Tia Corrinha pelo suporte e ter me acolhido em sua casa, e de ter me agüentado quase dois anos em sua casa todos os fins de semana;

- Aos professores do Centro, pelo apoio e pela humildade a passar seus conhecimentos, mas em especial aos professores: Nelson Rosa, sempre conseguindo a proeza de me deixar mais tranqüilo e esperançoso após cada reunião, além das conversas "além-graduação", me fazendo enxergar possibilidades; Alex Sandro Gomes, por mostrar o lado humano atrás das máquinas frias da computação;

- Aos funcionários do Centro, por sempre colaborar com nossa estadia nessa nossa segunda casa, em especial a Seu Carlos, Jorge, Hilda e Inácia;

- Aos amigos José de Arimatéa e Elton Renan, dois "cabra bom", "matuto arretado", com os quais tive o prazer e a honra de dividir quarto na Casa do Estudante Universitário da UFPE;

- Aos amigos de curso, sempre compartilhando os momentos de aperreio: Xico Bento, Arruda, Brayner, Pachêco, Leo, Vini Matador, Digão, Age Man, Glerter, Diego (o Dop), Fábio (Thime) ...

- Ao pessoal da Maratona de Programação, pelos momentos descontraídos e pelas conquistas que tivemos juntos;

- A galera do BNB, saudades daquele tempo de estagiário. Abraços aos amigos que fiz por lá, principalmente a Renata (a Maga), Clarissa, Josemberg (Bergulino), Denise, Iris, Bamzinha e Mario Anderson;

- Ao meu amigo Eduardo e sua mãe Jandira, que durante meu ensino médio foram minha segunda família enquanto estudava numa outra cidade, que apesar da distância, ainda os considero como tal;

- A Freddie Mercury, Brian May, Bruce Dickinson, Steve Harris, Andre Matos, Klaus Meine, Gene Simmons, Michael Kiske, eh, bem, vou parar por aqui senão eu não vou conseguir acabar... Suas músicas inspiram, ajudam a manter acordado durante os projetos do CIn, nos leva a shows em outras capitais, enfim... \w/;

- E finalmente, a duas pessoas muito importantes, que ficaram muito decepcionadas (eu aposto!!) procurando nos agradecimentos e não vendo seus nomes; coloquei por último de propósito, vocês são muito agoniadas, hehe. A primeira é minha irmã Juliana, ciumenta de primeira!! Achou que eu tinha esquecido de tu NE? É claro que eu ia lembrar de você, obrigado por tudo, principalmente nesses últimos tempos por ser minha amiga. (É, sou amigo de minha irmã!!!). A segunda, não menos importante, é minha companheira de todas as horas, que sempre me dá apoio e carinho, atenção, amor, e essas coisas todas. Rebeka: I (L) you!

Resumo

É conhecido o uso dos chamados Web Services para interligar sistemas distribuídos, e também que muito esforço tem sido feito em direção à padronização de como esses serviços são disponibilizados, e como é feita a composição destes serviços. Nesse sentido, foi criado o BPEL (Business Process Execution Language), uma linguagem abstrata de definição de processos de negócios, destinada à programação de mais alto nível. Dentre as implementações de BPEL, existe a ActiveBPEL, uma implementação OpenSource. Este trabalho tem como objetivo estender a ferramenta ActiveBPEL para incluir a especificação WS-Policy de WebServices.

Palavras-chave: Web services, Composição de web services, BPEL, Política, WS-Policy

Sumário

Lista de Figuras

Lista de Trechos de códigos

1 Introdução

Com a popularização da Internet, não apenas como um meio de entretenimento, mas como provedor de serviços, onde fazemos compras, efetuamos pagamentos, vemos as notícias praticamente na hora em que elas acontecem, entre outras inúmeras coisas, houve um aumento no uso de sistemas distribuídos, e como conseqüência, sua natural evolução.

O que inicialmente se resumia apenas à chamadas remotas de métodos, com o foco em como os serviços funcionam, atualmente acontece através de interações mais complexas entre os próprios serviços, onde claramente há uma mudança de foco, estando mais direcionado ao que estes serviços fazem e como eles interagem.

Nesse cenário de uso da Internet como provedor de serviços, conhecido por SOA (*Service Oriented Architecture)*, os *web services* têm um papel importante como integrador entre as entidades fornecedoras destes serviços. No entanto, a utilização de *web services* isolados não tem muita valia para casos mais complexos, e vê-se a necessidade de agrupá-los; a este agrupamento chamamos de composição de *web services*.

A composição de *web services* surge então, como uma forma de, a partir de serviços mais simples, conseguir desenvolver aplicações mais complexas [7]. Uma das principais linguagens utilizadas para descrever composições é BPEL (*Business Process Execution Language*) [3]. Esta linguagem define os modelos de negócio associados às chamadas dos *web services*. Existem muitas engines que implementam o BPEL, por exemplo: Apache ODE [4], Orchestra [10] e ActiveBPEL [1].

Como a complexidade da aplicação aumenta ao utilizar uma composição, temos novos problemas a enfrentar, mas com atenção especial a este: como saber se os *web services* participantes de uma composição obedecem a alguma

restrição aplicada à composição como um todo? Por exemplo, como afirmar que uma composição é segura? Isto só é verdade se cada *web service* participante for seguro.

Dentre as *engines* existentes, não há uma solução que consiga resolver este problema da forma como está enunciado. A engine que mais se aproximou de algo nesse sentido foi a ActiveBPEL, que dá suporte a leitura de políticas, mas não utiliza essas políticas associadas a nenhum *web service*. Aparentemente, o suporte a políticas no ActiveBPEL, ainda está em aberto, estando relacionado a um trabalho futuro pela equipe de desenvolvimento do ActiveBPEL. Não apenas por este motivo, mas também por esta ser uma engine de código aberto, foi escolhida como objeto de estudo deste trabalho.

O trabalho proposto é agregar conceitos de políticas às composições construídas usando a linguagem BPEL, incluindo uma forma de avaliar se a política aplicada à composição é compatível com as políticas associadas a cada *web service* participante da composição. Para este fim, inicialmente deverá ser modelada e implementada uma API (*Application Programming Interface*) que atenda à especificação mais nova de políticas para *web services*, o WS-Policy [17]. Uma vez implementada essa API, a *engine* escolhida deverá ser alterada para refletir essas mudanças.

É bom lembrar que o foco deste trabalho será a aplicação do conceito de políticas acopladas a composições, e não as composições em si. Apesar de ser importante entender como estas composições são feitas, o WS-Policy deverá ser implementado de forma genérica o suficiente para que seja possível anexá-lo a qualquer estrutura de dados ou componente no qual faça sentido agregar conceitos de políticas. Além do mais, a inclusão do tratamento de políticas terá caráter informativo apenas, ou seja, não irá alterar o tipo de resposta que a engine fornece toda vez que uma composição é invocada.

Os capítulos seguintes deste documento estão organizados como segue:

- **Capítulo 2** – Neste capítulo serão discutidos os conceitos básicos relacionados com este trabalho. Os pontos a serem discutidos serão: arquitetura orientada a serviços (SOA), *web services*, composição de *web services* + BPEL, políticas + WS-Policy e a engine ActiveBPEL. O entendimento destes conceitos é vital para um melhor aproveitamento do conteúdo deste trabalho;

- **Capítulo 3** – Este capítulo contém a proposta de inclusão de políticas nas composições, dividida em visão geral da proposta, requisitos, arquitetura, projeto e implementação. Para cada um destes tópicos, haverá detalhes tanto sobre a especificação WS-Policy quanto sobre a engine ActiveBPEL, inclusive a integração entre estes;

- **Capítulo 4** – Neste capítulo será apresentado um exemplo, a Calculadora remota, que ilustra o uso da abordagem proposta. Este exemplo servirá para validar a proposta de trabalho, bem como mostrar com a *engine* funciona. Ao final do capítulo, uma pequena análise dos resultados obtidos;

- **Capítulo 5** – Por fim, este será a conclusão do trabalho, com um breve resumo do que foi efetuado, mostrando as possíveis vantagens e desvantagens da proposta, bem como suas limitações. Associadas as limitações, propostas de aperfeiçoamento serão mostradas em conjunto com os trabalhos futuros.

2 Conceitos Básicos

Neste capítulo serão apresentados alguns conceitos básicos importantes para um melhor entendimento do trabalho. Primeiro, são apresentados os principais conceitos de serviços, com foco na arquitetura orientada a serviços. Logo em seguida, será mostrado o que são os *web services*, principal padrão desta arquitetura. A explicação sobre composição de *web services*, juntamente com uma breve descrição do BPEL, virá em seguida. Concluímos o capítulo mostrando como funciona a *engine* ActiveBPEL, foco da implementação deste trabalho.

2.1 Serviços / Arquitetura Orientada a Serviços (SOA)

SOA – *Service-Oriented Architecture*, ou Arquitetura Orientada a Serviços – pode ser definido como um grupo de serviços que se intercomunicam, um novo paradigma computacional no qual os desenvolvedores estão agrupados em três grupos, em termos de suas responsabilidades: *service requesters*, *service brokers*, e os *service providers*. [11].

- **Service providers** – Desenvolvem seus sistemas baseados apenas em protocolos e padrões, independente das aplicações em potencial que porventura utilizaria seus serviços;

- **Service brokers** – Responsáveis por publicar os serviços disponíveis aos clientes em potencial. Servem como um catálogo onde os *providers* informam que possuem serviços disponíveis.

- **Service requesters** – Procuram pelos serviços desejados nos *service brokers* e compõem a aplicação destino usando os serviços disponíveis.

A **Figura 2.1**, mostra a relação entre os elementos existentes em SOA.

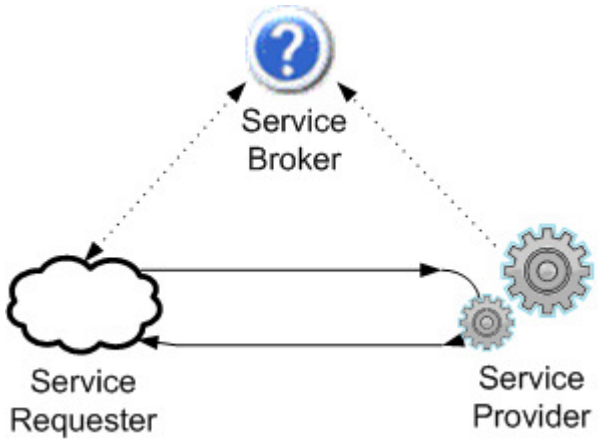

Figura 2.1 - Arquitetura de SOA

2.2 Web Services

Web service é um sistema de software desenvolvido para suportar a interoperabilidade entre máquinas numa rede [15]. Como pudemos perceber, a definição de *web service* se sobrepõe a de SOA. Isso é natural, pois o *web service* é a implementação concreta de SOA mais bem sucedida. Alguns críticos chegam ao ponto de afirmar que SOA é na verdade, apenas outro nome para Web Service [12], tamanha a semelhança entre suas definições.

