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

UM AMBIENTE DE DESENVOLVIMENTO DE COMPOSIÇÕES DE SERVIÇOS BASEADO EM MODELOS

TRABALHO DE GRADUAÇÃO

Aluno: André Ricardo da Silva Souza (arss@cin.ufpe.br)
Orientador: Nelson Souto Rosa (nsr@cin.ufpe.br)

DEZEMBRO DE 2008

Agradecimentos

Primeiramente eu agradeço a minha família. Sem ela eu não chegaria tão longe. Agradeço todo apoio e compreensão, pelas caronas na madrugada quando eu tinha que ficar até tarde no CIn fazendo projetos; por agüentar o meu stress em tempos de fim de período e na ultima semana dessa Monografia; por tudo.

Agradeço também aos meus amigos do CIn que, junto comigo, passaram por muita coisa durante essa graduação. Amigos que se reunião pra fazer projetos, geralmente alguns dias antes da entrega e sem restrições quanto a ser parte de um grupo ou outro. Amigos que se reúnem em fins de semana pra tentar de divertir um pouco. Amigos com quem, em certos momentos, tinha uma convivência maior que com a minha família. Dentre esses eu não poderia deixar de citar alguns: Cabeça (Bruno Barros), um cara que se parece mais comigo que o meu próprio irmão; Rebeka Gome, uma menina chata que gosta de me “arretar” a paciência; e Bill Gates (Christian Diego), meu fã numero 1. Não esquecendo também dos meus amigos da Improvess e os famosos *Threadianos*.

Por último, mas não menos importante, agradeço aos professores e funcionário do Centro de Informática que deram todo o suporte para a minha graduação.

Resumo

A composição de serviços é uma das áreas de maior interesse de pesquisa da atualidade. A motivação geral para composição de serviços está relacionada a dois pontos principais: possibilidade de criação de serviços mais elaborados a partir de serviços já existentes e a possibilidade de integração de negócios. Na prática, a composição permite que serviços de diferentes provedores sejam colocados juntos para criar serviços mais sofisticados. A importância da composição de serviços tem sido reconhecida na comunidade de pesquisa da Internet devido a sua flexibilidade na construção de aplicações a partir de serviços primitivos e de uma forma *plug-and-play* .

O objetivo desse trabalho é propor um ambiente para a composição de serviços faça uso de modelos de serviço para abordar duas áreas de pesquisa de SOA: descrição/execução da composição e a evolução dessa composição durante o seu ciclo de vida.

Sumário

Agradecimentos ..... **II**

Resumo ..... **III**

1 Introdução 7

2 Conceitos Básicos **10**

2.1 SOA (Service-Oriented Architecture) 10

2.1.1 SOAP 10

2.1.2 WSDL 11

2.2 Composição de Serviços 11

2.2.1 BPEL 12

3 Um ambiente para Composição de Serviços **13**

3.1 Visão Geral 13

3.2 Requisitos do Toolkit 13

3.2.1 Prioridade dos Requisitos 13

3.2.2 Requisitos Relacionados à Modelagem/Execução 14

3.2.3 Requisitos Relacionados à Evolução Composicional 15

3.2.4 Requisitos não Funcionais 17

3.3 Arquitetura 17

3.3.1 MoSC Repository 17

3.3.2 MoSC Modeling Tool 17

3.3.3 MoSC Auxiliary Engine 19

3.4 Projeto 21

3.4.1 Componentes Gráficos do MoSC Modeling Tool 21

3.4.2 Requisitos e Propriedades não Funcionais 24

3.4.3 Variáveis 27

3.4.4 MoSC-ML Transformer 29

3.4.5 Deployer 30

3.5 Implementação 30

3.5.1 MoSC Modeling Tool 30

3.6 Considerações Finais 40

4 Estado da Arte 40

4.1 Ferramentas para composição de service 46

5 Conclusões e Trabalhos Futuros 52

6 Referências Bibliográficas 54

Data e assinaturas 57

Lista de Figuras

Figura 3.1 - Arquitetura do MoSC Modeling Tool.....18

Figura 3.2 - Arquitetura do MoSC Auxiliary Engine..... 20

Figura 3.3 - Modelo de uma composição para o Ambiente Gráfico 22

Figura 3.4 - Modelagem dos Requisitos não Funcionais 26

Figura 3.5 - Modelagem das Variáveis 28

Figura 3.6 - MoSC Modeling Tool 31

Figura 3.7 - Define Interface..... 33

Figura 3.8 - Janela “Define Interface”..... 33

Figura 3.9 - Escolhendo a opção “Define Operation” 34

Figura 3.10 – Define Operation..... 35

Figura 3.11 – Variable Association 36

Figure 3.12 – Variable Association..... 36

Figura 3.13 - Botão para editar os Requisitos não funcionais da Atividade.... 37

Figura 3.14 - Editar as Propriedades não funcionais da composição..... 37

Figura 3.15 - Adicionando Requisitos não Funcionais a uma Activity 38

Figura 3.16 - Arquivo MoSC-ML gerado 39

Figura 4.1 - Visão geral da arquitetura Picasso.....41

1 Introdução

Computação Orientada a Serviços (*Service Oriented Computing)* e SOA (*Service Oriented Architecture*) [22][31] são conceitos importantes na área de sistemas distribuídos que têm sido amplamente utilizados em projetos e pesquisas relacionadas a serviços [17]. Em SOC/SOA, a funcionalidade da aplicação é descrita de tal forma que permite seu uso no desenvolvimento de aplicações que integram funcionalidades de outras aplicações, o que é conhecido como composição de serviços ou serviços compostos. Cada serviço encapsula a funcionalidade da aplicação tornando-a disponível através de uma interface. Além disto, serviços são normalmente autônomos e heterogêneos no sentido de que executam sobre diferentes plataformas e pertencem a diferentes organizações.

SOC e SOA têm tido um papel importante no modelo de negócio de software conhecido como SaaS (*Service as Software*). SaaS é um modelo de disponibilização de aplicações no qual as empresas (fornecedores de software) desenvolvem, hospedam, operam e fornecem o software a outras empresas (consumidoras) que pagam para usar a aplicação e não para possuí-las. Na prática, as empresas consumidoras alugam uma funcionalidade que é disponibilizada remotamente, por exemplo, através da Internet.

A composição de serviços é uma das áreas de maior interesse de pesquisa da atualidade. A motivação geral para composição de serviços está relacionada a dois pontos principais: possibilidade de criação de serviços mais elaborados a partir de serviços já existentes e a possibilidade de integração de negócios [32][33][34]. Na prática, a composição permite que serviços de diferentes provedores sejam colocados juntos para criar serviços mais sofisticados. A importância da composição de serviços tem sido reconhecida na comunidade de pesquisa da Internet devido a sua flexibilidade na construção de aplicações a partir de serviços primitivos e de uma forma *plug-and-play* [35][36][37][38].

Muitas empresas têm disponibilizado suas principais competências na Internet como uma coleção de serviços conhecidos como *Web Services*. Web *Services* de organizações diferentes ou da mesma organização são combinados para formar o que é conhecido como serviço *Business-to-Business* (B2B).

Colaborações B2B comumente demandam interações de longa duração guiadas por um modelo de processo explícito. Por exemplo, um serviço de gerenciamento financeiro pode ser criado compondo serviços mais especializados para pagamentos, cálculo de taxas e gerenciamento de moeda.

No contexto da composição de serviços, é crescente também o interesse pela adoção de modelos para expressar a composição, como uma evolução as abordagens tradicionais baseadas em BPEL4WS (*Business Process Language for Web Services*). O uso de modelos representa um importante avanço na área de composição, pois permite a descrição de informações mais elaboradas sobre o comportamento, estado, carga, conectividade e configuração , dos serviços [16][39]. Desta forma, a composição passa a ser modelada em um nível de abstração maior, permitindo o tratamento de interações mais complexas e maior rapidez no projeto de composição dos serviços.

Para tratar com novas demandas, a composição de serviços precisa evoluir continuamente (evolução composicional) por duas razões principais. Primeiro, quando uma nova funcionalidade não implementada pelos serviços existentes é necessária, serviços mais básicos podem ser combinados para implementá-la. Segundo, a composição pode evoluir para prover uma funcionalidade com algumas qualidades desejadas Além disto, a evolução pode ser estática ou dinâmica e pode se referir a mudanças estruturais (mudanças na topologia da composição), mudanças geográficas (mudanças na localização dos serviços), mudanças funcionais (mudança na assinatura dos serviços) e mudanças na implementação [36][40][41][42].

Outra razão, que também será abordada nesse trabalho, diz respeito a requisitos não funcionais de cada composição pode ter em relação aos serviços que lhe compõe. Caso algum desses requisitos começar a não ser atendido uma evolução composicional pode ser necessária.

O que se observa é que cada um dos trabalhos analisados foca em alguns dos pontos relevantes desse documento, mas não em todos. Esse documento propõe a criação de um *Toolkit* com três entidades: *MoSC Modeling Tool*, um ambiente gráfico de modelagem usando modelos; o MoSC *Repository* um repositório de serviços que fornece buscas avançadas através de informações agora relevantes como: Semântica e Requisitos não

Funcionais; e o *MoSC Auxiliary Engine*, uma entidade que abstrai qual tipo de *Engine* real está sendo usada e gerencia as questões evolucionais da composição.

Além deste capítulo de introdução, o presente trabalho inclui ainda mais cinco capítulos, como descrito a seguir:

Capítulo 2: Neste capítulo serão introduzidos todos os conceitos básicos associados a este trabalho. Os aspectos relacionados à arquitetura orientada a serviços (SOA), aos *Web Services* e a composição de serviços serão detalhados. Adicionalmente, alguns conceitos de MDA serão discutidos nesse capítulo.

Capítulo 3: Este capítulo apresenta a abordagem proposta para incluir a questão de modelos nas composições de serviços.

Capítulo 4: Nesse ponto, uma discussão sobre os trabalhos relacionados a este trabalho é apresentada. Será apresentada uma série de abordagens para resolver tanto questões sobre a inclusão de modelos na Composição de Serviços quanto para a questão da Evolução Composicional.

Capítulo 5: Por fim, esse capítulo apresenta as conclusões e os próximos passos dessa pesquisa.

2 Conceitos Básicos

Nesse capítulo serão introduzidos os conceitos e tecnologias básicos para o entendimento do restante desse trabalho. Inicialmente será dada uma visão geral sobre a arquitetura orientada a serviços, logo em seguida será abordada as tecnologias relacionadas.

2.1 SOA (*Service-Oriented Architecture*)

Muitas empresas estão implantando Sistemas de Informação e muitas vezes esses o aspecto de integração com outros sistemas vem a ser um dos requisitos mais importante [43]. Mas como prover, ou garantir, fácil integração com outros sistemas? Pensando em problemas como esse que a arquiteturas orientadas a serviço estão sendo propostas.

Em SOA, a unidade mais básica é o Serviço, que são entidades que provem alguma funcionalidade. A partir desses serviços e de suas interfaces novos serviços podem ser construídos. Como se pode reparar, as palavras chave para esse tipo de arquitetura são: integração e reuso [17][22].

Web Services são um exemplo de como essa arquitetura está se difundindo rapidamente. E isso se deve ao fato dessa tecnologia usar de soluções simples para resolver problemas relacionados à integração entre aplicações. *Web Services* usam protocolos baseados em XML (SOAP[45], WSDL[46]) sobre HTTP pra realizar a comunicação. Isso garante total independência de Sistemas Operacionais e Linguagens de Programação. E, por se tratar de um serviço, essa entidade mantém um baixo acoplamento com outros serviços e aplicações. As tecnologias relacionadas com serviços e *Web Services*, instâncias mais populares de serviços, serão abordadas nas próximas seções.