Um exemplo simples de *web service* pode ser visto na **Figura 2.2**:

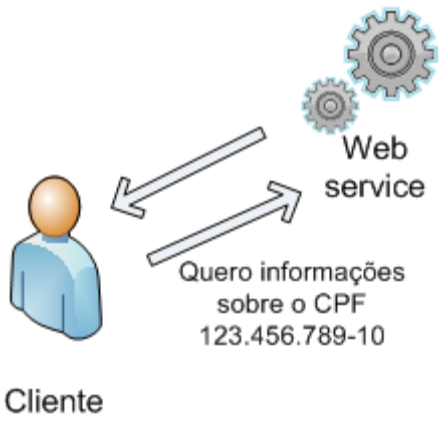

Figura 2.2 - Exemplo de Web Service

O cliente, que pode ser uma instituição financeira, por exemplo, deseja saber qual a situação cadastral do titular do CPF. Para isto, envia uma solicitação ao servidor do SERASA, que retorna uma resposta ao cliente contendo as informações desejadas.

Podemos nos questionar, se já existiam outras plataformas de sistema distribuído, qual o diferencial do *web service*, comparando-o com as outras plataformas? O ponto crucial que responde à esta pergunta é o uso do HTTP como meio de transporte. Isto permite que os *web services* possam trafegar livremente através de *firewalls*, pois normalmente o tráfico é liberado para a porta 80.

Na **Figura 2.3** vemos a arquitetura padrão de *web services*. Nela vemos três entidades importantes: o provedor dos serviços (Servidor), responsável pelas regras de negócio do serviço, e por conter a implementação propriamente dita do serviço; o diretório de *web services* (UDDI), que indica em qual servidor determinado serviço está localizado; por fim, o cliente, que se comunica com o diretório para saber quem provê o serviço o qual ele está precisando, e logo em seguida invoca o serviço no servidor.

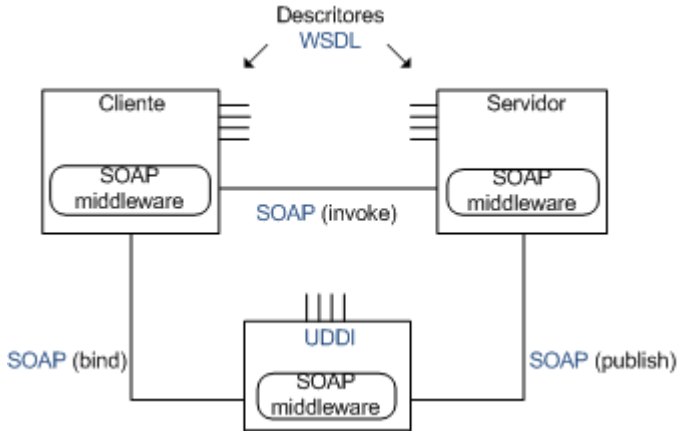

Figura 2.3 - Arquitetura de web services

Alguns protocolos e tecnologias são utilizados nesse processo, apresentados logo em seguida.

2.2.1 SOAP (Simple Object Access Protocol)

Este protocolo foi desenvolvido com o intuito de auxiliar o compartilhamento de informações estruturadas entre *web services* [13]. SOAP utiliza o formato XML para montar as mensagens estruturadas, divididas em cabeçalho e corpo. O cabeçalho contém informações de controle, como timeout do serviço, por exemplo, e o corpo contém a mensagem.

Uma vez que SOAP é apenas a definição de um formato de mensagens a ser trocado, o protocolo de transporte fica a cargo de quem deseje implementá-lo, embora sua especificação padrão defina apenas HTTP e SMTP. O motivo destes dois padrões terem sido os únicos citados na especificação é na verdade a explicação para o surgimento de SOAP: a facilidade de comunicação entre máquinas em domínios diferentes.

Como vantagens do padrão SOAP, podemos citar a independência entre sistemas operacionais, linguagens e sua simplicidade. Entretanto, há algumas

desvantagens, um preço a se pagar pela interoperabilidade: para a representação de elementos com uma estrutura mais complexa, há muito mais dados de descrição do elemento do que conteúdo em si (este problema deve-se ao uso de XML). Outro problema é a sobrecarga na pilha de protocolos de rede: o SOAP é embutido dentro de um pacote HTTP, o que o torna menos eficiente que se este estivesse diretamente num pacote TCP/UDP.

2.2.2 WSDL (Web Services Description Language)

Esta linguagem no formato XML é utilizada para descrever *web services* [14]. O WSDL é um descritor utilizado pelos clientes para saber informações sobre o *web service*: o que o serviço faz, quais são os métodos disponíveis, quais os parâmetros que estes métodos utilizam, qual a estrutura dos dados utilizados pelo serviço, quais os tipos de falhas que podem ocorrer, como deverá ser invocado, entre outras informações.

Na prática, para um cliente invocar um *web service*, ele precisa inicialmente pedir ao servidor o descritor do serviço, para saber como invocá-lo, para logo em seguida, invocá-lo propriamente.

Com comportamento igual a interfaces em outras linguagens, o WSDL tem caráter apenas descritivo, agindo apenas como um contrato, indicando quais serviços estão ou deverão ser implementados [8]. Mesmo o WSDL tendo suporte a extensões, qualquer tratamento de semântica ficará a critério das partes envolvidas, que deverão previamente combinar o que cada elemento presente na extensão representa.

Podemos ver a estrutura de um documento WSDL em **Figura 2.4**. O documento é dividido em duas partes: abstrata e concreta. Na parte abstrata é definido como o *web service* funciona: as operações, os tipos, mensagens e *port*

types. Já na parte concreta é definido como o serviço é oferecido, como os clientes podem conectar-se e quais os protocolos de transporte utilizados.

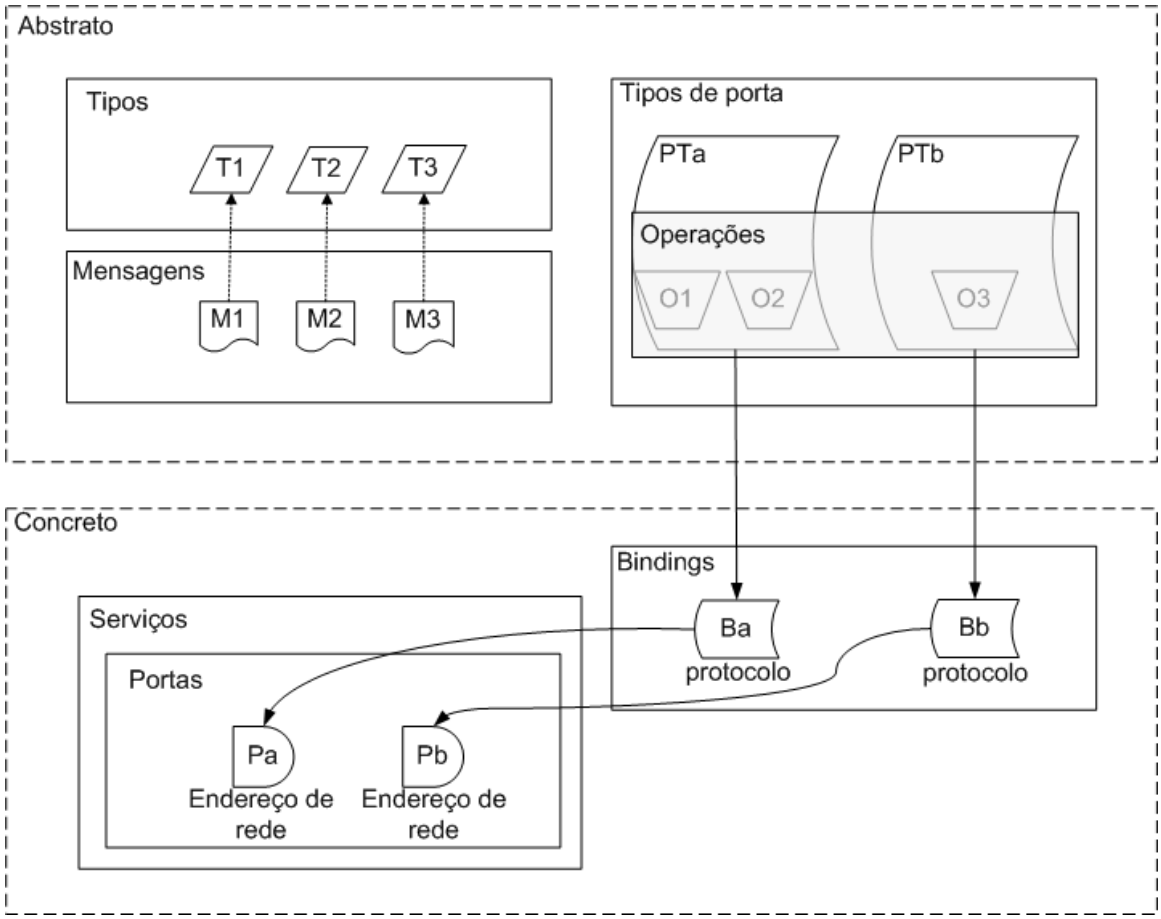

Figura 2.4 - Estrutura de um documento WSDL

As mensagens são utilizadas como parâmetros de entrada, saída ou para caso de falha em cada operação. Estas mensagens podem ser de tipos pré-definidos, ou alternativamente, de tipos definidos no próprio WSDL, na seção *types*.

2.2.3 UDDI (*Universal Description Discovery and Integration)*

O UDDI (*Universal Description Discovery and Integration*) serve como um registro de *web services*, um centralizador de informações [9]. Se o servidor tiver muitos clientes em potencial, pode não ser viável informar seu WSDL para cada possível cliente, ao passo que um cliente pode não conhecer a priori qual servidor provê o serviço que ele necessita. Foi nesse contexto que surgiu a necessidade do UDDI.

As informações no UDDI se dividem em três categorias:

- **Páginas brancas** – Fornecem informações em linguagem natural, possivelmente em múltiplas línguas, sobre o desenvolvedor do serviço, informações de contato, descrição da empresa, entre outras;
- **Páginas amarelas** – Possuem informações taxonômicas sobre o serviço, em qual (is) área (s) da indústria ele impacta. Resumindo, provê uma classificação mais formal para o serviço;
- **Páginas verdes** – Contêm informações técnicas sobre o serviço, principalmente como acessá-lo, quais os protocolos usados para invocá-lo, etc.

2.3 Composição de Web Services / BPEL

A composição de *web services* surge como solução para um dos maiores preceitos defendidos por SOA: o reuso. Ao utilizar serviços já existentes como parte de uma composição, ganhamos tempo de implementação, além de utilizar um serviço estável, apenas se preocupando na forma que estes serviços irão interagir.

Os tipos básicos de composições são mostrados na **Figura 2.5**:

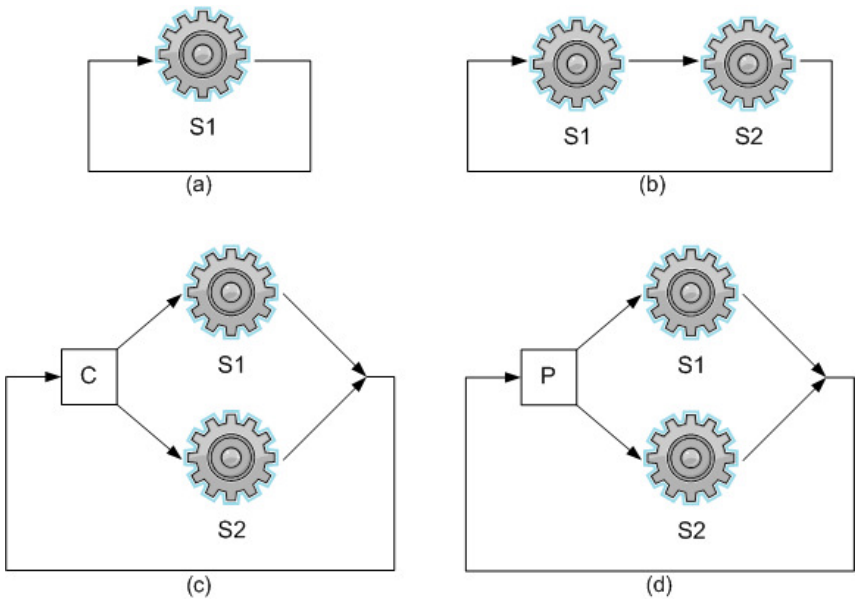

Figura 2.5 - Tipos de composições de serviços

O primeiro tipo (a) é a composição iterativa, onde um serviço é executado inúmeras vezes, utilizando seu próprio resultado como entrada. O segundo tipo (b) é a composição seqüencial, onde um serviço S1 executa, termina sua execução, e só depois o serviço S2 é invocado, utilizando o resultado de S1 como entrada. Em alguns casos, quando a utilização de um serviço depende de alguma condição, surge a composição com escolha (c). O operador de escolha, denotado por C, é responsável por avaliar a condição de uso do serviço, ou se for o caso, escolher aleatoriamente um deles. Por fim, temos a composição paralela (d), onde o operador de paralelismo (P) invoca dois ou mais serviços ao mesmo tempo.

Uma das propriedades mais importantes de uma composição é que ela também é um *web service*. Dessa forma, *web services* simples podem se agrupar num *web service* mais complexo, que por sua vez pode ser utilizado em outra composição. Isto permite um desenvolvimento gradual e iterativo, além de facilitar o teste de componentes menores, garantindo que estes funcionam bem antes de integrá-los com o restante.

Apesar de não existir um padrão que especifique composições de *web services*, o que tem se feito nesse sentido, é descrever as composições na forma de processos de negócio, principalmente usando o padrão WS-BPEL [3].

A especificação WS-BPEL é uma linguagem que descreve os processos de negócio, permitindo que vários *web services* trabalhem conjuntamente com objetivo de executar uma tarefa do negócio. Já s processos de negócio são uma coleção de invocações de serviços e atividades relacionadas que produzem um resultado de negócio, para uma ou mais empresas, visando especialmente, a integração de sistemas, parceiros comerciais e usuários corporativos [7].

Como vemos, a composição de *web services* ainda está intimamente ligada aos processos de negócio. Dessa forma, a maneira de compor *web services* está mais associada aos modelos de negócio do que a composição em si, ocasionada pelo uso de um padrão que inicialmente não foi projetado exclusivamente para composição *de web services*.

2.3.1 WS-BPEL (Business Process Execution Language)

Esta linguagem é utilizada para especificação dos processos de negócio, descrevendo as relações entre os web services participantes da composição. Apesar de também ser uma linguagem descritora, WS-BPEL possui estruturas de linguagens de programação já existentes, como loops, desvios condicionais, variáveis e atribuições, dando mais possibilidades à construção da composição.

Para descrever um processo em WS-BPEL, podemos nos utilizar de vários componentes. Os mais importantes são:

- ***Partner*** – Denota quais são os serviços participantes da composição. Normalmente indica o endereço do servidor onde os web services estão disponibilizados;

- ***Variable*** – Esta seção é bastante importante, pois nela podem ser declaradas variáveis, onde os dados serão armazenados temporariamente, por exemplo, entre duas chamadas consecutivas entre os *web services* componentes, podendo inclusive ter os seus valores alterados de acordo com alguma condição, dando uma maior flexibilidade à composição;

- ***Fault Handlers*** – Indica quais atividades / operações lançarão falhas, e o que deve ser feito caso estas falhas ocorram;

- ***Activity*** – Elemento básico do BPEL, uma atividade pode ser de dois tipos: simples ou estruturada. Existem inúmeros tipos de atividades, e cada uma possui um significado bem definido. As principais atividades aparecem na **Tabela 2.1**:

Tabela 2.1 - Principais atividades de WS-BPEL

Atividade	Descrição	Tipo
Assign	Copia valores de uma variável para outra	Simples
Empty	Atividade vazia (não realiza nenhuma tarefa)	Simples
Invoke	Responsável por invocar um *web service* participante	Simples
Receive	Sua função é receber a requisição à composição por parte do cliente	Simples
Reply	Envia o resultado da composição ao cliente	Simples
Sequence	Executa este grupo de atividades de forma seqüencial	Estruturada
Switch	Executa desvios condicionais, escolhendo uma das atividades a ser executada, baseado em alguma condição	Estruturada

Throw	Indica explicitamente a ocorrência de uma falha	Simples
While	Executa *loops* de execução, ou seja, enquanto um condição for verdadeira, a atividade será executada continuamente	Estruturada

2.4 Engine ActiveBPEL

Para executar uma composição baseada em processos de negócio, é necessário que exista um ambiente especializado, que esteja apto a entender o padrão BPEL. Uma engine BPEL consegue prover este ambiente, e é um software capaz de gerenciar um conjunto de composições em execução.

Mesmo existindo várias engines com este propósito, nosso foco será a engine ActiveBPEL [1]. Esta *engine* foi escolhida porque seu desenvolvimento já encontra-se num estágio avançado (à época deste trabalho, a ferramenta encontrava-se na Versão 5), com soluções bem definidas e muitos ciclos de *bugs* já corrigidos.

Além disso, esta engine utiliza a política de código-aberto, e é escrita em Java, o que propicia um melhor suporte a quem deseje alterá-la, através de uma comunidade ativa de desenvolvedores.

Para implantar uma composição na engine, um certo cuidado é necessário na hora de gerar os arquivos da composição, pois a engine só reconhece esta composição se estes arquivos estiverem organizados numa determinada estrutura.

A composição deverá estar no formato .bpr, que na verdade é um arquivo compactado no formado ZIP renomeado para .bpr. A estrutura de arquivos exigida é mostrada na **Figura 2.6**:

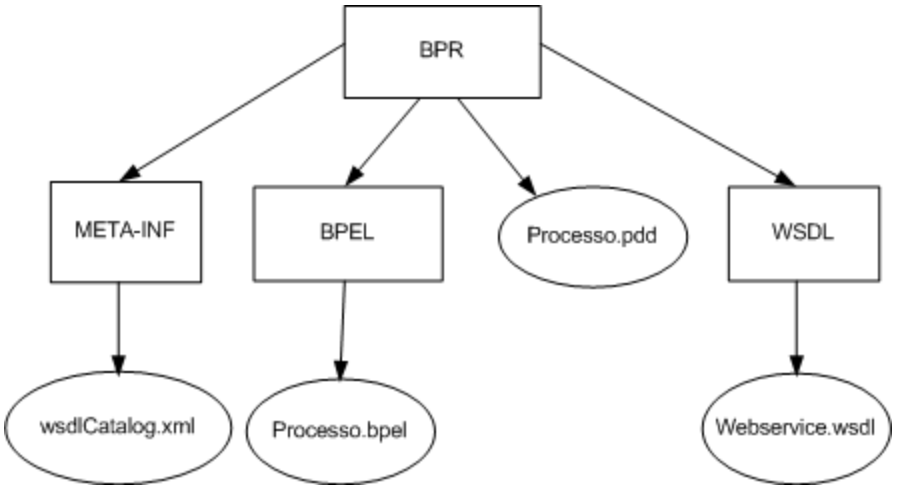

Figura 2.6 - Estrutura do arquivo .bpr

- **catalog.xml** – Contém informações sobre todos os recursos presentes no arquivo .bpr: descritores WSDL, esquemas XSD, outras composições .bpr, etc;

- ***.pdd** – Arquivo do tipo *Process Deployment Descriptor*, fornece informações sobre a implantação da composição na engine, por exemplo, se os *web services* estiverem num servidor a parte é nesse arquivo que está descrito onde localizá-los;

- ***.wsdl** – Contém os descritores WSDL de cada *Partner*, tanto dos web services componentes, quanto da própria composição.

Para a engine executar corretamente, é necessária a utilização de um servidor de aplicação capaz de executar *servlets* Java, além de fornecer uma interface administrativa para a engine. O servidor escolhido, por ser de ampla aceitação e inclusive recomendando pela comunidade, foi o Apache Tomcat 5.5 [5].

2.5 WS-Policy

Para uma melhor representação mais fiel dos requisitos de um serviço, às vezes precisamos definir algumas restrições, e nem sempre é possível fazer isto através da descrição dos parâmetros de cada operação, expressando basicamente os requisitos funcionais do serviço.

Estas restrições normalmente agregam às entidades dos *web services* características não-funcionais, como performance, segurança, disponibilidade, precisão, entre outros. Como o descritor WSDL inicialmente não pensou em prover estas características, é necessária a utilização de outro padrão que consiga provê-la.

Nesse contexto, surge o padrão WS-Policy [17], criado para permitir que um sistema baseado em *web services* fosse capaz de expressar restrições, capacidades ou características gerais das suas entidades, através de políticas.

O padrão ainda encontra-se numa fase de amadurecimento, portanto, suas soluções não ainda figuram como definitivas. Como em todo padrão novo, é preciso uma fase de adaptação e de testes pela comunidade, com o intuito de elicitar mais requisitos para o padrão.

O padrão possui os seguintes componentes:

- **Assertion** – Corresponde a um predicado simples que indica uma restrição. Este componente corresponde a um elemento XML genérico, especificado por algum esquema XSD;

- **Operator** – Permitem duas operações básicas de escolha entre as asserções: *All*, cuja política é obedecida somente se todos os subcomponentes tiverem políticas satisfeitas, e *ExactlyOne*, onde a política é obedecida se exatamente um subcomponente possuir uma política compatível;

- **Policy** – É apenas uma coleção de asserções ou operadores, que é compatível somente se todos os seus subcomponentes forem compatíveis, análogo ao operador *All*.

Por si só, o padrão não especifica como estas políticas serão associadas a elementos, por exemplo, a um descritor WSDL. Isto fica a cargo do padrão *Web Services Policy Attachment* [16], que define onde as políticas serão inseridas nos mais diversos elementos da arquitetura de *Web Services*.

No caso específico do descritor WSDL, a política é inserida dentro da tag *definitions*, na área chamada *extensions*. Cabe a cada entidade que receba os WSDL entender quais extensões estão definidas.

3 Proposta

Neste capítulo, será detalhada a proposta deste trabalho, que consiste na alteração da engine ActiveBPEL para incluir o tratamento de políticas, através da especificação WS-Policy.

Inicialmente será apresentada uma visão geral sobre o que foi feito no trabalho, e logo em seguida os requisitos do sistema. Continuaremos com a arquitetura, mostrando como os componentes do sistema serão interligados. O projeto do trabalho é o próximo tópico, e por fim concluiremos o capítulo com detalhes sobre a implementação do trabalho.

3.1 Visão Geral

A proposta deste trabalho consiste em adicionar conceitos de políticas à engine ActiveBPEL, de forma a prover características não-funcionais a uma composição. Uma política pode descrever restrições associadas ao uso do serviço, por exemplo, restrições de segurança, ou de desempenho.

A **Figura 3.1** apresenta um exemplo de como a engine ActiveBPEL ficará após a inclusão dos conceitos desde trabalho. Inicialmente um cliente receberá o descritor da composição (1), e invocará a engine (2), que irá procurar composição no repositório (3) e executá-la. Por fim, o verificador de políticas (4) será acionado para verificar a compatibilidade entre as políticas de cada web service participante.