2.1.1 SOAP

SOAP (*Simple Object Access Protocol*) [45] é um protocolo que se difundiu na internet rapidamente pela sua simplicidade e por mais duas razões simples [43]:

1. SOAP pode ser ter a questão dos tipos abstraída, ficando os programadores sendo os responsáveis por tratar as conversões de tipos. Assim, o protocolo trabalha apenas com strings dentro de usa estrutura de comunicação, chamada de Envelope. Esse fato resolve problemas de interoperabilidade entre arquiteturas de sistemas, pois SOAP trabalha pode trabalhar apenas com o mais comum dos tipos de dados que é string.
2. SOAP pode ser usado sobre quaisquer protocolos, assim, sobre HTTP esse protocolo pode ser usado em qualquer servidor pois não tem problemas de segurança com Firewalls, que em geral, liberam o tráfego na porta 80 do HTTP.

2.1.2 WSDL

WSDL (*Web Service Description Language*)[46] é a forma padrão de se descrever *Web Services*. WSDL é um documento baseado em XML que prove todas as informações sobre a interface de um serviço. Dentre elas, tem-se: Informações sobre protocolos de acesso, endereço de servidor que prove o serviço, interfaces das operações disponíveis nesse serviço e definição dos tipos usados nas operações desse serviço.

2.2 Composição de Serviços

Esse é o principal conceito para esse trabalho, a composição de serviços vem para fornecer base para a arquitetura SOA pelo fato que é através desse conceito que serviços são unidos e trabalham em coordenadamente para a realização de uma tarefa mais complexa.

De fato, a criação de uma aplicação de forma orientada a serviços não difere tanto sobre programação imperativa, por exemplo. Exceto pelo fato de ser estar trabalhando essencialmente de forma distribuída e que a principal linguagem de criação de composição de serviços é baseada em XML. Essa linguagem se chama BPEL [44] e vem sendo largamente utilizada pela indústria para essa função de descrição de composições de serviços. Na próxima seção serão discutidos em detalhes os aspectos dessa linguagem.

2.2.1 BPEL

BPEL (*Business Process Execution Language)* [44] é, como o nome já diz, uma linguagem de execução de processos de negócio. Lins [43] define Processos de Negócio como uma coleção de invocações de serviços e atividades relacionadas que produzem um resultado de negócio, tanto para uma empresa quanto para diversas outras. A partir dessa definição pode-se perceber que BPEL foi criado em um contexto bem mais abrangente que o de compor *Web Services*.

Mesmo com esse overhead de complexidade na construção de composições, essa linguagem é a linguagem mais usada pela indústria para essa operação. Tornando-se praticamente um padrão para a composição de serviços ("praticamente", pois nenhum órgão chegou a definir BPEL realmente como um padrão para composição de *Web Services*).

Assim como WSDL, BPEL é baseada em XML, mas a relação entre as duas linguagens é bem maior. Como é no WSDL que se encontra a definição de interfaces e tipos, o BPEL da composição precisa necessariamente se relacionar com esses arquivos para poder validar as informações da descrição da composição. Outro aspecto importante é que uma composição se comporta de maneira semelhante à *Web Services* "simples", pois a interface da composição também é descrita em WSDL. Logo, a partir de um arquivo WSDL não é possível determinar se o WSDL pertence a um serviço "simples" ou a uma composição de serviços.

A sintaxe de BPEL chega a ser um pouco semelhante a alguma linguagem imperativa. De maneira semelhante, nela existem variáveis, estruturas condicionais, de repetição, de seqüencialidade, de paralelismo e muitas outras estruturas facilmente encontradas em qualquer linguagem de programação imperativa. Mas, ao invés de se ter funções ou objetos, temos chamadas a operações de serviços.

Ainda existem outras opções para composição de *Web Services* além de BPEL, mas ainda não alcançam a relevância que BPEL tem nessa área.

3 Um ambiente para Composição de Serviços

No capítulo anterior pode-se perceber uma grande quantidade de abordagens tanto para a questão da modelagem quanto para a questão da evolução composicional. No decorrer desse capítulo será descrito as principais fases do desenvolvimento de um *Toolkit* para a modelagem e evolução de serviços baseados em modelos desses serviços.

3.1 Visão Geral

Para abordar todos os aspectos que serão descritos mais a frente, estamos propondo um *Toolkit* com algumas ferramentas que, trabalhando de forma integrada irá prover para o usuário um modo ágil de desenvolvimento de composições baseado em modelos com suporte a evolução composicional.

3.2 Requisitos do Toolkit

Após toda analise do estado da arte da área de composição de serviços, foram elicitados os requisitos tanto com relação à modelagem quanto com relação à evolucional. Nas duas próximas seções serão detalhados esses requisitos individualmente.

3.2.1 Prioridade dos Requisitos

Os requisitos do sistema recebem uma priorização estabelecida de acordo com os elementos descritos a seguir. Isto é importante para definição de prioridades na fase de desenvolvimento.

- **Essencial**: requisito necessário para o funcionamento pleno do sistema;
- **Importante**: requisito que não interfere no funcionamento do sistema, mas sua ausência resulta num funcionamento não satisfatório;
- **Desejável**: requisito que não interfere no funcionamento básico do sistema. Pode ser implementado por último ou não implementado.

3.2.2 Requisitos Relacionados à Modelagem/Execução

Os requisitos relacionados à modelagem e execução de uma composição nessa proposta estão descritos abaixo:

[RF01] Modelagem baseada em Diagramas de Fluxo de UML

Para a modelagem gráfica das composições deve ser usado um diagrama baseado em Diagramas de fluxo, e deve possuir, pelo menos, quatro entidades básicas:

- Atividade: Entidade que representa a invocação de uma operação de um serviço;
- Estrutura Condicional: Entidade capaz de modelar uma escolha em alguma parte da composição referente ao status atual das variáveis da mesma;
- Estrutura de Repetição: Entidade capaz de definir a repetição de um determinado trecho da composição ate que um determinado estado das variáveis da composição seja alcançado.

Prioridade: Essencial.

[RF02] Independência de Linguagem de Execução

Seguindo a idéia proveniente do MDA com relação à PIM e PSM, a modelagem, ou seja, a descrição da composição deverá poder ser executada em qualquer plataforma de execução de composição.

Prioridade: Essencial.

[RF03] Independência de Plataforma de execução

Mesmo BPEL4WS sendo um padrão para a definição de Composição de serviços, na prática vemos que existem "dialetos" de BPEL onde cada *Engine* só consegue executar uma composição descrita naquele dialeto. Por conta disso, um requisito importante é a questão a independência da plataforma, que viria a ser uma extensão do requisito anterior.

Prioridade: Essencial.

[RF04] Suporte à especificação de Requisitos Não Funcionais

Requisitos não funcionais (também conhecido como Modelo de Qualidade) também devem ser abordados na solução proposta, tanto na parte de modelagem da composição (em tempo de projeto) quanto na parte Evolucional (tanto em tempo de projeto quanto em tempo de execução).

Prioridade: Essencial.

[RF05] Suporte à Semântica de Serviços

Em tempo de projeto, o projetista deve poder especificar semanticamente a funcionalidade da composição que ele está desenvolvendo.

Prioridade: Importante.

3.2.3 Requisitos Relacionados à Evolução Composicional

Os requisitos relacionados à Evolução Composicional nessa proposta estão descritos abaixo:

[RF06] Busca de Serviços

Deverá existir uma busca de serviços que será usada para encontrar serviços com determinadas características, dentre as quais: interface, descrição semântica e Propriedades não Funcionais (diferentemente de requisitos não funcionais, Propriedades não Funcionais não são apenas requisitos de projeto, mas sim uma descrição sobre as características do serviço).

Prioridade: Essencial.

[RF07] Validação usando Aspectos Semânticos

Verificações semânticas sempre devem ser usadas para garantir, na medida do possível, que as funcionalidades de uma composição não foram alteradas depois de uma evolução.

Prioridade: Essencial.

[RF08] Suporte a decisão sobre evolução de serviços

Evoluir uma composição pode ser necessário em várias situações, o sistema deve dar suporte para:

- Indisponibilidade: Quando um Serviço que compõe a Composição em questão ficar indisponível algum outro Serviço com características compatíveis deve ser usado para substituí-lo.
- Quebra de Requisito não Funcional: Quando um Serviço que compõe a Composição em questão começar a não obedecer aos Requisitos não Funcionais alocados para ele (ex.: preço, segurança etc.), outro Serviço com características compatíveis e que obedece aos requisitos exigidos deve ser usado para substituí-lo.
- Melhoria Disponível: Quando o sistema de busca identificar um Serviço que possui as mesmas características de algum outro que está sendo usando em alguma composição, mas com Requisitos não Funcionais superiores, o mesmo deve ser usado para substituir o Serviço menos adequado.

Prioridade: Essencial.

[RF09] Suporte a Modelos Evolucionais

Modelos evolucionais devem especificar aspectos como:

- De Tempo: Quais as janelas de tempo em que o sistema está apto a evoluir?
- Com relação aos Requisitos não Funcionais: Qual a prioridade dentre os Requisitos não Funcionais para classificação de um Serviço como melhor?

Prioridade: Essencial.

3.2.4 Requisitos não Funcionais

Os requisitos não-funcionais são detalhados a seguir.

[RNF01] Facilidade de uso

O sistema deve apresentar uma interface gráfica simples, com boa usabilidade e de fácil aprendizado.

Prioridade: importante

3.3 Arquitetura

Com o objetivo de atender todos os requisitos especificados na seção anterior foram definidas três entidades (ou aplicativos) no *Toolkit*: o *MoSC Modeling Tool*, a *MoSC Repository* e a *MoSC Auxiliary Engine*. Todas essas entidades possuem características e atribuições que serão detalhadas nas seções subseqüentes.

3.3.1 MoSC Repository

Esse aplicativo será responsável por armazenar e recuperar os Serviços que poderão ser usados para a criação de uma composição. O *MoSC repository* deverá dar suporte a busca de Serviços levando em consideração Requisitos não Funcionais e Semântica.

3.3.2 MoSC Modeling Tool

MoSC Modeling Tool é a ferramenta de modelagem da composição. E o principal resultado tangível do Trabalho de Graduação descrito nesse documento. A figura 3.1 mostra a arquitetura básica dessa entidade e sua relação com as outras entidades do *MoSC Toolkit*.