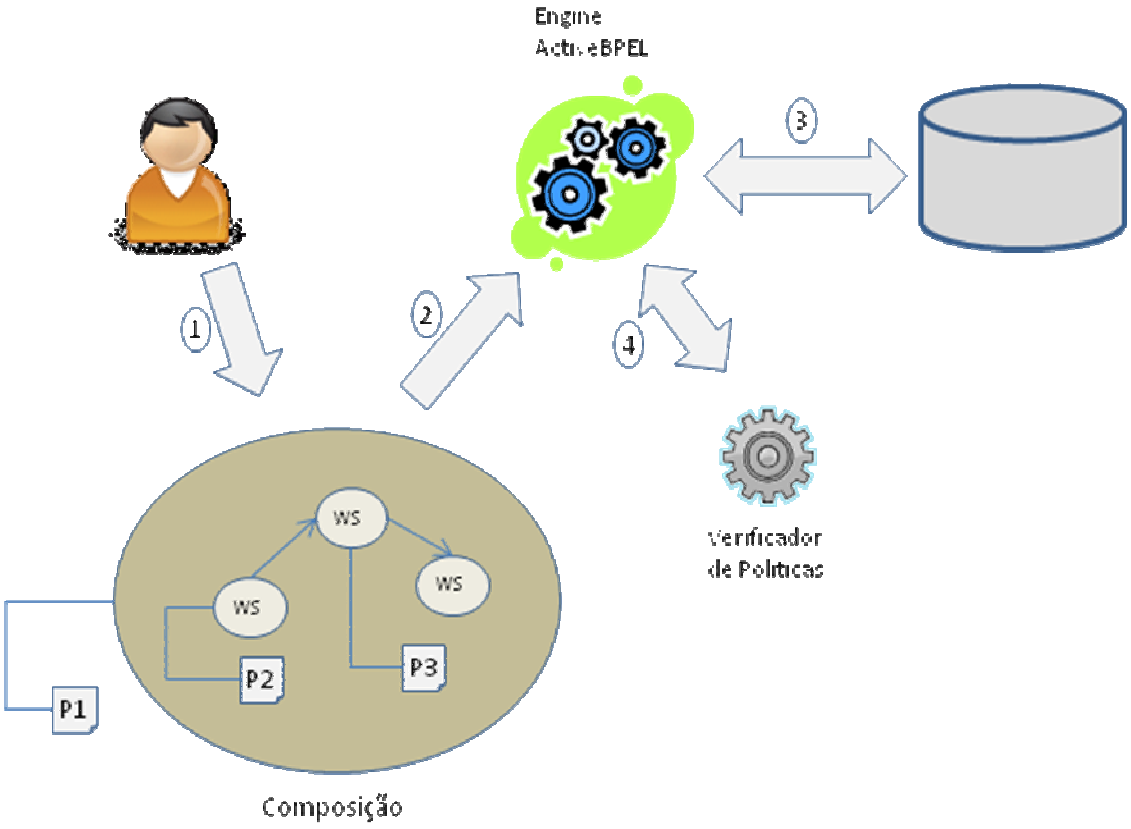

Figura 3.1 - Visão geral da proposta

Nas próximas seções será explicado em detalhes como o componente PolicyAdapter se comporta, e como ele interage com a engine.

3.2 Requisitos

Para garantir que seja incluído o tratamento de políticas na engine, sem alterar a forma como esta trata as composições, mantendo assim a compatibilidade com outras composições que não se utilizem de políticas, é preciso atender a alguns requisitos. Os principais requisitos associados a esta proposta são:

3.2.1 Requisitos funcionais

- **[RF01]** – Uma composição e um *web service* isolado deverão suportar o conceito de política, ou seja, associado a cada um deles deverá ter um descritor de quais são os requisitos não-funcionais relacionados;
- **[RF02]** – A definição da política deverá ser independente das restrições (requisitos não-funcionais) aos quais ela está relacionada;
- **[RF03]** – A engine alterada deverá ser capaz de reconhecer estes descritores de políticas associados a cada composição e/ou *web service* componente em tempo de execução;
- **[RF04]** – Dada uma composição e suas políticas, a engine alterada deverá informar se, para uma dada requisição, as políticas da composição e dos *web services* associados àquela requisição são compatíveis;
- **[RF05]** – A política poderá estar associada à composição, a um *web service* isolado ou apenas a uma operação.

3.2.2 Requisitos não-funcionais

- **[RNF01]** – O conceito de política deverá ser descrito através do uso de algum padrão conhecido;
- **[RNF02]** – A solução proposta deverá ser extensível para inclusão futura de políticas mais elaboradas;

- **[RNF03]** – O mecanismo de verificação das políticas deverá ser eficiente

Algumas observações podem ser feitas sobre alguns dos requisitos neste trabalho. Em [RF05], não será considerada a associação entre uma política e uma operação devido à complexidade exigida; esta associação estará listada como trabalho futuro. O [RNF03] está relacionado a um conceito de certa forma subjetivo: eficiência. Alguma métrica deve ser aplicada para verificar esta eficiência, e este não é o foco deste trabalho.

3.3 Arquitetura

A arquitetura da solução proposta foi planejada de forma a incluir na *engine* ActiveBPEL os conceitos de políticas, atendendo assim aos requisitos funcionais e não-funcionais vistos na seção anterior. Além disso, foi escolhida uma arquitetura que se encaixasse bem na da *engine*, sem ser preciso alterar muita coisa nela.

A **Figura 3.2** mostra a arquitetura geral da proposta integrada com a *engine* ActiveBPEL. O ator externo (que pode ser um cliente, outro serviço, ou até mesmo a própria *engine*) invoca uma composição que foi previamente implantada na *engine* ActiveBPEL. O componente InvokeBPEL é o responsável por receber esta requisição, e verificar no Web Service Repository o descritor WSDL desta composição. Ao recuperar o descritor, o componente InvokeBPEL informa ao componente PolicyAdapter se há políticas associadas à composição.

Em seguida, o componente InvokeBPEL informa ao componente InvokeWS que um *web service* deverá ser invocado. Mais uma vez, uma consulta é feita ao Web Service Repository, e caso o WSDL indique que o *web service* não está no repositório, o *web service* remoto indicado será invocado pelo componente. O

InvokeWS também informa ao PolicyAdapter se há políticas associadas ao *web service* (pois mesmo que ele seja remoto suas políticas estarão no WSDL do Web Service Repository local).

Por fim, o Policy Adapter compara as políticas relativas à composição (informadas pelo InvokeBPEL) com as políticas de cada *web service* componente (informadas pelo InvokeWS), exibindo no Console do servidor informações sobre estas políticas, inclusive se elas são compatíveis.

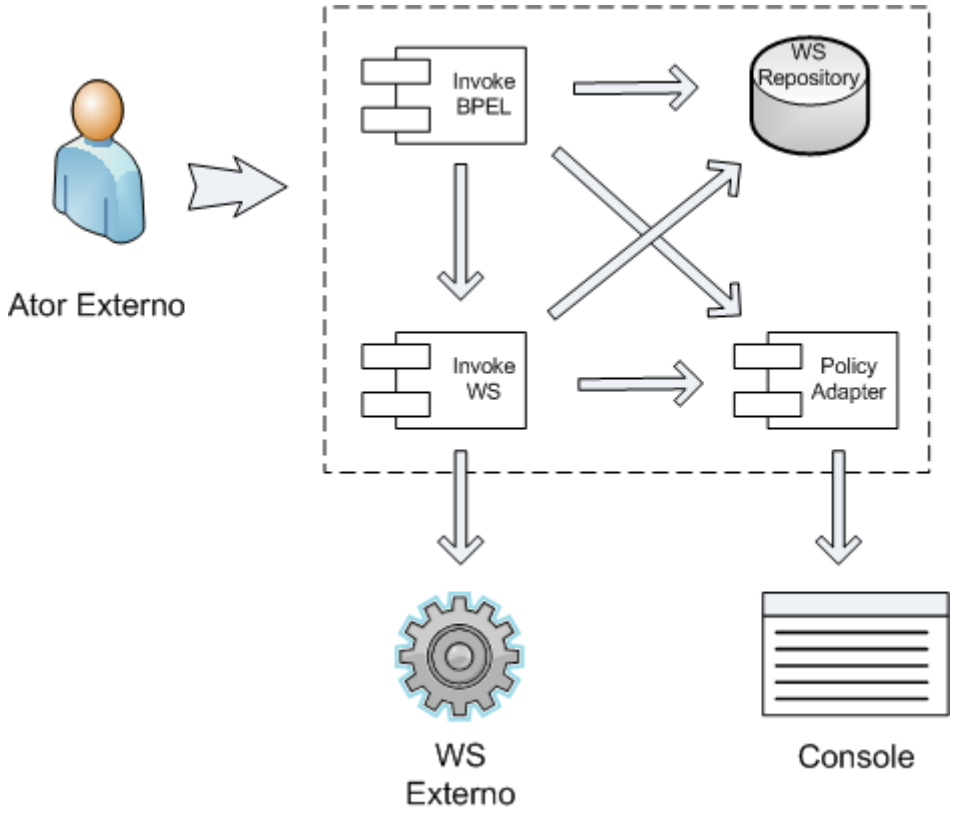

Figura 3.2 - Arquitetura da solução proposta integrada com a engine ActiveBPEL

Dentre os componentes listados, o PolicyAdapter merece uma melhor atenção, não apenas por ser o único componente inteiramente desenvolvido neste trabalho, mas também por ser um dos poucos que pode ser subdividido em outros subcomponentes.

Na **Figura 3.3** temos a arquitetura do componente PolicyAdapter. O subcomponente central é o PolicyEvaluator, responsável por todo controle do PolicyAdapter. É responsável por receber os descritores WSDL, extrair as políticas de cada um deles, criar um descritor de política para os que possuírem políticas associadas, e armazená-los no componente Policy Descriptor Repository.