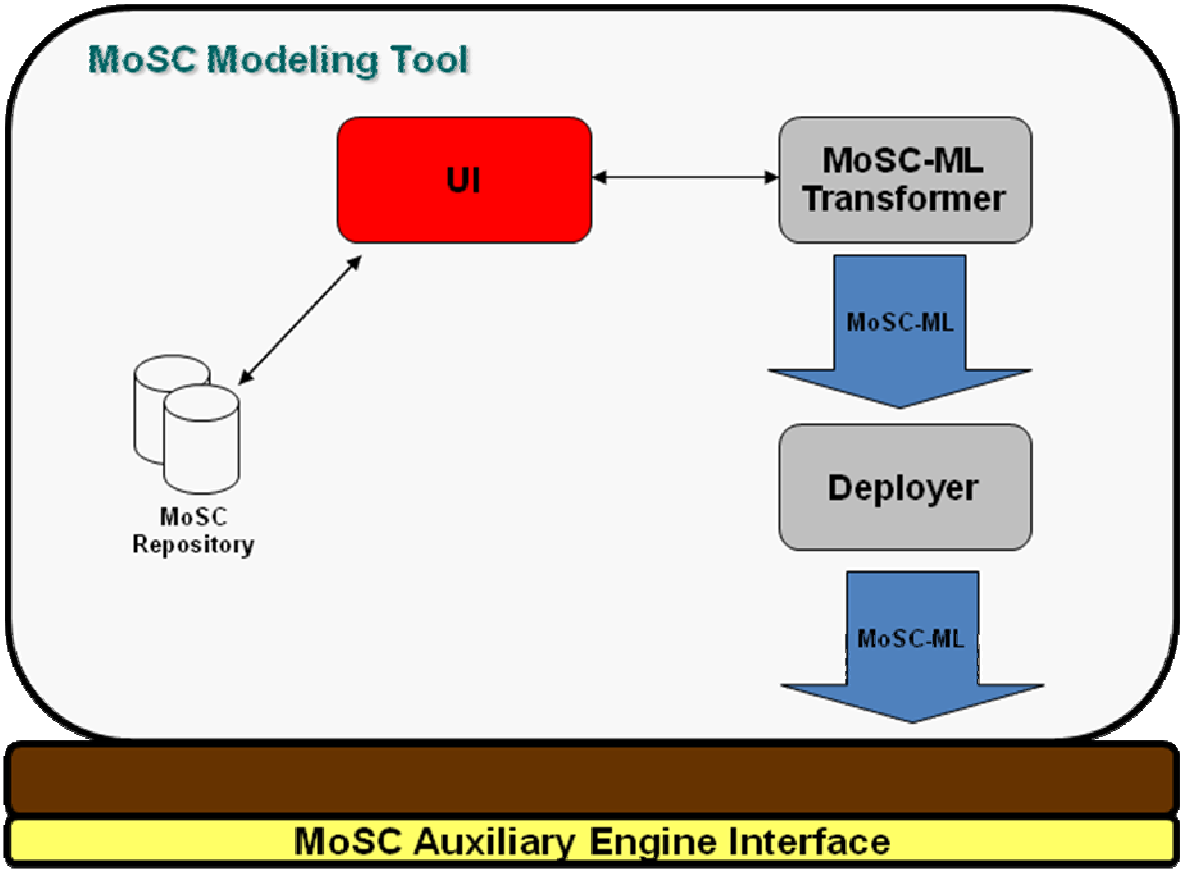

Figura 3.1 - Arquitetura do MoSC Modeling Tool

A arquitetura do *MoSC Modeling Tool* apresentada é formada, basicamente, por 4 entidades Básicas: UI (*User Interface*), MoSC-ML, *MoSC-ML Transformer* e *Deployer*.

- **MoSC-ML**

MoSC-ML é uma linguagem desenvolvida especialmente para descrever as composições feitas pela ferramenta e que é usada para o *deploy* da composição na *MoSC Auxiliary Engine*.

Essa linguagem interna é baseada em BPEL, essa decisão foi tomada pelo fato de BPEL ser uma linguagem muito popular para a composição de Serviços e fazer MoSC-ML parecer familiar para os desenvolvedores.

MoSC-ML foi feito para descrever todos os aspectos de uma modelagem de composição de serviço feito com o *MoSC Modeling Tool*, inclusive questões como Requisitos não Funcionais e descrição de modelos de evolução são abordados.

- **MoSC-ML Transformer**

O *MoSC-ML Transformer* é a entidade responsável por transformar a visualização gráfica criada pelo usuário na *MoSC Modeling Tool* em uma composição descrita em MoSC-ML.

- **Deployer**

O Deployer é responsável por gerenciar a comunicação entre o *MoSC Modeling Tool* e a *MoSC Auxiliary Engine*. Ele é responsável pelo implantação/"desimplantação"(*undeploy*) das Composições no *Auxiliary Engine* e também é responsável pode intermediar qualquer operação com a *Auxiliary Engine*, funcionando como um *proxy* dessa Entidade no *MoSC Modeling Tool*.

- **User Interface**

A *User Interface* (Interface Gráfica com Usuário) é o módulo de edição gráfica no *MoSC Modeling tool*. Esse módulo é responsável por fornecer ao usuário uma forma fácil de criar uma composição de serviços. Como especificado nos requisitos, a forma de composição é baseada na construção de um diagrama de fluxo, onde cada atividade do diagrama é na verdade uma invocação de um Serviço presente na biblioteca. Também foram definidos dois tipos básicos de estruturas de controle: *If/Else*, uma estrutura condicional; e *while*, uma estrutura de repetição.

Essas estruturas serão detalhadas na seção 3.5, a seção de implementação.

3.3.3 MoSC Auxiliary Engine

Com o intuito de prover independência de Plataforma de Execução (*Engine*) e realizar o gerenciamento da composição, inclusive nos aspectos evolucionais a *MoSC Auxiliary Engine* (ou simplesmente *Auxiliary Engine*) foi idealizada.

A arquitetura da *Auxiliary Engine* está descrita na figura 3.2 e será detalhada abaixo.

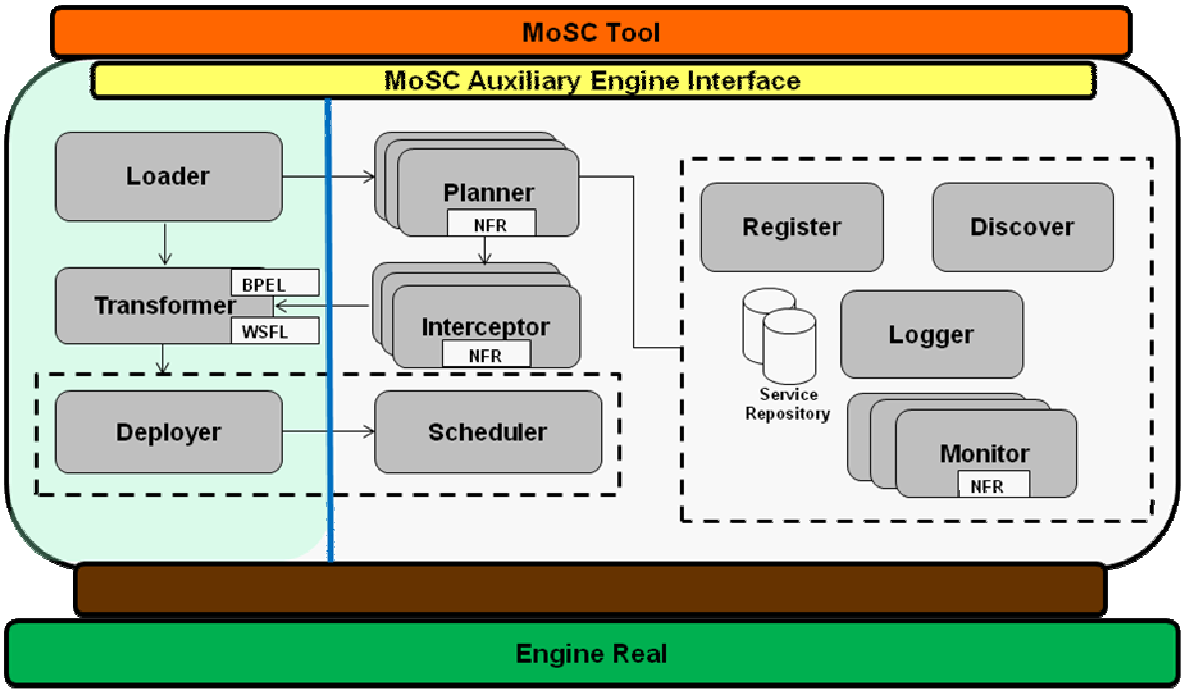

Figura 3.2 - Arquitetura do MoSC Auxiliary Engine

Como é possível observar na Arquitetura apresentada a *Auxiliary Engine* fornece uma abstração sobre qual *Engine* Real está sendo usada realmente e para isso alguns módulos, como: *Transformer, deployer e Scheduler*, devem possuir implementações distintas para cada *Engine* Real que está sendo usada.

A *Auxiliary Engine* pode ser dividida em quatro grupos de módulos, são eles:

- Grupo Básico

  O Grupo básico é formado pelos sub-módulos: *Loader, Transfomer e Deployer*. Esse grupo é responsável pelas atividades mais básicas da *Auxiliary Engine* como tratar as requisições feitas através da Interface da *Auxiliary Engine*, bem como ler o MoSC-ML de entrada (*Loader*); Transformar MoSC-ML para o dialeto da *Engine* Real (*Transformer*); realizar *deploy* e *undeploy* de uma composição na *Engine* Real considerando as características individuais de cada *engine* (*Deployer*).

- Grupo de suporte à evolução

Esse grupo é responsável por gerenciar os aspectos evolucionais da composição, respeitando os modelos evolucionais da composição (*Planner*) e os aspectos técnicos sobre quando uma composição pode efetivamente ser atualizada na *Auxiliary Engine* (*Scheduler*).

- Grupo de Monitoramento

  Esse grupo é responsável por auxiliar a o grupo de suporte a evolução com informações, como: obediência aos Requisitos não Funcionais através de trechos de código inserido dentro do código da composição para a criação de *Log* (*Interseptors*) que são monitorados em tempo real pelo Monitor que gera eventos para o *Planner* caso algum evento aconteça.

- Grupo de Busca

  Esse também é um módulo que auxilia o grupo de suporte à evolução pesquisando nos repositórios cadastrados por serviços que podem ser usados para a evolução de alguma composição em execução no momento (*Register* e *Discover*).

O foco desse Trabalho de Graduação é o modulo *MoSC Modeling Tool* e ele será detalhado nas duas próximas Seções.

3.4 Projeto

Como já foi mencionado, o escopo desse documento é a implementação do *MoSC Modeling Tool*, que é a ferramenta de modelagem de composição de serviços. Questões referentes às outras aplicações dos *MoSC Toolkit* serão abordadas apenas na seção de Trabalhos Futuros.

Nas próximas seções serão abordadas as decisões tomadas durante todo o projeto em diferentes aspectos, como: Editor Gráfico, Requisitos e Propriedades não funcionais e os aspectos dos Modelos Evolucionais.

3.4.1 Componentes Gráficos do MoSC Modeling Tool

Uma das partes principais do *MoSC Modeling Tool* é o seu editor gráfico. Afinal, é através do Editor que o usuário irá criar a composição de serviços. Tão grande quanto a sua importância é a quantidade de esforço

necessária para implementar uma interface gráfica com usuário. Para contornar, ou ao menos amenizar, essa questão foi usada a Plataforma Eclipse como base desenvolvimento, juntamente com alguns outros plug-ins para desenvolvimento de ambientes gráficos, como o GEF, *Graphical Editing Framework*.

- **Os Elementos Básicos**

Para a implementação do ambiente gráfico foi necessária a criação de modelos para todos os componentes do mesmo. A figura 3.3 mostra o diagrama de classes desses modelos.

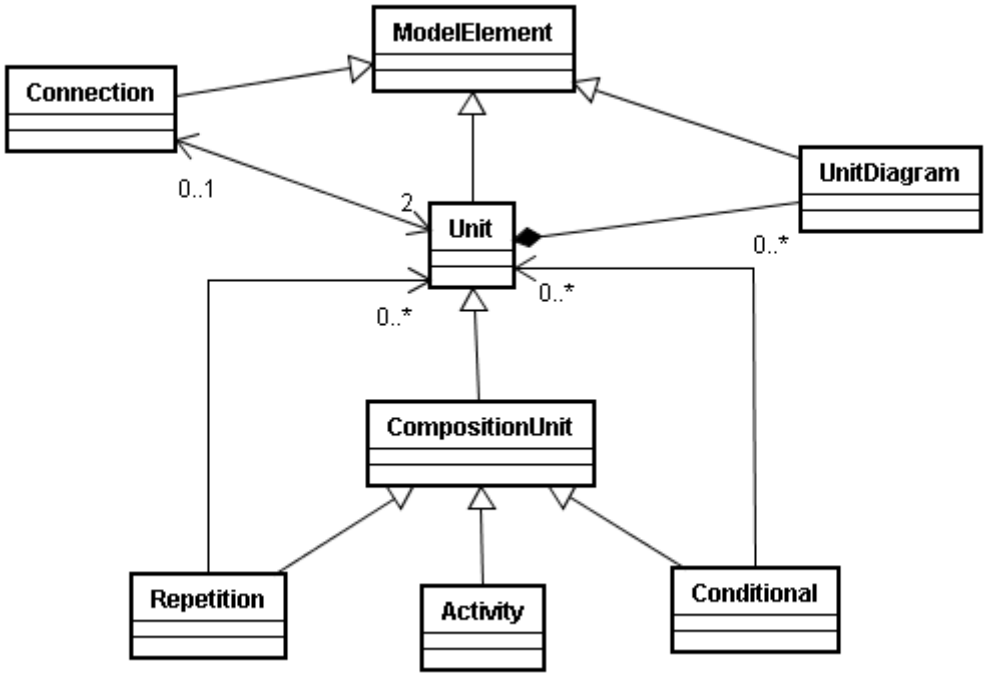

Figura 3.3 - Modelo de uma composição para o Ambiente Gráfico

Essa é a modelagem que é usada internamente pelo ambiente gráfico. Posteriormente, toda composição, que é uma instanciação desse modelo, é transformada em MoSC-ML que, por sua vez, pode ser transmitido para a *Auxiliary Engine* e finalmente executado. Serão detalhados agora os modelos mais importantes desse diagrama:

- **UnitDiagram**

  Hierarquicamente, esse é o principal modelo da composição no ponto de vista do ambiente gráfico. Todos os outros componentes que são instanciados na composição fazem parte desse modelo. Estão

associados a esse modelo todos os parâmetros globais da composição, como: a interface da composição e o Modelo Evolucional Global.

- **CompositionUnit**

  Essa classe representa a abstração de um componente semanticamente relevante para a composição. Por isso, ate o momento, as subclasses dele são: Activity, Conditional e Repetition.

- **Activity**

  Esse modelo representa uma Atividade no Diagrama da Composição. Como mencionado anteriormente, uma atividade é uma invocação de uma operação pertencente a um Serviço. Uma instância desse modelo possui as seguintes propriedades:

  - Nome: um aspecto muito importante com relação às atividades e que não é baseado de um diagrama de modelo é o nome, ou identificação, dessa instância de atividade. Essa identificação deve ser única para cada atividade da composição. Pois é através da identificação da atividade que aspectos individuais serão passados para a *Auxiliary Engine*.
  - Operação: Essa propriedade define qual operação de qual serviço se trata a invocação. Essa propriedade é definida pelo usuário com base no *MoSC Repository* que, como mostrado na seção de Arquitetura, é integrado com o *MoSC Modeling Tool*.
  - Requisitos não Funcionais: o usuário vai poder definir quais os Requisitos não Funcionais serão relevantes para aquela invocação de operação. Essas informações serão repassadas para a *Auxiliary Engine* e serão relevantes para a o monitoramento da composição e para a busca de serviços melhores para a composição.
  - Modelos Evolucionais Locais

- **Conditional**

  Esse modelo representa uma estrutura de controle condicional na composição. Através de uma condição o fluxo de execução da composição pode escolher uma direção diferente. Uma direção de a

condição for avaliada como verdadeira e outra direção se a condição for avaliada como falsa.

- **Repetition**

  Esse modelo representa uma estrutura de controle de repetição na composição. Através desse elemento um determinado trecho da composição pode ser executado de forma iterativa enquanto a condição que foi pré-estabelecida não seja mais avaliada pela *Engine* como verdadeira.

- **Connection**

  O fluxo de execução é definido através das conexões entre os componentes sempre através da noção de "requisitos de execução", ou seja, observando hierarquicamente, elementos que são "pais" de uma Atividade são também requisitos para que a sua execução possa se iniciada.

  As conexões também têm outro significado para uma composição. Elas representam transições entre os componentes da Composição. A principal operação feita durante uma transição no *MoSC Modeling Tool* é a atribuição de variáveis. Cada conexão pode definir uma série de operações de atribuição entre todas as variáveis da composição. Quando uma Atividade vai ser executada, todas as atribuições de todas as conexões que formam os pré-requisitos de execução da mesma são realizadas. Essa operação é comumente usada para inicializar as variáveis de entrada de uma invocação antes que ela seja feita.

3.4.2 Requisitos e Propriedades não Funcionais

Requisitos e Propriedades não Funcionais são propriedades que são definidos por um tipo de usuário cada. As Propriedades não Funcionais são definidas pelo projetista do serviço, a pessoa que sabe todas as características técnicas do serviço que está sendo disponibilizado; os Requisitos não Funcionais são definidos pelo o projetista da Composição que irá consumir um serviço e está atrelada a funcionalidade do serviço na composição e não a um serviço em particular.

As informações sobre Requisitos não Funcionais são especificadas pelo usuário durante a criação da composição e só fazem sentido quando está relacionado a Atividades. Estruturas condicionais, de repetição e conexões de fluxo não possuem Requisitos não Funcionais nessa versão do *MoSC Modeling Tool*.

Com relação a Propriedades não Funcionais, essas fazem sentido também a atividades de uma composição, mas de forma diferente. As Propriedades não Funcionais de uma Atividade estão relacionadas à operação que foi atribuída a ela pelo projetista da composição e essas informações vêm diretamente do repositório de serviços cabendo ao usuário apenas visualizar as informações e ao sistema de validar a relação dessas propriedades e dos requisitos que foram especificados pelo usuário.

Todavia, existe mais um uso para as Propriedades não Funcionais. Uma composição, sendo um Serviço, pode também ter especificadas as suas Propriedades não Funcionais pelo seu projetista que, posteriormente, pode ser usada pelo projetista de outra composição.

Nessa implementação do *MoSC Modeling Tool*, os Requisitos e Propriedades não Funcionais estão sendo tratador através de um par [chave, valor] (e.g. [desempenho, 300 ms], [segurança, nível 1]), apesar dessa ser uma modelagem simples, essa abordagem consegue ser bastante expressiva.

Requisitos e Propriedades não Funcionais são características muito semelhantes, então elas foram modeladas como uma Classe: NonFunctionalRequiremet. Assim, toda instância de Activity possui uma lista de Requisitos não Funcionais, assim como toda instância de UnitDiagram possui uma lista de Propriedades não Funcionais. Nos dois casos tem-se uma lista de NonFunctionalRequirement em cada instância.

Os Requisitos e Propriedades não funcionais foram modelados na ferramenta como está descrito pelo UML da figura 3.4. Todas as entidades envolvidas nessa modelagem serão descritas abaixo. Como Requisitos e Propriedades não Funcionais são modelados na mesma Entidade, nos referenciaremos daqui pra frente a eles como NFRP (*Non Functional Requirement or Property*)

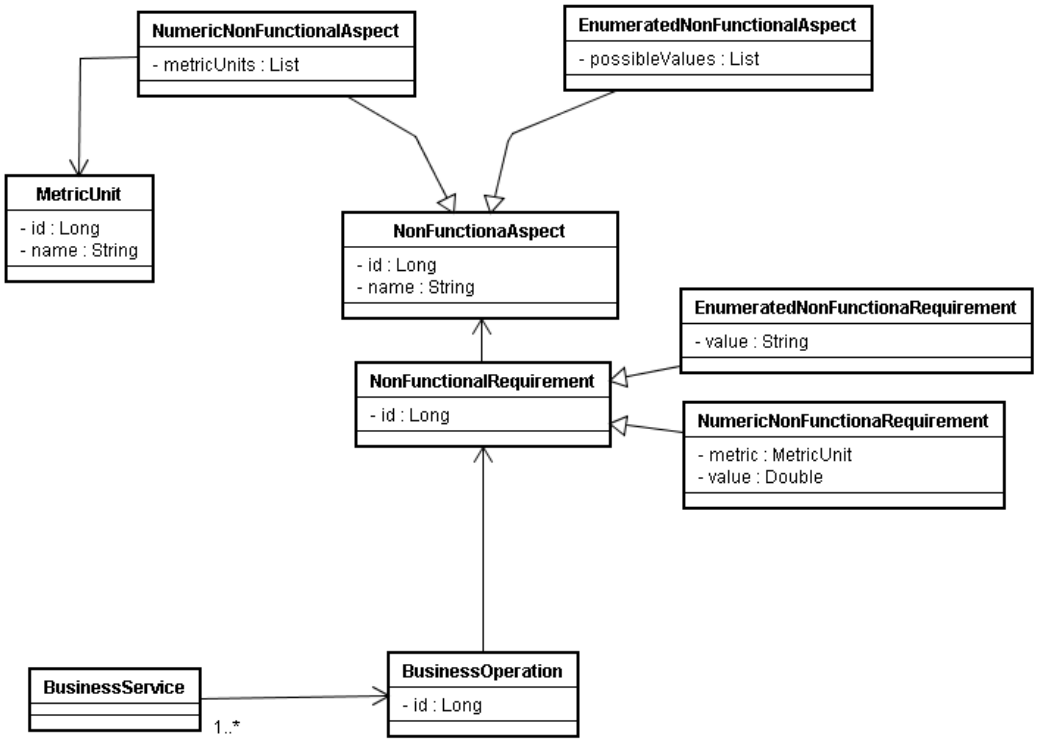

Figura 3.4 - Modelagem dos Requisitos não Funcionais

- **MetricUnit**

  NFRP, especialmente do tipo contínuos, possuem Unidades de Medidas que ajudam a especificar o formato da informação fornecida pelo usuário. Cada tipo de Propriedade ou requisito não funcional contínuo pode especificar quais unidades de medidas fazem sentido para ele.

- **NonFunctionalAspect**

  Um NonFunctionalAspect (ou NFA) foi uma entidade criada para representar um tipo especifico de Aspecto não funcional com seus valores e métricas possíveis.

- **NonFunctionalRequirement**

  Uma instância de NonFunctionalRequirement é uma instanciação de um Aspecto não Funcional. Digamos que se Desempenho é um Aspecto não Funcional, quando instanciado em forma de NonFunctionalRequirement ele ganha um valor e uma unidade (e.g. valor: 100 e unidade: ns).

- **NumericNonFunctionalAspect**

Essa entidade é uma subclasse de NonFunctionalAspect que representa NFRP numéricos e contínuos, como: desempenho, custo etc.. Esse tipo de aspecto contém quais as unidades de medida que são relevantes para esse aspecto, por exemplo, para desempenho pode-se usar unidades de tempo como segundo, milissegundo; e para custo pode-se usar moeda, como dólar, Euro ou Real.

- **EnumeratedNonFunctionalAspect**

  Essa entidade é uma subclasse de NonFunctionalAspect que representa NFRP discretos e pré-definidos, como níveis de segurança ou algum status booleano, como existência de comunicação criptografada. O que é definido no aspecto além do nome do mesmo é apenas quais são os valores possíveis que o usuário desse Aspecto vai poder escolher como valor do NRFP associado.