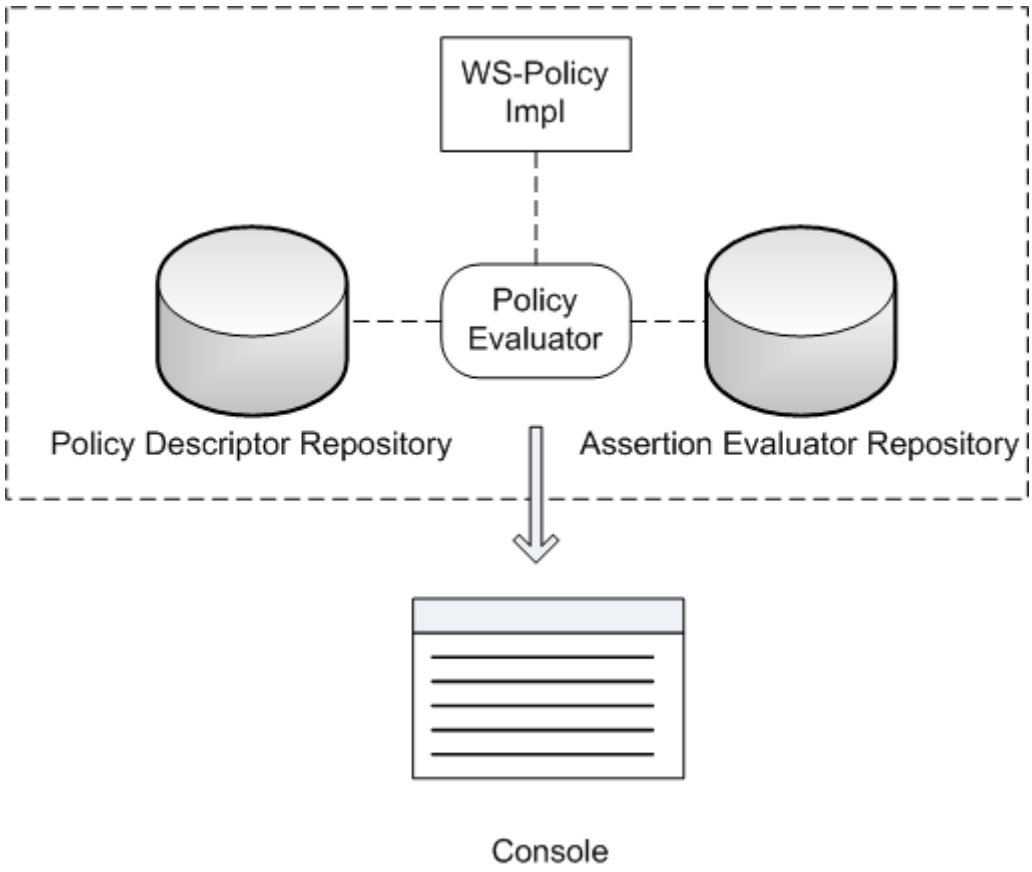

Figura 3.3 - Arquitetura do componente PolicyAdapter

Após a execução da composição, a *engine* notificará o componente PolicyAdapter. Mais uma vez, o subcomponente Policy Evaluator será invocado, dessa vez responsável por avaliar cada política do BPEL, consultando no Assertion Evaluator Repository se existe um avaliador para aquela política; caso não haja, um avaliador padrão será utilizado. O subcomponente WS-PolicyImpl

é o responsável pela análise sintática de políticas, e será detalhado na próxima seção.

3.4 Projeto

Uma vez definidos os requisitos e a arquitetura da proposta, o sistema foi modelado de acordo com os diagramas de classes apresentados na **Figura 3.4** e na **Figura 3.5**.

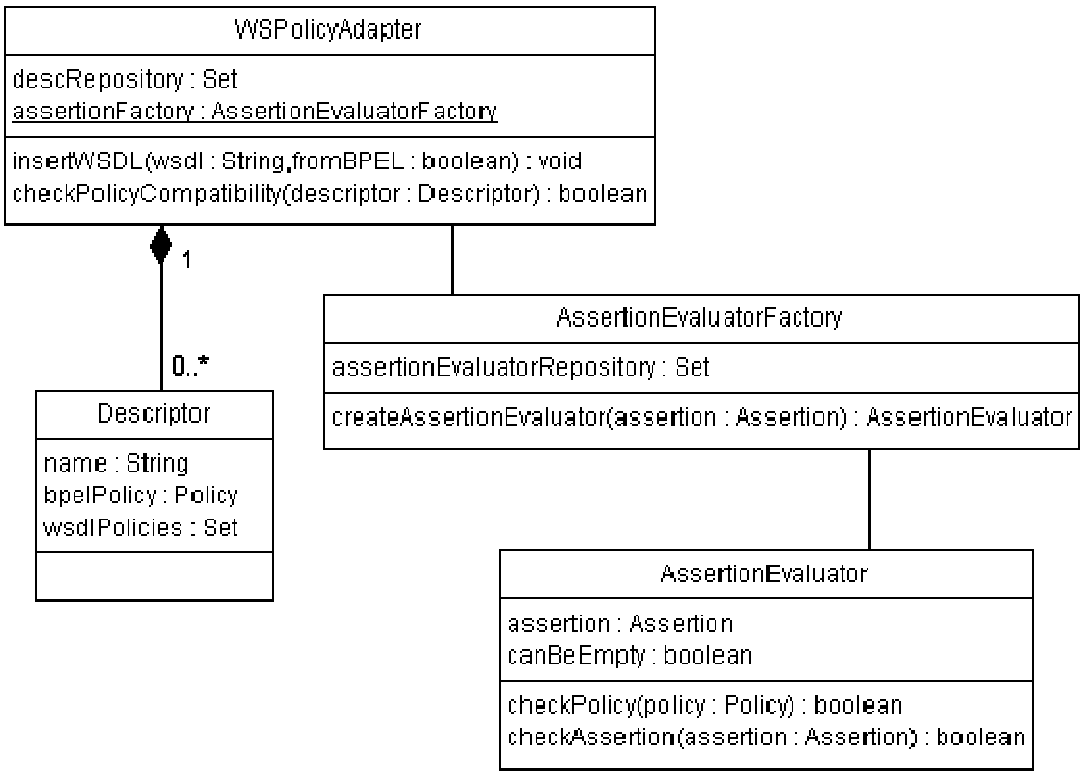

Figura 3.4 - Diagrama de classes do componente PolicyAdapter

A **Tabela 3.1** contém uma descrição das classes e interfaces do componente PolicyAdapter.

Tabela 3.1 - Descrição das classes/interfaces do componente PolicyAdapter

Classe / interface	Descrição
WSPolicyAdapter	É a classe principal do componente PolicyAdapter. Representa o subcomponente PolicyEvaluator, e é responsável por se comunicar com a engine ActiveBPEL.
AssertionEvaluator	Classe que deve ser extendida para cada novo tipo de política que deseje-se avaliar. É também a classe padrão caso não haja um avaliador disponível para uma determinada política.
AssertionEvaluatorFactory	Classe responsável por receber o nome da política e atribuí-lo a um avaliador.
Descriptor	Entidade que contém as políticas do BPEL e dos WSDL associados.

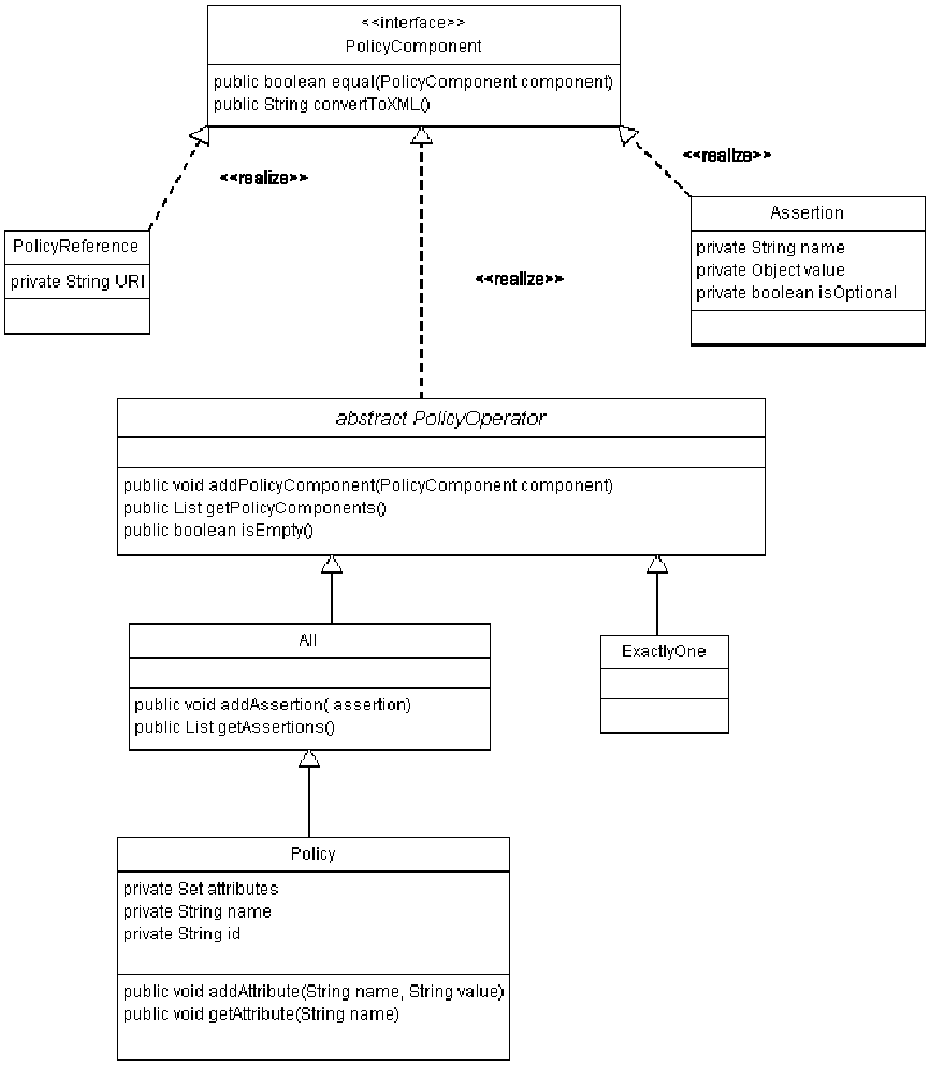

Figura 3.5 - Diagrama de classes do componente WS-PolicyImpl

A **Tabela 3.2** contém uma descrição das classes e interfaces do subcomponente WS-PolicyImpl.

Tabela 3.2 - Descrição das classes/interfaces do subcomponente WS-PolicyImpl

Classe / interface	Descrição
PolicyComponent	Interface que representa todos os componentes de políticas.

PolicyReference	Classe que representa o componente de política responsável por referenciar uma política externa.
Assertion	Classe que representa o componente de asserção, primitiva de uma política.
PolicyOperator	Classe abstrata que representa um operador de políticas, responsável por agrupar políticas de alguma maneira.
All	Classe que representa o operador All, responsável por indicar que todos os componentes incluídos devem ser considerados na avaliação para que a política seja verdadeira.
ExactlyOne	Classe que representa o operador ExactlyOne, responsável por indicar que somente um dos componentes incluídos deve ser considerado na avaliação para que a política seja verdadeira.
Policy	Principal classe do subcomponente, agrupa todos os operadores e componentes de políticas. Representa a política associada a um *web service*.

3.5 Implementação

A *engine* ActiveBPEL, como foi visto na seção 2.4, foi implementada totalmente em Java. Dessa forma, não faria muito sentido implementar esta proposta em outra linguagem senão Java.

Com a arquitetura e a modelagem em mãos, o próximo passo foi a criação de classes em Java que as representassem coerentemente. Existem três pontos de implementação distintos, onde dois deles seguem mais ou menos os maiores sub-componentes mostrados na parte de arquitetura: Policy Evaluator e WS-PolicyImpl. O terceiro ponto de implementação se refere à integração da solução com a engine já existente.

3.5.1 Policy Evaluator

Para a construção do Policy Evaluator, quatro classes foram implementadas: AssertionEvaluator, AssertionEvaluatorFactory, Descriptor e WSPolicyAdapter.