- **NumericNonFunctionalRequirement**

  Essa entidade se trata de uma instanciação de um NFRP com base em uma instância de um NumericNonFunctionalAspect. Essa entidade possui as definições necessárias sobre uma NFRP contínua, como o seu valor numérico e a unidade de medida associada aquele valor.

- **EnumeratedNonFunctionalRequirement**

  Assim como NumericNonFunctionalRequirement está para NumericNonFunctionalAspect, EnumeratedNonFunctionalRequirement está para EnumeratedNonFunctionalAspect. Essa entidade possui a informação sobre qual dos possíveis valores que o NFRP poderia ter (informação contida em seu aspecto associado) foi escolhido pelo projetista da composição ou do serviço.

3.4.3 Variáveis

Uma parte fundamental para a construção de composições ou até mesmo qualquer aplicação num paradigma imperativo é a gerência de variáveis. E principalmente quando grande parte do código é gerada automaticamente. Nessa seção será apresentada a abordagem para o tratamento de variáveis no *MoSC Modeling Tool*.

A principal fonte com informações sobre tipos de variáveis se encontra no WSDL dos serviços que formam a composição. Esses tipos estão presentes nas mensagens que são trocadas entre as partes para a invocação de uma operação desse Serviço. Através de um documento WSDL pode-se criar uma grande variedade de tipos de variáveis, como: tipos simples vindos de algum esquema importado pelo WSDL, tipos complexos formados por tipos simples, hierarquias de tipos complexos (tipos complexos formado por outros tipos complexos), além de elementos de XML também podem ser considerados tipos.

Mas uma das partes mais delicadas quando se trata com criação de novos tipos em linguagens baseadas em XML é questão envolvendo *namespaces*. Através dos *namespaces* é que se podem achar as verdadeiras definições de um tipo de variável.

Essa implementação do *MoSC Modeling Tool* dá suporte apenas a tipos simples, básicos de XML ou não. Por causa dessa decisão, faz-se necessário realizar um tratamento especifico para todos os tipos complexos que possam vir a serem usados pelos serviços que fazem parte da composição que está sendo criada. Para isso, foi modelado o problema do mapeamento das variáveis de maneira hierárquica também, como pode ser visto na figura 3.5.

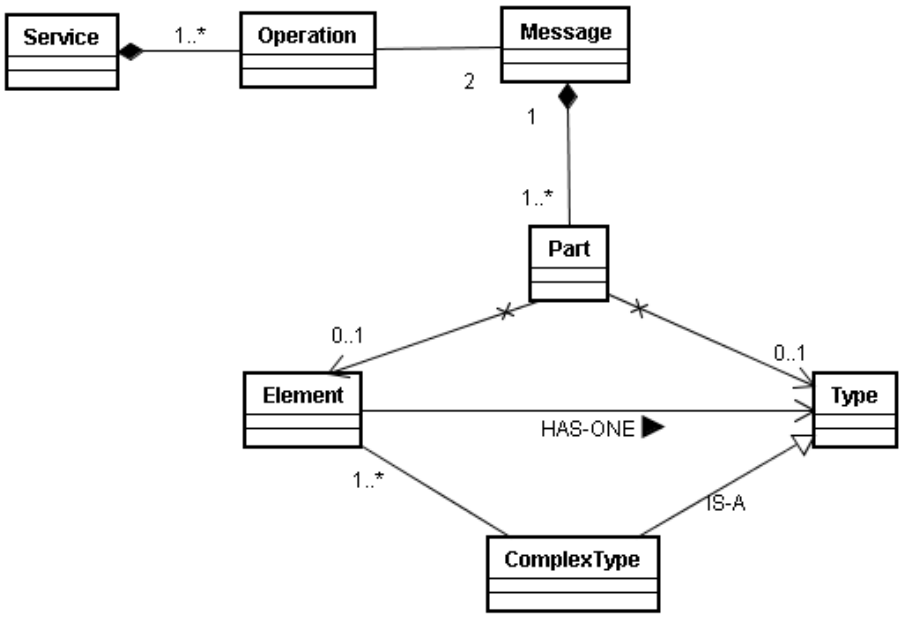

Figura 3.5 - Modelagem das Variáveis

Na figura 3.5 pode-se perceber que a origem as definições das variáveis é a interface das operações dos serviços. Todas as entidades descritas abaixo são inspiradas diretamente do padrão WSDL que é onde todas as informações sobre os tipos e sobre as interfaces estão especificadas.

- **Message**

  Uma *message* (mensagem) é a estrutura de alto nível que representa um conjunto de informações trocadas pelos serviços. É dentro de uma mensagem que são especificados os dados que são trocados em cada mensagem.

- **Part**

  Essa entidade representa cada parte da mensagem individualmente.

- **Type, ComplexType e Element**

  Essas entidades representam efetivamente o formato dos dados que são trocados dentro das mensagens. Essas estruturas podem se estruturar hierarquicamente e essa estrutura é importante para mapeamento dos mesmos durante as operações de atribuição.

3.4.4 MoSC-ML Transformer

Durante a construção de uma composição usando o *MoSC Modeling Tool* código MoSC-ML é gerado automaticamente através do modulo *Transformer*.

Durante a criação do MoSC-ML, o *Transformer* tenta gerar um código mais funcionalmente semelhante com o que está expresso no editor gráfico. Nessa primeira versão dessa ferramenta o código que esse módulo gera apesar de funcionalmente equivalente com a composição descrita no editor gráfico, esse código se demonstra ter um desempenho menor, pois trata os requisitos de uma determinada atividade de forma local e não global. Essa deficiência na geração do código impede que blocos de atividades que poderiam executar totalmente em paralelo tenham trechos executados seqüencialmente. Mas as relações de dependência nunca são ignoradas ou quebradas, fazendo com que a funcionalidade da composição seja mantida.

3.5 Implementação

Nessa seção, serão detalhados os aspectos operacionais da construção e funcionamento dessa ferramenta.

Como mencionado na seção de 3.4, por causa da complexidade de fazer uma ferramenta com um ambiente gráfico para modelagem foi decidido pelo uso da Plataforma Eclipse e essa decisão implica em algumas outras decisões de projeto, como:

- ✓Linguagem de Programação: Para construir uma aplicação com base no Eclipse é necessário usar a linguagem de programação Java. Essa escolha não teve nenhum custo adicional, pois Java é uma linguagem de fácil entendimento, poderosa e propicia uma fácil integração entre componentes, além ser Orientada a Objeto (OO) propicia também um grande reuso de código.
- ✓Versão do Eclipse: Para essa versão da ferramenta de modelagem está sendo usada a versa 3.4.1 do eclipse, chamado Eclipse Ganymede.
- ✓Plug-ins Dependentes: Para a criação de um Ambiente para modelagem gráfica faz-se necessário de mais alguns plug-ins do eclipse, o principal deles é o GEF (*Graphical Editing Framework*) que é responsável pelo editor gráfico em si. A versão do GEF que está sendo utilizada é a 3.4.1;

Nas próximas subseções, será apresentada a ferramenta e suas funcionalidades. Detalhes sobre a implementação de cada funcionalidade serão explicados à medida que elas forem sendo apresentadas.

3.5.1 MoSC Modeling Tool

Nessa seção será apresentada a versão final do *MoSC Modeling Tool* que foi desenvolvida para esse Trabalho de Graduação. A Figura 3.6 mostra uma visão geral da ferramenta e seus principais componentes, e esses, por sua vez, serão detalhados a seguir.

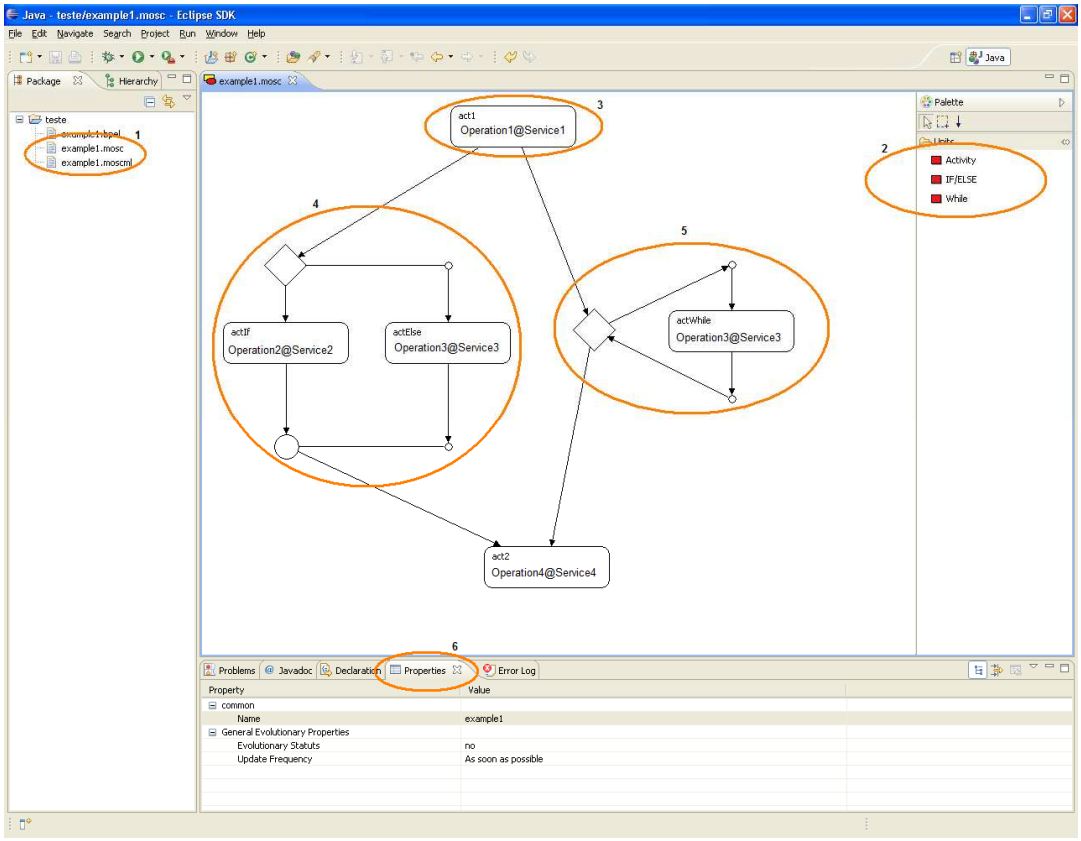

Figura 3.6 - MoSC Modeling Tool

Na figura 3.6, temos as principais partes do Ambiente Gráfico, são elas:

1. Arquivos da Composição: Para cada composição criada são criados dois arquivos, um arquivo com extensão *.mosc e outro com extensão *.moscml que representam a composição para o ambiente gráfico e a composição descrita em MoSC-ML, respectivamente;
2. Paleta de Componentes: Nessa paleta de componentes temos os elementos básicos do *MoSC Modeling Tool* que foram especificados em seções anteriores: o Activity, o Connection, o Repetition e o Conditional;
3. Activity: Um principal componente do Ambiente gráfico. Como mencionado anteriormente, esse é o componente que representa uma invocação de uma operação de outro serviço. Vale ressaltar duas características desse componente: a primeira é o nome, que deve ser único dentro de toda a composição, pois é a partir desse nome que são gerados pelo

Transformer os nomes das variáveis referentes a cada invocação de operação e que aparece na parte superior da figura referente à Activity. A segunda característica é proveniente da Operação que o usuário atribui a atividade, tanto o nome da operação quanto o nome do serviço aparecem na parte inferior da atividade.

4. Conditional: Essa é a representação gráfica da estrutura condicional. As atividades que estiverem na mesma linha de execução principal (na imagem, onde a ActIf se encontra) será executado caso a condição (uma propriedade da Estrutura Condicional) seja avaliada como Verdadeira no momento da execução dessa Condição, caso a condição seja avaliada como Falsa a linha de execução secundária (na imagem, onde o ActElse se encontra) será executada. Nessa versão do *MoSC Modeling Tool* já é permitido encadeamentos de estruturas condicionais e de repetição.
5. Repetition: Essa é a representação gráfica da estrutura de repetição. As atividades que estiverem dispostas na linha de execução dessa estrutura (na imagem, onde o ActWhile se encontra) serão repetidas até que a condição (uma propriedade da estrutura de repetição) seja avaliada como Falsa.
6. Property View: Essa área da janela do Eclipse é destinada a exibição e alteração de Propriedades existentes em elementos gráficos. Todas as propriedades das estruturas que foram mencionadas até o momento (e.g. Nomes das atividades, Condição das estruturas Condicionais e de Repetição) podem ser alteradas por essa *View*.

Criando uma interface para a composição

A primeira etapa na criação de toda a composição deve ser a definição da *interface* da mesma. Nessa definição é que o usuário vai poder especificar quais são os parâmetros e o retorno desse serviço. As figuras 3.7 e 3.8 detalham como essa operação é feita graficamente.

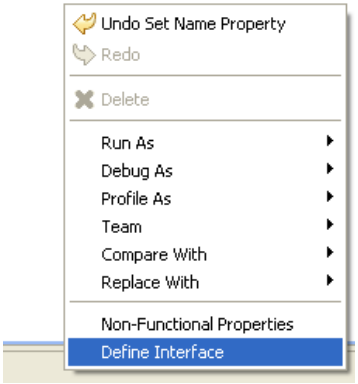

Figura 3.7 - Define Interface

O menu de contexto mostrado na figura 3.7 é exibido no editor gráfica quando se clica com o botão direito do mouse sobre uma parte do ambiente gráfico onde não se tem nenhuma outro componente selecionado. Essa operação, assim, aciona as funcionalidades referentes à composição de modo global.

Logo após essa operação, irá aparecer uma janela como a mostrada na figura 3.8.

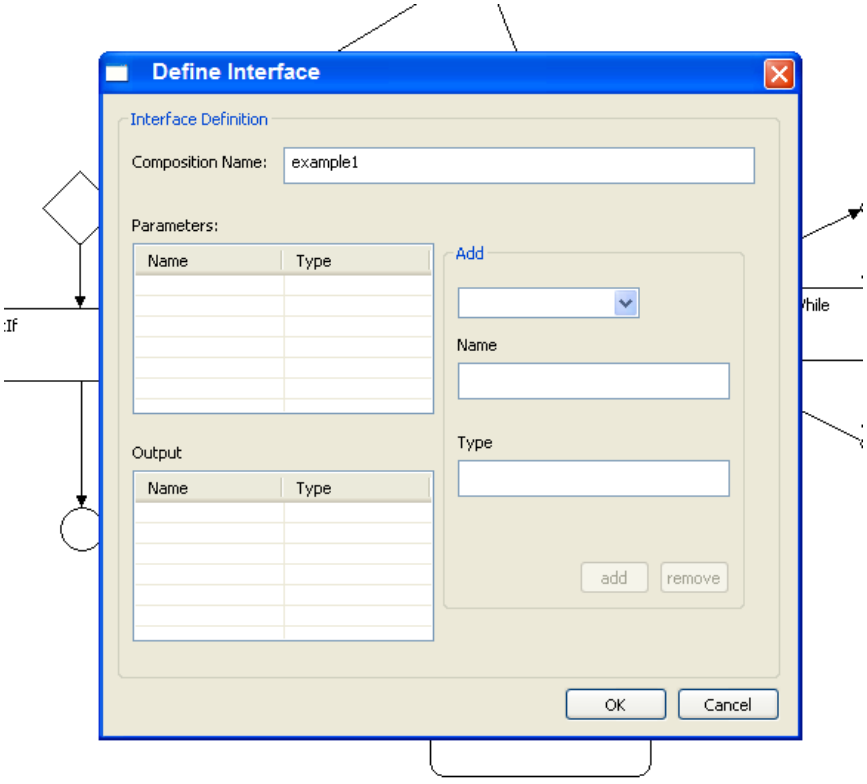

Figura 3.8 - Janela "Define Interface"

Nessa janela, o usuário poderá adicionar ou remover parâmetros ou elementos na estrutura de retorno.

Associando uma Operação a uma Activity

Associar uma operação a uma Activity é um passo fundamental para a execução de uma composição. Nessa versão do *MoSC Modeling Tool* todas as Activity's tem que estar associadas a alguma operação para que o *Transformer* possa gerar o MoSC-ML corretamente. As Figuras 3.9 e 3.10 mostram como associar uma operação a um Activity.

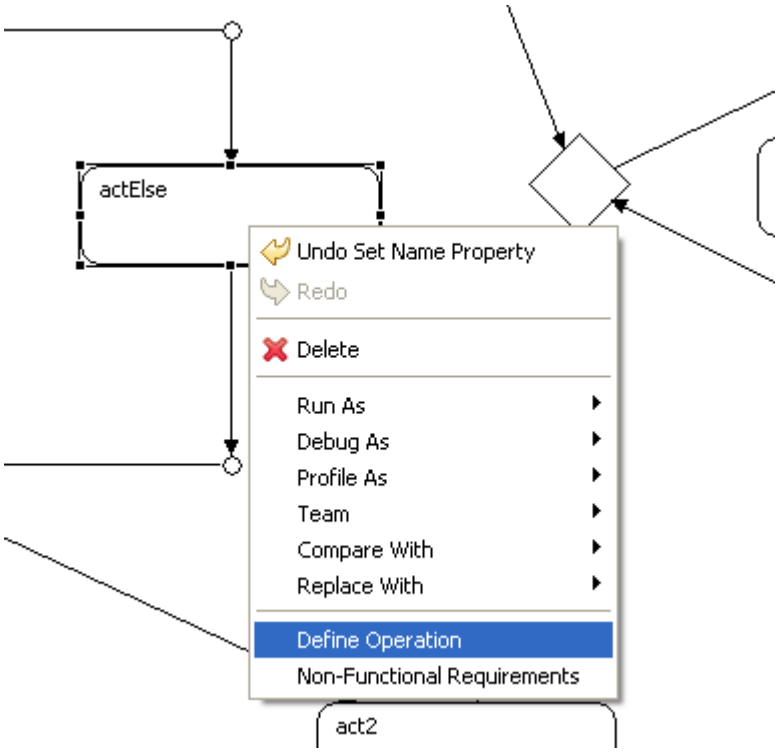

Figura 3.9 - Escolhendo a opção "Define Operation"

Na figura 3.9, o usuário clicou com o botão direito do mouse em uma atividade e em seguida escolheu a opção "*Define Operation*". Logo após isso, aparece uma Janela como podemos ver na figura 3.10.

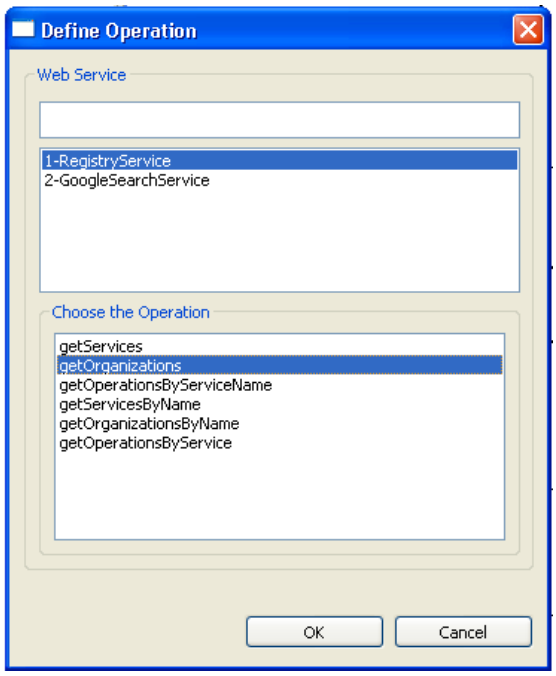

Figura 3.10 – Define Operation

Após aparecer a janela mostrada na figura 3.10 o usuário digitou o nome do serviço e depois o nome da operação que ele gostaria de associar aquela Activity. Depois que ele fez a escolha, ele clica no botão "*Ok*" e a operação e associada aquela Activity.

Associação entre as variáveis

Boa parte das operações sobre as variáveis são abstraídas do usuário, por exemplo, todas as variáveis automaticamente em duas situações: quando o usuário define a operação de uma atividade e quando o usuário define a interface da composição; para essas duas operações as variáveis de Entrada e Saída de cada operação são criadas e ficam disponíveis para uso do usuário.

Mas a operação de associação de variáveis é tão importante e está tão ligada a funcionalidade da composição que não é possível que ela seja feita automaticamente com a estrutura dessa primeira versão MoSC *Modeling Tool*. Com isso, o usuário deve especificar manualmente as associações entre as variáveis em cada transição da composição (nas conexões entre as atividades).

Nas figuras 3.11 e 3.12 detalham graficamente como é feita essa operação.

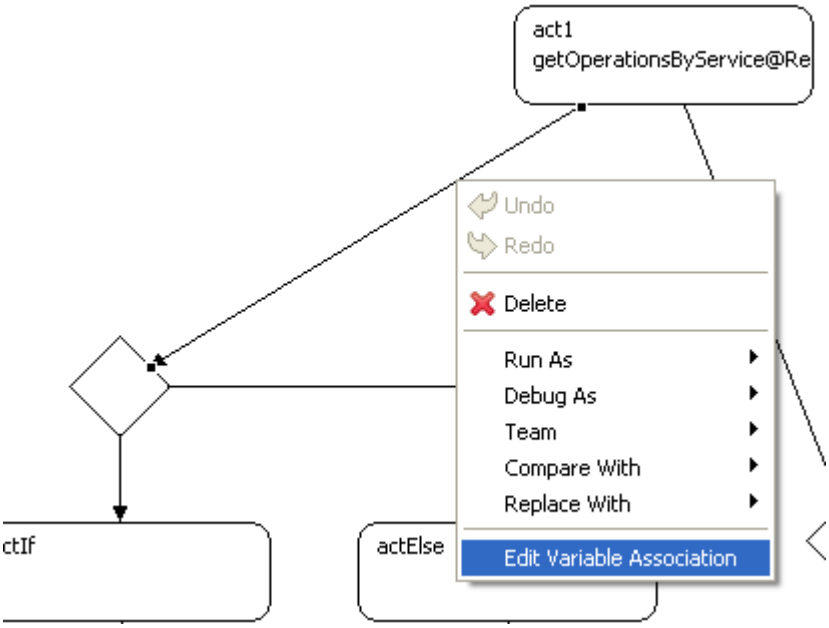

Figura 3.11 – Variable Association

Na figura 3.11, o usuário clica com o botão direito sobre uma conexão entre duas atividades, e seleciona o botão "*Edit Variable Association*". Essa operação irá definir quais associações iram acontecer quando aquela transição for executada.