No **Código 3.1** estão detalhados os três métodos mais importantes da classe WSPolicyAdapter.

```
01   public void startBPEL(MessageContext aContext) throws
AeException {
02     AeMutableServiceDesc serviceDesc = (AeMutableServiceDesc)
aContext.getService().getServiceDescription();
03     String serviceName = serviceDesc.getName();
04
05     String transportURL = (String)
aContext.getProperty(MessageContext.TRANS_URL);
       AeWsdlReference wsdlRef = new
06 AeWsdlReference((AeWsdlReference)
   serviceDesc.getWsdlReference(), transportURL);
07
08     try {
09       wsdlRef.init();
10
11       AeBPELExtendedWSDLDef wsdlDef = wsdlRef.getWsdlDef();
12       List<IAePolicy> policyExtElements =
wsdlDef.getPolicyExtElements();
13       Policy policy = mergePolicyElements(policyExtElements);
```

```
14
15        Descriptor desc = new Descriptor();
16        desc.setName(serviceName);
17        desc.setBpelPolicy(policy);
18
19        this.bpelInvokes.push(desc);
20      } catch (AeException e) {
21        throw e;
22      }
23    }
24
25    public void wsdlHandler(String address,
AeBPELExtendedWSDLDef obj) {
26      List<IAePolicy> policyExtElements =
obj.getPolicyExtElements();
27
28      Descriptor currentBpel = this.bpelInvokes.peek();
29      Policy policy = mergePolicyElements(policyExtElements);
30      currentBpel.getWsdlPolicies().put(address, policy);
31
32      System.out.println("wsdlHandler - " + address);
33      System.out.println("currentBPEL - " +
currentBpel.getName());
34    }
35
36    public void endBPEL() {
37      Descriptor desc = this.bpelInvokes.pop();
38
39      System.out.println("Foram encontradas as seguintes
políticas no BPEL:");
40      System.out.println(desc.getBpelPolicy().toXML());
41      System.out.println(desc.getWsdlPolicies().toString());
42      System.out.println("----------------------");
43
44      boolean compatible = desc.checkPolicyCompability();
45      System.out.println(String.format("As políticas %s
compatíveis", (compatible?"são":"não são")));
46    }
```

Código 3.1 - Trecho do código da classe WSPolicyAdapter

O método startBPEL (linha 01) é chamado toda vez que o componente InvokeBPEL inicia sua execução (conforme explicado na seção 3.3), e atua da seguinte forma: o método recebe uma mensagem SOAP do componente InvokeBPEL (linha 01), busca o descritor WSDL associado no Web Service Repository e retira dele as informações sobre políticas (linhas 02-13). Um Descriptor é criado e a política é associada a ele (linhas 15-17). Logo em

seguida este Descriptor é armazenado no Policy Descriptor Repository (linha 19), implementado através da estrutura de dados pilha.

O método wsdlHandler (linha 25) é chamado toda vez que o componente InvokeWS está prestes a executar um *web service*. Similarmente ao método startBPEL, do descritor WSDL são extraídas as políticas (linhas 26 e 29), que são adicionadas ao Descriptor que está no topo da pilha – pois este é o Descriptor associado à composição que invocou aquele *web service*.

Já o método endBPEL (linha 36), como podemos imaginar, é executado toda vez que o InvokeBPEL está prestes a finalizar sua execução. O Descriptor é retirado da pilha, informações sobre as políticas associadas à composição e aos *web services* são impressas (linhas 39-42) e é checada a compatibilidade destas políticas logo em seguida (linha 44).

A checagem de compatibilidade é feita pela classe Descriptor (**Código 3.2**)

```
01   public boolean checkPolicyCompability() {
02     List<PolicyComponent> subComponents =
this.bpelPolicy.getPolicyComponents();
03     Set<Entry<String, Policy>> entrySet =
this.wsdlPolicies.entrySet();
04     boolean result = true;
05
06     for (PolicyComponent comp : subComponents) {
07       if (comp instanceof Assertion) {
08         Assertion ass = (Assertion) comp;
09         AssertionEvaluator eval =
AssertionEvaluatorFactory.createAssertionEvaluator(ass);
10         for (Entry<String, Policy> wsdlEntry : entrySet) {
11           Policy wsdlPolicy = wsdlEntry.getValue();
12           result &= eval.checkPolicy(wsdlPolicy);
13         }
14       } else {
15         //TODO Pra depois: validar outros tipos de operadores
16         result = false;
17       }
18     }
19
20     return result;
21   }
```

Código 3.2 - Trecho do código da classe Descriptor

O conjunto de subcomponentes da política do BPEL é iterado (linha 06), de forma que para cada subcomponente, é verificado com o AssertionEvaluatorFactory o avaliador correspondente para aquele subcomponente (linha 09). Para cada web service componente, é checado com o avaliador se aquele subcomponente da política é compatível (linha 12). Este método só retornará verdadeiro se somente se, para todos os subcomponentes da política, haja compatibilidade com a política dos *web services* participantes.

Há ainda uma restrição que pode ser observada num comando condicional IF (linha 07): o único tipo de subcomponente da política que está sendo suportado neste trabalho é Assertion.

A classe AssertionEvaluatorFactory possui apenas um método, como podemos ver no **Código 3.3**:

```
01 public class AssertionEvaluatorFactory {
02
03   private static final String PACKAGE_BASE =
"br.ufpe.cin.wspolicy.integration.assertions.element";
04
05   @SuppressWarnings("unchecked")
06   public static AssertionEvaluator
createAssertionEvaluator(Assertion ass) {
07     AssertionEvaluator result = new
AssertionEvaluator(ass.isOptional());
08
09     String name = ass.getName();
10     name = name.replace(":", ".");
11     try {
12       Class<AssertionEvaluator> clazz =
(Class<AssertionEvaluator>) Class.forName(PACKAGE_BASE + "."
+ name);
13       result = clazz.newInstance();
14     } catch (Exception e) {
15       System.out.println(e.getClass().getCanonicalName() + "
: " + e.getMessage());
16       System.out.println("Using default evaluator : " +
result.getClass().getCanonicalName());
17     }
18     result.setAssertion(ass);
19
20     return result;
21   }
22
23 }
```

Código 3.3 - Código-fonte da classe AssertionEvaluatorFactory

O método createAssertionEvaluator (linha 06) utiliza o conceito de reflexão para buscar no Assertion Evaluator Repository (vide seção 3.3) o avaliador correspondente a Assertion passada como parâmetro. Para isto, o nome do elemento XML é utilizado como nome da classe Java, e o *namespace* é utilizado como sub-pacote (linha 12). Caso o avaliador não seja encontrado, significa que para aquela Assertion não foi implementado um avaliador específico, e como plano de contingência, um avaliador padrão deverá ser utilizado (linha 16).

O avaliador padrão é implementado pela classe AssertionEvaluator, mostrada no **Código 3.4**.

```
01 public boolean checkPolicy(Policy policy) {
02   List<PolicyComponent> policyComponents =
   policy.getPolicyComponents();
03   boolean compatible = false;
04   boolean found = false;
05
06   for (PolicyComponent policyComponent : policyComponents) {
07     if (policyComponent instanceof Assertion) {
08       Assertion ass = (Assertion) policyComponent;
09       if (this.assertion.getName().equals(ass.getName())) {
10         found = true;
11         compatible = checkAssertion(ass);
12         break;
13       }
14     }
15   }
16
17   if (!found) {
18     compatible = this.canBeEmpty;
19   }
20
21   return compatible;
22 }
```

Código 3.4 - Trecho do código da classe AssertionEvaluator

Esta classe é a superclasse de todos os avaliadores, ou seja, quem desejar criar um avaliador para uma Assertion em específico, deverá estender esta classe, reimplementando o método checkAssertion. O tratamento do avaliador padrão consiste em fazer uma comparação sintática das asserções. Se forem idênticas, não há motivo para crer que não sejam compatíveis.

O método checkPolicy (linha 01) compara a Assertion representada pelo avaliador com cada subcomponente da política passada por parâmetro (linha 06). Aqui, o comportamento é similar ao da classe Descriptor: a política da Assertion só é compatível com a do web service se dentre os subcomponentes for encontrado uma Assertion que seja do mesmo tipo da primeira e estas asserções forem compatíveis (linhas 07-11).

Caso não seja encontrada nenhuma Assertion nos subcomponentes da política do *web service*, um atributo da classe indicará o que fazer: canBeEmpty (linha 18). Este atributo é útil em casos onde uma Assertion é opcional, ou sua restrição associada não chegue a comprometer a compatibilidade entre as políticas.

Para efeitos de demonstração do funcionamento da proposta, um avaliador foi implementado, o ExecTime (**Código 3.5**).

```
01   private enum Operator {
02     LESS_THAN, LESS_OR_EQUALS_THAN, EQUALS, GREATER_THAN,
GREATER_OR_EQUALS_THAN;
03
04     public static Operator fromString(String str) {
05       if (str.equals("<")) return LESS_THAN;
06       if (str.equals("<=")) return LESS_OR_EQUALS_THAN;
07       if (str.equals("=")) return EQUALS;
08       if (str.equals(">")) return GREATER_THAN;
09       if (str.equals(">=")) return GREATER_OR_EQUALS_THAN;
10       return null;
11     }
12   }
13
14   public boolean checkAssertion(Assertion assertion) {
15     boolean fullCompare = super.checkAssertion(assertion);
16     if (fullCompare) return true;
17
18     Operator myComp =
Operator.fromString(getAssertion().getAttributeValue("operator"));
19     Operator otherComp =
Operator.fromString(assertion.getAttributeValue("operator"));
20     Integer myValue =
Integer.parseInt(getAssertion().getAttributeValue("milliseconds"));
21     Integer otherValue =
Integer.parseInt(assertion.getAttributeValue("milliseconds"));
22
23     boolean result = false;
24     if (myComp == Operator.EQUALS && otherComp == Operator.EQUALS)
```

```
    {
25        result = (myValue == otherValue);
26      } else if ((myComp == Operator.LESS_THAN || myComp ==
    Operator.LESS_OR_EQUALS_THAN) &&
            ((otherComp == Operator.EQUALS) || (otherComp ==
27  Operator.LESS_OR_EQUALS_THAN) || (otherComp ==
    Operator.LESS_THAN))) {
28        result = (myComp == Operator.LESS_THAN || otherComp ==
    Operator.LESS_THAN) ? myValue > otherValue : myValue >= otherValue;
29      } else  if ((myComp == Operator.GREATER_THAN || myComp ==
    Operator.GREATER_OR_EQUALS_THAN) &&
            ((otherComp == Operator.EQUALS) || (otherComp ==
30  Operator.GREATER_OR_EQUALS_THAN) || (otherComp ==
    Operator.GREATER_THAN))) {
31        result = (myComp == Operator.GREATER_THAN || otherComp ==
    Operator.GREATER_THAN) ? myValue < otherValue : myValue <=
    otherValue;
32      }
33
34      return result;
35    }
```

Código 3.5 - Trecho do código da classe ExecTime

Este avaliador foi feito para validar uma Assertion do tipo <perf:ExecTime>. Como o nome já pode dar uma idéia, esta Assertion simples atribui uma característica de performance ao *web service*/composição, recebendo dois parâmetros: *millisseconds*, o valor de tempo, e *operator*, que pode ser um destes: >, <, =, <= ou >=. Por exemplo, a Assertion <perf:ExecTime millisseconds="1000" operator="<" /> indica que o tempo de execução da entidade associada (web service ou composição) deverá ser menor que 1 segundo.

Analisando o código, vemos que nele só há algumas comparações de qual operador está na asserção, e para cada tipo de operador, algumas comparações são feitas entre os valores em milissegundos (linha 24-31).

3.5.2 WS-PolicyImpl

O componente WS-PolicyImpl é a implementação da especificação WS-Policy 1.5 em Java [17]. Além do modelo sugerido pela especificação, e apresentado na seção 3.4, alguns ajustes de implementação foram feitos para conseguir criar políticas corretamente.

O principal desafio de implementação para este componente foi a leitura de documentos XML genéricos em busca de políticas. O **Código 3.6** da classe Policy mostra um leitor de XML.

```
01   private static void createCompFromElement(PolicyOperator
parent, Element rootElement) {
02     PolicyComponent comp = null;
03     String qualifiedName = rootElement.getQualifiedName();
04     List<?> elements = rootElement.elements();
05
06     if (qualifiedName.equalsIgnoreCase("wsp:Policy")) {
07       comp = new Policy();
08       for (Object element : elements) {
09         createCompFromElement((PolicyOperator) comp,
(Element) element);
10       }
11       List<?> attributes = rootElement.attributes();
12       for (Object attribute : attributes) {
13         Attribute atr = (Attribute) attribute;
14         ((Policy) comp).addAttribute(atr.getName(),
atr.getValue());
15       }
16     } else if (qualifiedName.equalsIgnoreCase("wsp:All")) {
17       comp = new All();
18       for (Object element : elements) {
19         createCompFromElement((PolicyOperator) comp,
(Element) element);
20       }
21     } else if
(qualifiedName.equalsIgnoreCase("wsp:ExactlyOne")) {
22       comp = new ExactlyOne();
23       for (Object element : elements) {
24         createCompFromElement((PolicyOperator) comp,
(Element) element);
25       }
26     } else if
(qualifiedName.equalsIgnoreCase("wsp:PolicyReference")) {
27       PolicyReference ref = new PolicyReference();
28       String attributeValue =
rootElement.attributeValue("URI");
29       if (attributeValue == null) {
30         return;
31       }
32       ref.setURI(attributeValue);
33       comp = ref;
```

```
34      } else {
35        Assertion assertion = new Assertion();
36        assertion.setValue(rootElement);
37        comp = assertion;
38      }
39      parent.addPolicyComponent(comp);
40    }
```

Código 3.6 - Trecho de código da classe Policy

Este leitor utiliza-se fortemente de recursão para checar cada tipo de operador (linhas 09, 19 e 24), além de manter uma referência para o nó pai (linha 39).

3.5.3 Integração com a *engine* ActiveBPEL

Uma boa parte do planejamento da proposta esteve em torno de como seria a integração com a engine ActiveBPEL. Felizmente, a arquitetura da própria engine ajudou, e apenas duas classes foram alteradas: AeBpelHandler, responsável por invocar a composição e AeAxisInvokeHandler, responsável por invocar os *web services*.

A classe AeAxisInvokeHandler teve uma linha de código alterada, com a inclusão da chamada ao método wsdlHandler, e a classe AeBpelHandler teve duas linhas de código alteradas: no ínicio do método invoke, uma chamada ao método startBPEL, e outra ao final do invoke, com uma chamada ao método endBPEL. Todas as chamadas eram feitas à classe WSPolicyAdapter.

4 Exemplo: Calculadora

Neste capítulo será apresentado um exemplo de utilização da engine ActiveBPEL, já com o conceito de políticas incluso. Este exemplo servirá não apenas para demonstrar como a engine funciona, mas também para validar a implementação desta proposta.

Inicialmente teremos uma breve descrição do exemplo, e quais os tipos de operação ele suporta. Logo em seguida, será mostrada a composição utilizada para construir o exemplo. No tópico seguinte, o passo-a-passo de alguns exemplos de execução será comentado, e por fim teremos uma breve análise de como este exemplo interagiu com a implementação proposta.

4.1 Descrição do Exemplo

O exemplo utilizado para testar a implementação da proposta foi o da Calculadora Remota. É um exemplo conceitualmente muito simples, fácil de entender e implementar, porém suficiente para mostrar os conceitos de políticas, e dessa forma, validar a proposta.

A Calculadora representa uma máquina de calcular simples, com cinco operações básicas: soma, subtração, multiplicação, divisão e raiz quadrada. Uma vez escolhida a operação a ser executada e informados os operandos, a calculadora escolhe a operação correta a ser chamada, executa esta operação, e informa o resultado do cálculo. Uma visão geral pode ser vista na **Figura 4.1**:

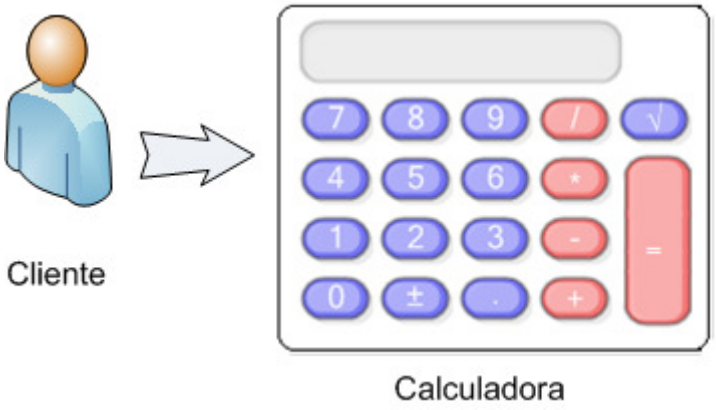

Figura 4.1- Visão geral da Calculadora

As operações binárias – aquelas que recebem dois operandos como parâmetros – são as operações de soma, subtração, multiplicação e divisão. A raiz quadrada é a única operação unária, portanto precisa apenas de um operando.

As operações de divisão e de raiz quadrada apresentam algumas peculiaridades que as torna interessantes para deixar o exemplo mais rico: a possibilidade de acontecer erros. A operação de divisão não suporta divisão por zero, e a operação de raiz quadrada exige que o radicando seja não-negativo.

A possibilidade de ocorrência destes erros sugere que seja feito um tratamento de falhas ao implementar o exemplo. Naturalmente, as operações de soma, subtração e multiplicação não são passíveis de erros, logo não necessitam de tratamento.

4.2 Composição da Calculadora

Como vimos na seção anterior, a Calculadora recebe o tipo de operação e os operandos, e procura qual operação corresponde àquele tipo. Na **Figura 4.2** temos a composição criada para a Calculadora:

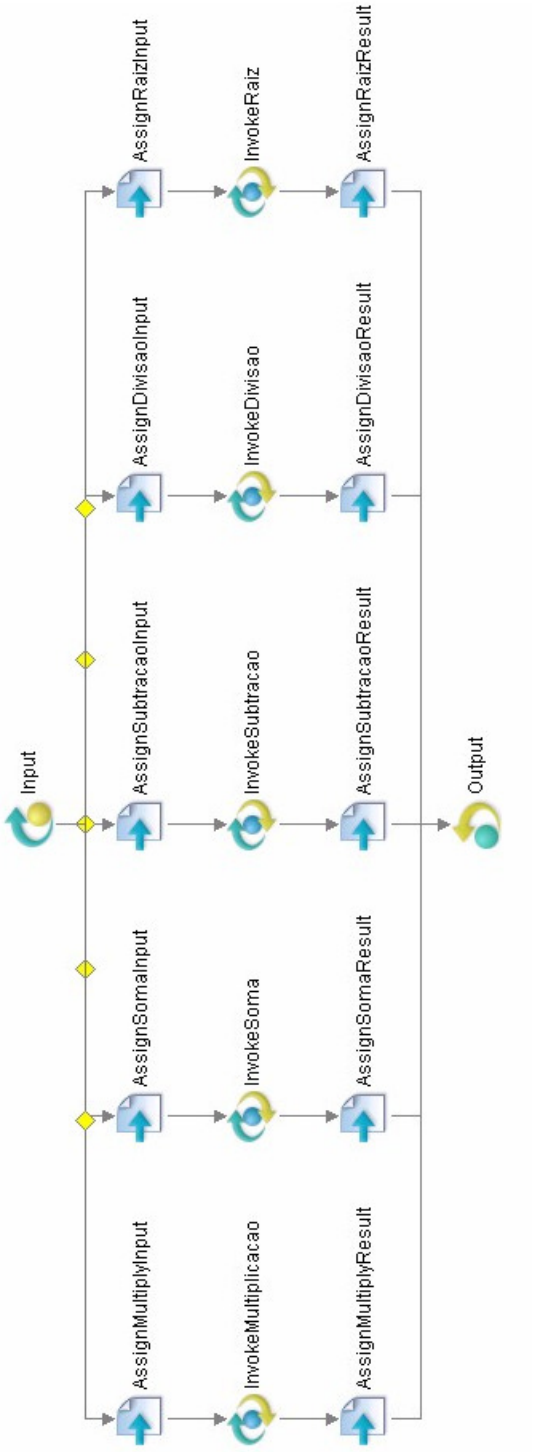

Figura 4.2 - Composição da Calculadora

Esta composição possui quatro tipos distintos de atividades:

- **Receive** – Responsável por receber o tipo de operação e os operandos quando a composição é invocada; a única atividade do tipo Receive que temos é a atividade Input;

- **Invoke** – Responsável por fazer uma chamada a um *web service* componente; existem cinco atividades do tipo Invoke, uma para cada operação aritmética;

- **Assign** – Copia o resultado de uma variável para outra, fazendo adaptações nos parâmetros se necessário; existem duas atividades de Assign para cada operação: uma que copia da variável de entrada para o parâmetro de entrada do *web service* componente e outra que copia a resposta do *web service* para a variável de saída da composição;

- **Reply** – Responsável por informar uma resposta a alguma entidade; a única atividade do tipo Reply que temos é a atividade Output, responsável por informar o resultado.

Além das atividades listadas, existem cinco links que merecem uma melhor explicação: aqueles que ligam a atividade Input com o primeiro Assign de cada operação. Estes links possuem uma propriedade chamada "Transition Condition", que indica por qual caminho o fluxo de execução deverá seguir.

A Transition Condition de cada link variou de acordo com a operação, um pequeno exemplo é mostrado em **Código 4.1**:

```
$executeRequest/calcmsg:type = '*'
```

Código 4.1 - Transition Condition para a operação de multiplicação

As condições de transição para as outras operações são semelhantes, e não serão listadas uma a uma.

4.3 Execução da Calculadora

Uma vez que a composição foi definida, agora já podemos executá-la. Para isso, precisamos verificar no seu descritor WSDL qual a operação que faz isto, e quais os parâmetros que esta operação recebe.

```
01  <wsdl:portType name="Calculadora">
02    <wsdl:operation name="execute">
03      <wsdl:input message="calc:executeRequest" />
04      <wsdl:output message="calc:executeResponse" />
05      <wsdl:fault message="calc:executeFault"
06        name="errorHandling" />
07    </wsdl:operation>
08  </wsdl:portType>
```

Código 4.2 - Trecho do descritor WSDL da composição Calculadora

No **Código 4.2**, vemos que a operação da composição chama-se execute (linha 02), e possui um parâmetro de entrada (linha 03), um parâmetro de saída (linha 04) e um parâmetro de tratamento de falhas (linha 04).

Cada parâmetro está associado a uma mensagem que também foi definida no WSDL da Calculadora. Para não poluir esta seção do documento, serão anexados ao fim desta proposta os códigos utilizados no exemplo.

Utilizando um aplicativo cliente de *web services* (existem muitos, porém para esta execução o próprio Web Services Explorer embutido na IDE Eclipse foi utilizado [6]), a mensagem SOAP que está em **Código 4.3** é enviada à engine:

```
01  <soapenv:Envelope>
02    <soapenv:Body>
03      <CalculatorRequest>
```

```
04        <type>^</type>
05        <operand1>15</operand1>
06        <operand2>0</operand2>
07        <operand3>0</operand3>
08        <operand4>0</operand4>
09      </CalculatorRequest>
10    </soapenv:Body>
11  </soapenv:Envelope>
```

Código 4.3 - Mensagem SOAP de envio

Nesta mensagem, vemos o elemento CalculatorRequest (linha 03), composto pelo tipo da operação (linha 04), e pelos operandos (linhas 05-08). Neste exemplo, estamos querendo descobrir qual é a raiz quadrada (tipo) do número 15 (operand1).