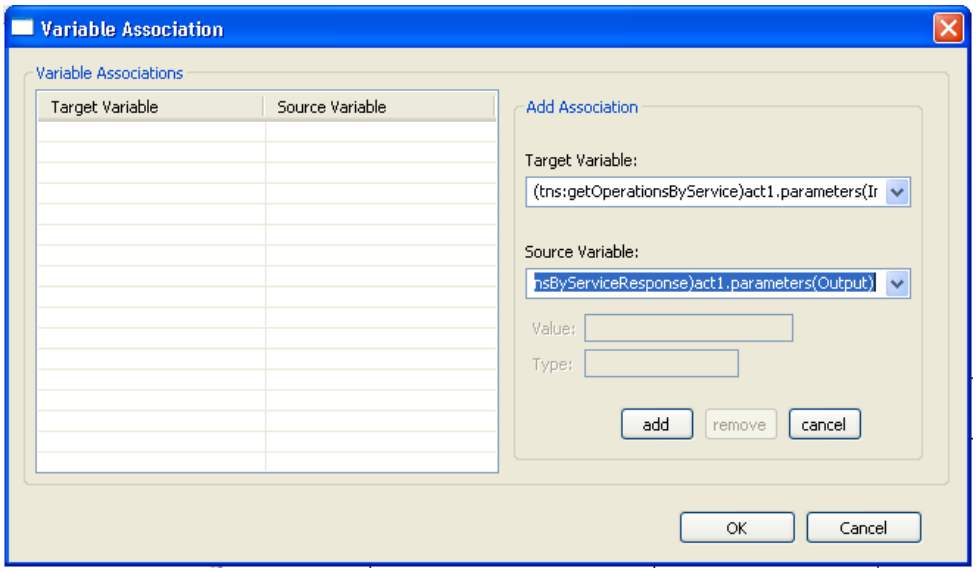

Figure 3.12 – Variable Association

Na figura 3.12, o usuário vai poder associar as variáveis a sua escolha. Nesse ponto, o usuário tem duas opções a fazer, ou ele atribui uma variável a outra, ou ele atribui uma constante (valor literal) a uma variável. Como falado anteriormente, essa versão do *MoSC Modeling Tool* trata apenas variáveis de tipos simples.

Definindo Requisitos e Propriedades não Funcionais

Os conceitos de Requisito e Propriedade não funcional é tão semelhantes que para usuário, levando em consideração apenas a *Interface Gráfica*, trabalhar com uma das duas coisas é uma questão apenas de localização do menu. A entidade Activity tem um botão para editar Requisitos não Funcionais (Figura 3.13) enquanto que no menu global tem a o botão para editar as Propriedades não Funcionais (Figura 3.14).

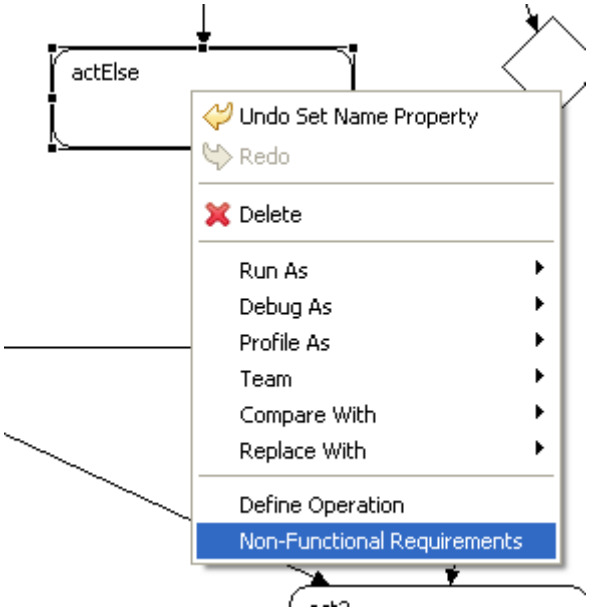

Figura 3.13 - Botão para editar os Requisitos não funcionais da Atividade

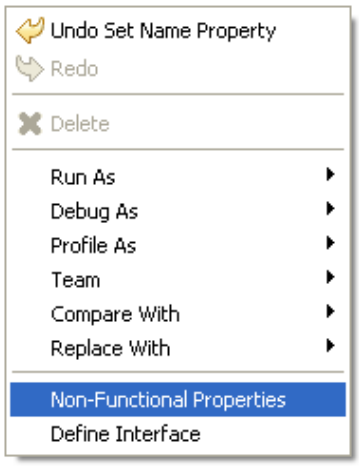

Figura 3.14 - Editar as Propriedades não funcionais da composição

A figura 3.15 mostra a janela de edição de Requisitos não Funcionais (que é semelhante à janela de edição de propriedades não funcionais). Nessa janela o usuário deve primeiramente escolher qual aspecto não funcional ele quer adicionar e depois ele deve inserir os valores para esse requisito ou propriedade.

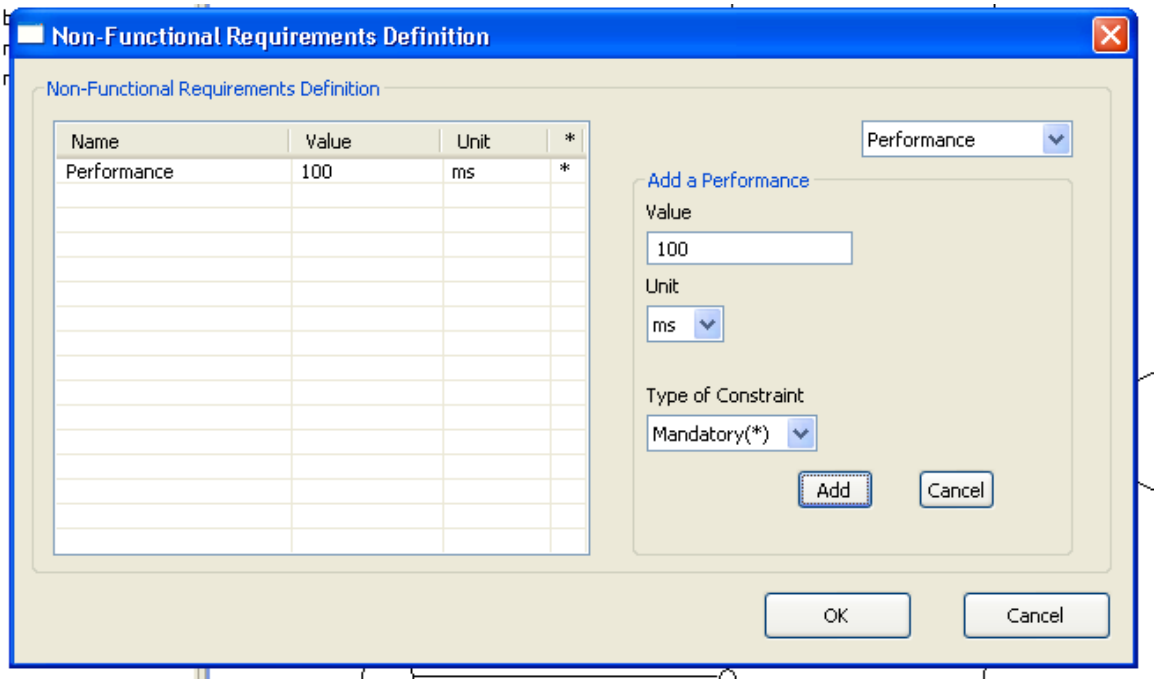

Figura 3.15 - Adicionando Requisitos não Funcionais a uma Activity

Geração de MoSC-ML

A geração do código MoSC-ML referente a composição é feita automaticamente todas as vezes que o arquivo da composição é salvo. Assim é gerado um arquivo com o mesmo nome do arquivo *.mosc, mas o arquivo da composição em MoSC-ML tem a extenção *.moscml.

A figura 3.16, mostra um exemplo de um arquivo MoSC-ML gerado.

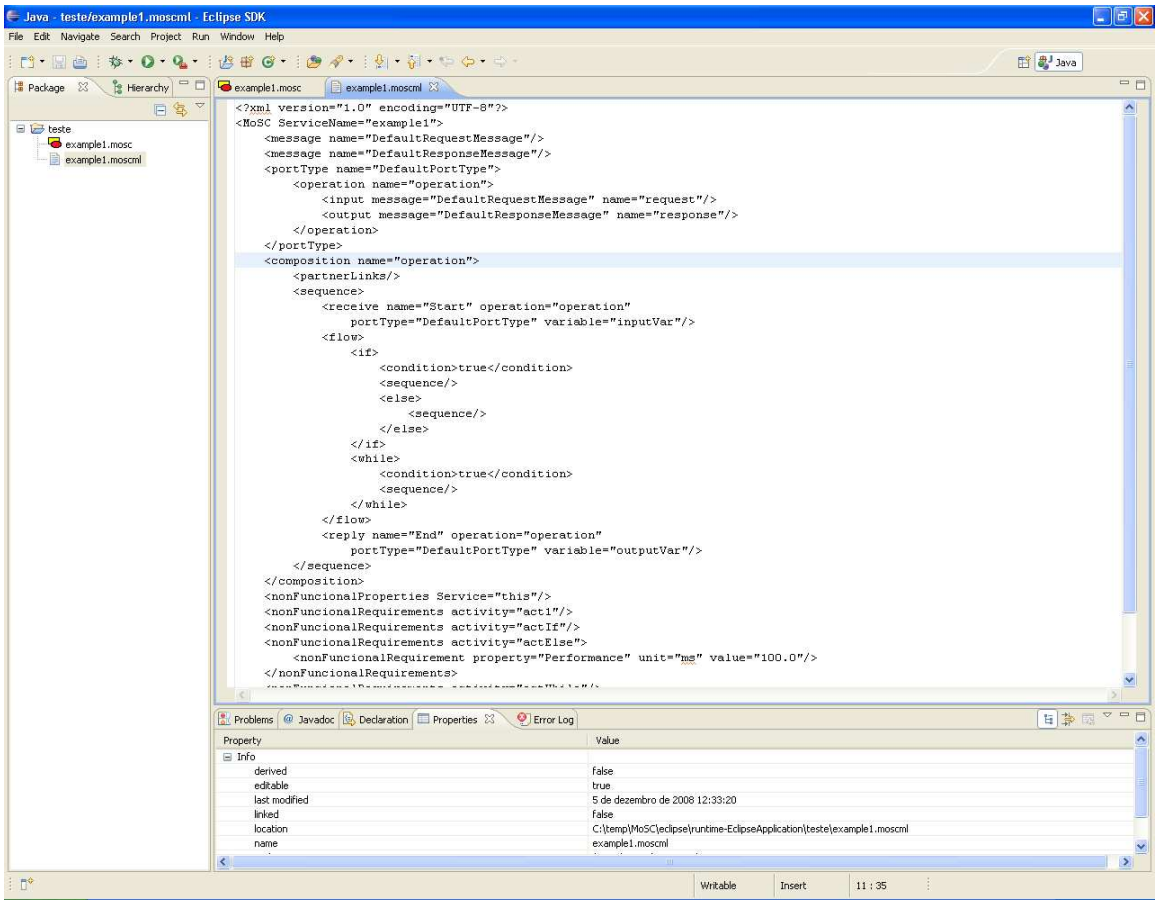

Figura 3.16 - Arquivo MoSC-ML gerado

Aspectos Evolucionais

Nessa seção iremos tratar das características da ferramenta referentes aos aspectos evolucionais discutidos durante a fase de requisitos e projeto.