Observe que o elemento CalculatorRequest possui até quatro operadores; ele foi definido dessa forma para suportar diversos tipos de operações; se a operação a ser invocada utilizar menos operandos que o necessário, os outros são ignorados.

A engine recebe a requisição, executa a composição, e logo em seguida responde com a mensagem SOAP que está em **Código 4.4**:

```
01  <soapenv:Envelope >
02    <soapenv:Body>
03      <CalculatorResponse>
04        <result>3.872983346207417</result>
05      </CalculatorResponse>
06    </soapenv:Body>
07  </soapenv:Envelope>
```

Código 4.4 - Mensagem SOAP de resposta

Na mensagem de resposta temos o elemento CalculatorResponse (linha 03), que contém apenas o atributo result (linha04). Este atributo contém o resultado da composição.

Na **Figura 4.3** vemos como ficou o fluxo de execução da composição:

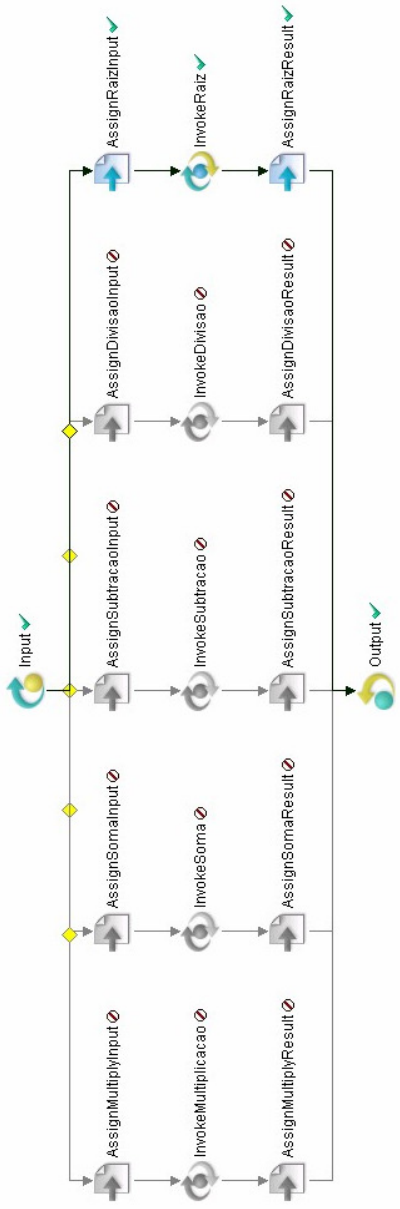

Figura 4.3 - Fluxo de execução da composição Calculadora (operação raiz quadrada)

4.4 Análise da Calculadora

Na seção anterior vimos apenas a composição Calculadora em execução, porém não explicitamos em nenhum momento quais políticas deveriam ser associadas a ela. Para fazer isto, devemos incluir informações sobre as políticas nos descritores WSDL da composição e de cada *web service*.

Para validar este trabalho, além da criação da composição Calculadora, políticas foram atribuídas à composição e ao web service que calcula raiz quadrada.

A política atribuída a cada um deles foi bem simples. O requisito não funcional performance foi escolhido por ter uma boa relação com o exemplo. A propriedade escolhida neste requisito foi o tempo de execução. Vale lembrar que este tempo de execução não é mensurado cada vez que a composição e/ou *web service* é invocado; esta informação aparece apenas da forma descritiva.

Ao executar novamente a composição, agora com os descritores WSDL atualizados para incluir as políticas, vemos que a mensagem SOAP de resposta que a engine fornece não mudou. Isso acontece porque esta proposta do trabalho não alterou o funcionamento da engine, pois isso acarretaria incompatibilidade por parte das composições que não desejam ter políticas.

Ao invés de alterar as mensagens SOAP de retorno, o componente responsável por verificar as políticas exibe estas informações no console do servidor (**Código 4.5**):

```
01 startBPEL - CalculatorLTService
02
03 wsdlHandler - http://localhost:8080/CalculatorOperations/services/RaizPort?wsdl
04
05 endBPEL - CalculatorLTService
06
07 Foram encontradas as seguintes políticas no BPEL:
08   <wsp:Policy>
09     <perf:ExecTime milliseconds="1000" operator="<"/>
10   </wsp:Policy>
11
```

```
12 Foram encontradas as seguintes políticas nos WS:
13    =>
http://localhost:8080/CalculatorOperations/services/RaizPort?wsdl
14       <wsp:Policy>
15         <perf:ExecTime milliseconds="600" operator="="/>
16       </wsp:Policy>
17
18 As políticas do BPEL e do WS são compatíveis
```

Código 4.5 - Mensagens exibidas pelo componente PolicyAdapter

Durante a execução da composição, o componente PolicyAdapter encontra uma política no BPEL (linhas 07-10), uma política no *web service* que calcula raiz quadrada (linhas 13-16), e as compara. Na linha 18, uma mensagem de confirmação é exibida: as políticas são compatíveis.

Analisando as políticas que foram estabelecidas, vemos que a composição indica que seu tempo de execução deve ser inferior a 1000 milissegundos (linha 09), e que o *web service* que calcula raiz quadrada indica que seu tempo de execução é igual a 600 milissegundos. Assim percebemos que as políticas realmente são compatíveis.

5 Conclusões e Trabalhos Futuros

Este capítulo contém as conclusões e contribuições deste trabalho, além das vantagens, limitações e trabalhos futuros.

5.1 Conclusões

Atualmente, o principal foco dos serviços ainda está muito atrelado aos requisitos funcionais: “quais os parâmetros que este serviço recebe?”, “o que ele oferece como resposta?”, “o que ele faz?”. Para alguns serviços, responder a estes tipos de perguntas não é o suficiente.

A inclusão de restrições associadas a serviços ainda é um problema em aberto, e é necessária quando se deseja agregar requisitos não-funcionais a estes serviços. Estes requisitos não-funcionais expressam um desejo, ou ainda uma limitação: “este serviço é seguro?”, “este serviço executa em menos do que X milissegundos?”.

A proposta deste trabalho foi incluir numa *engine* de composição de *web services* o conceito de políticas. Políticas são formas de representar estes requisitos não-funcionais, e dessa forma, vão de encontro com as perguntas feitas no parágrafo anterior.

A principal contribuição deste trabalho, portanto, diz respeito ao tratamento de políticas, não obrigatoriamente associadas à *web services*, como foi o foco deste trabalho, mas possivelmente extensível a outras áreas do conhecimento.

Através dos testes efetuados, a implementação da proposta foi de acordo com as expectativas do trabalho, respondendo satisfatoriamente inclusive em casos onde as políticas eram compostas por mais de um asserção, ou ainda

quando a composição, em seu fluxo de execução, invocava mais de um *web service*., sendo necessário comparar as políticas de cada *web service* mais de uma vez com as da composição.

No entanto, algumas limitações foram encontradas nesta solução, tais como:

- Os operadores de políticas, All e ExactlyOne, apesar de implementados de acordo com a especificação, não puderam ser utilizados nas políticas devido a um maior grau de complexidade;
- Um cliente que execute a composição a priori não consegue entender se as políticas foram compatíveis, pois as mensagens que ele recebe da composição nada informam a este respeito.

5.2 Trabalhos futuros

Como trabalhos futuros podem estar relacionados:

- **Pesquisa por novas especificações de políticas que representem bem os requisitos não-funcionais** – A criação dessas especificações está numa fase muito embrionária, então as políticas ainda são definidas de uma forma *ad hoc*, sem padronização;
- **Implementação da função normalize da especificação WS-Policy** – A função normalize desta especificação compacta uma política, deixando-a mais simples. Contudo, sua implementação é não-trivial, e acabou não entrando no escopo desta proposta. O uso de políticas

normalizadas economiza espaço no descritor WSDL e facilita a leitura por parte do PolicyAdapter;

- **Utilização de um avaliador de lógica de primeira ordem** – Uma vez que os operadores All e ExactlyOne não foram utilizados devido à complexidade de estruturá-los dinamicamente num código, um avaliador de lógica de primeira ordem poderia ser utilizado para fazer comparações entre a estrutura de políticas cada vez mais complexas;

- **Transformação da implementação da proposta em API** – Apesar da *engine* escolhida ser código aberto, a implementação da proposta ainda está muito dependente da *engine* na qual foi criada. Transformá-la numa API a parte seria interessante para uma maior aceitação na comunidade, e para dar-lhe maior flexibilidade na hora de escolher com qual *engine* de composição de web services ela irá interagir.

Outros trabalhos futuros podem estar relacionados indiretamente, por exemplo, criação de novas *engines* para composição.

Referências Bibliográficas

[1] ActiveBPEL, L.L.C. "The Open Source BPEL Engine", disponível em http://www.activebpel.org, acessado em 20/11/2008

[2] Alonso, G. et al, (2003) "Web Services: Concepts, Architectures and Applications", Springer-Verlag, Alemanha.

[3] Andrews, T. et al. "Business Process Execution Language for Web Services, version 1.1", disponível em http://www.ibm.com/developerworks/library/ws-bpel, acessado em 20/11/2008

[4] Apache Software Foundation. “Apache ODE”, disponível em http://ode.apache.org, acessado em 20/11/2008

[5] Apache Software Foundation. “Apache Tomcat”, disponível em http://tomcat.apache.org, acessado em 20/11/2008

[6] Eclipse Foundation. “Eclipse IDE”, disponível em: http://www.eclipse.org, acessado em 20/11/2008

[7] Lins, F. A. A. (2007), “Composição Adaptativa de Web Services”, Dissertação de Mestrado, Universidade Federal de Pernambuco.

[8] Liskov, B. (1987), “Keynote address - data abstraction and hierarchy”, in 'OOPSLA '87: Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)', ACM, New York, NY, USA, pp. 17--34.

[9] OASIS. “UDDI Version 3.0.2”, disponível em http://uddi.org/pubs/uddi-v3.0.2-20041019.htm, acessado em 20/11/2008

[10] OW2 Consortium. “Orchestra: Open Source BPEL/BPM Solution”, disponível em http://orchestra.objectweb.org, acessado em 20/11/2008

[11] Tai, S.; Khalaf, R. & Mikalsen, T. (2004), “Composition of coordinated web services”, em 'Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware', Springer-Verlag New York, Inc., New York, NY, USA, pp. 294--310.

[12] Tim Bray. “SOA too complex: ‘just vender BS’”, disponível em http://blogs.zdnet.com/service-oriented/?p=597, acessado em 20/11/2008

[13] W3C. “Simple Object Access Protocol (SOAP) 1.1”, disponível em http://www.w3.org/TR/soap, acessado em 20/11/2008

[14] W3C. “Web Services Description Language (WSDL) 1.1”, disponível em http://www.w3.org/TR/wsdl, acessado em 20/11/2008

[15] W3C. “Web Services Glossary”, disponível em http://www.w3.org/TR/ws-gloss, acessado em 20/11/2008

[16] W3C. “Web Services Policy 1.5 – Attachment”, disponível em http://www.w3.org/TR/ws-policy-attach, acessado em 20/11/2008

[17] W3C. “Web Services Policy 1.5 – Framework”, disponível em http://www.w3.org/TR/ws-policy, acessado em 20/11/2008

Data e assinaturas

Recife – PE, 15 de dezembro de 2008

_____

Bruno Leonardo Barros Silva
(Aluno)

_____

Nelson Souto Rosa
(Orientador)