- ✓**Políticas Evolucionais globais**: Dentre as propriedades da composição, temos as políticas evolucionais globais onde é possível definir se a composição vai poder evoluir e a freqüência em que essa composição vai poder evoluir (*as soon as possible, dayly, weekly, monthly e defined by user*);
- ✓**Polílicas Evolucionais Locais**: Políticas locais definem políticas individuais para cada atividade. Existem dois tipos de políticas individuais que podem ser definidas:
  - o *Status de Evolvability*: Define se essa atividade vai poder evoluir ao longo do tempo ou não.
  - o Política com relação aos Requisitos não Funcionais: Para cada requisito não funcional adicionado o usuário deve

definir se ele é obrigatório ou se ele é desejável enquanto que haja uma melhoria em algum outro requisito durante uma evolução ou não. Essa escolha é feita durante a própria ação de edição de Requisitos não Funcionais.

3.6 Considerações Finais

Ao final desse desenvolvimento percebe-se que não se construiu uma ferramenta de composição de serviços completa, mas o que foi feito foi parte de uma ferramenta que servirá de protótipo para vários conceitos que agora estão integrados. Muitas das coisas que foram implementadas no *MoSC Modeling Tool* ainda não estão em nenhuma outra ferramenta existente no mercado e isso prova a relevância desse trabalho.

4 Estado da Arte

Nessa seção serão discutidas as principais propostas para abordar as questões de composição de serviços baseado em modelos.

Picasso [20] é uma arquitetura onde os serviços são fracamente acoplados e orquestrados usando SOA (*Service Oriented Architecture*) [12]. Ao mesmo tempo, na Picasso, as descrições dos serviços e as interações entre eles são modeladas usando o processo de modelagem descrito para o MDA (*Model-Driven Architecture*) [15]. Na prática, Picasso é uma instância concreta de SOA e MDA para automação. Além disto, Picasso tem como hipótese que "o problema da interoperabilidade entre serviços passa a ser de consistência de modelos, mas do que consistência de *interfaces*". Neste caso, o comportamento do serviço passa a ser expresso não pela sua interface, mas pelo seu modelo.

De acordo com [20], os benefícios de Picasso incluem: serviços *plug-and-play*, evolução de serviços, modelos de serviços descentralizados, independência de protocolo e evolução, customização.

No Picasso, todos os serviços interagem através de modelos de serviço. Um modelo de serviço é a representação de um serviço dentro do SOA e define (1) a descrição externamente visível do serviço, (2) o comportamento,

(3) o estado e (4) as operações providas pelo serviço a outros serviços. O modelo de serviço é utilizado para introspecção, composição de serviços, interoperabilidade, descoberta e abstração.

A Figura 4.1 ilustra uma visão geral da arquitetura Picasso.

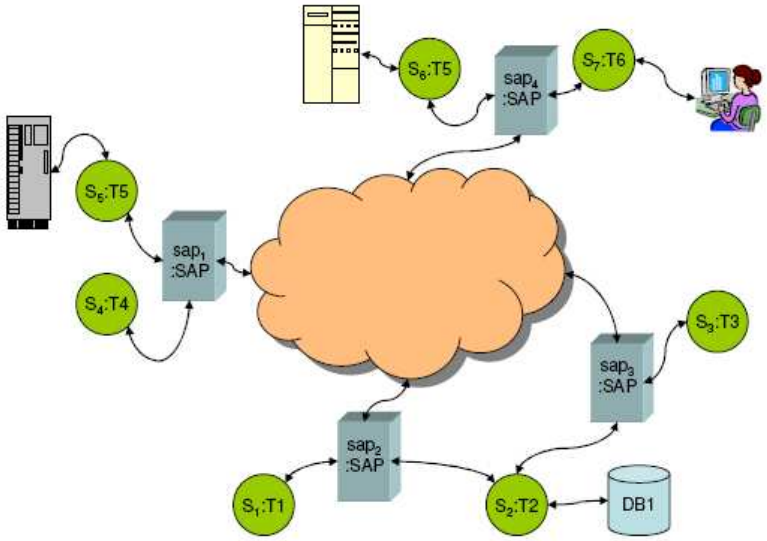

Figura 4.1 - Visão geral da arquitetura Picasso

Serviços se comunicam trocando modelos, onde estes modelos contem operações e o estado do serviço.

Schattkowsky [19] propõe um desenvolvimento baseado em modelo para sistemas embarcados em tempo real. A proposta consiste de uma máquina virtual UML (UVM) que funciona como ambiente de execução para (um subconjunto de) especificações UML. As especificações UML executáveis consistem de diagramas de classes, de transições de estados (STDs) e diagramas de seqüência (SD) que cobrem interrupções, exceções e *timeouts*. As especificações são compiladas em programas binários consistindo de estrutura de dados (STDs) e *bytecodes* (SD) que executam diretamente na UVM.

Pfadenhauer [18] define três elementos principais que devem ser observados nas propostas sobre composição de serviço guiada por modelo: representação/especificação da composição, tarefas e ambientes. A representação e especificação incluem o modelo de composição, a linguagem para expressar a composição e a linguagem de composição executável. Quanto às tarefas, as seguintes tarefas devem ser observadas: teste, monitoramento, verificação semântica e sintática, descoberta/*bind*, implantação e

mapeamento. Por fim, os ambientes incluem os ambientes de modelagem, composição e execução.

Skogan [21] propõe (1) o uso do Diagrama de Atividade UML para composição de *Web Services* e (2) MDA para geração de especificações executáveis em diferentes linguagens de composição, e.g., BPEL4WS. A proposta consiste de um *guideline* de como usar UML para especificar a composição, um *UML Profile* e regras de transformação que produzem modelos UML da composição de serviços que são executáveis. Básico nesta abordagem é a utilização de UML como uma "plataforma de integração" para modelagem de composição de *Web Services* executáveis. Para isto considera-se que UML é capaz de expressar composição de WS, as composições descritas em UML podem ser mapeadas composições executáveis[1] (e.g., BPEL4WS), e as especificações descritas em UML são independentes da linguagem de composição executável.

Orriens [16] também propõe o uso de UML (Diagrama de Classes) para expressar a composição de serviços, mas usa ainda OCL para expressar as regras de composição. Na proposta apresentada, o processo de desenvolvimento baseado na composição de serviço inicia com uma composição abstrata, seguida por uma definição de quando e como os serviços devem executar. Nesta fase, são definidas possíveis composições de serviços concretos que podem realizar a composição abstrata definida na fase anterior. A seguir, o usuário deve definir entre as possíveis composições geradas na fase anterior, qual a composição desejada. Por fim, na última fase, a composição resultante é mapeada em uma composição executável (e.g., BPEL4WS).

Belhajjame [4] propõe uma abordagem baseada em modelo para mecanizar a composição de ser serviço no contexto de empresas virtuais. Nesta proposta, a composição de serviços é gerada automaticamente a partir de uma descrição feita em uma linguagem declarativa (e.g., SQL) formalmente definida. Nesta definição, o usuário define os serviços que deverão executar (serviços participantes) e uma composição é gerada automaticamente aplicando-se as regras de negócio da empresa virtual.

[1] Uma especificação é executável se ela pode ser executada por uma "engine" como ActiveBPEL.

Oberleitner [14] apresenta uma abordagem para geração de composições de serviços baseada em modelos. As composições são geradas a partir de modelos UML (diagramas de estado e de atividade) e mapeadas em plataformas *Web Services*, CORBA, .Net e COM. Os diagramas UML são transformados em especificações de composição executáveis em uma *engine*.

Zeng [23] apresenta uma proposta para composição de serviços com três características básicas. Primeiro, o foco da composição é em tempo de execução e não em tempo de projeto como as propostas definidas em [19][21][16][4]. Segundo, é definido um modelo de qualidade para *Web Services* capaz de expressar requisitos não-funcionais dos serviços (e.g., preço de execução, duração da execução, reputação, confiabilidade e disponibilidade, juntamente com restrições globais e preferências do usuário (e.g., restrições orçamentárias). Por fim, é definido um processo de seleção dos serviços em tempo de execução que procura otimizar os serviços selecionados considerando os critérios de qualidade definidos no modelo de qualidade dos serviços. Como notação para definir a composição de serviços, a escolha foi o uso de *statechart*, baseado em dois argumentos: a base formal de *statecharts* e a existência de operadores similares aos operadores encontrados em linguagens de composição executáveis como BPEL. Este segundo fato, permite que modelos em *statechart* sejam mais facilmente mapeados nestas linguagens.

Zeng [24] apresenta um *middleware* (*AgFlow*) capaz de selecionar *Web Services* para a composição maximizando a satisfação do usuário e satisfazendo restrições impostas pelos usuários e pela estrutura da composição. Os pontos principais desta proposta referem-se à definição de uma estratégia para modelagem de requisitos de QoS (neste contexto, requisitos não funcionais), uma composição de serviços "*QoS-aware*" onde requisitos não funcionais são observados e maximizados no processo de composição e a composição de serviços é considerada em tempo de execução (dinâmica), i.e., mudanças dos requisitos não-funcionais dos serviços em tempo de execução, provocadas por mudanças internas ou do ambiente, são levadas em consideração.

Issa [10] concentra-se em requisito de QoS da composição de serviços, da mesma forma que [24], mas os requisitos considerados privilegiam a

precisão e a "*freshness*" dos dados. Para isto, é proposto um *middleware publish/subscribe* baseado no padrão *WS-Notification* que é capaz de sentir e rotear qualquer mudança de informação no nível de *Web Services*.

Grønmo [9] apresenta uma estratégia baseada em MDA para composição de *Web Services* onde semântica web (ontologias) e descrição de propriedades não-funcionais são utilizadas para escolha de *Web Services* que atendam a uma determinada composição. Inicialmente é definida uma composição abstrata (diagrama de atividade) que inclui propriedades não-funcionais (diagrama de classes). Esta composição inicial é transformada em uma composição concreta em BPEL. Instâncias dos *Web Services* presentes na descrição BPEL são procuradas em repositórios considerando requisitos não funcionais definidos na composição. Os *Web Services* são então ranqueados e selecionados para a execução da composição.

Brambilla [6] propõe uma abordagem visual para a modelagem de aplicações web através do uso de uma ferramenta *case* Webratio (www.webratio.com, [25]) que suporta vários padrões (Java/JSP, RDBMS, *Web Services*, HTML & XHTML Web interfaces, Mobile Wap, AJAX). A modelagem é especificada através de uma linguagem de modelagem web chamada WebML [26]. O modelo gerado sofre uma série de transformações e como produto final tem-se uma aplicação *web* compatível com a arquitetura MVC.

Brambilla [5] estende a sua proposta anterior [6] incluindo *web semântica* através de uma metodologia para desenvolver *Web Services* semânticos utilizando o framework WSMO que inclui um modelo conceitual WSMO, uma linguagem WSML e um ambiente de execução WSMX e uma ferramenta de modelagem WSMT. A busca por *Web Services* é feita pelo *Glue*, uma *engine* compatível com WSMO para busca de *Web Services*. Uma linguagem para expressar a orquestração da composição em WSMO está em desenvolvimento, sendo que a composição é executada em uma extensão com suporte a web semântica do WebRatio.

Manolescu [11] Apresenta uma abordagem conceitual para o projeto de *Web Services* formalmente definida através de uma extensão da linguagem WebML e implementada através do WebRatio. É feita uma análise comparativa entre uma composição expressa em BPEL4WS e WebML,

mostrando que WebML tem suporte as todos os padrões presentes em BPEL4WS (a maioria de forma primitiva e outros através de arranjos) e ainda possui suporte a outros padrões como *"Multi Merge"*, *"Discriminator"*, *"Milestone"*, entre outros.

Barret [1] propõe o uso de UML 2.0 e MDA para composição de *Web Services* partindo das *interfaces* (WSDL) e gerando, através de transformações, uma composição executável expressa em WS-BPEL. Em [3] o trabalho é evoluído e nos é apresentado padrões de distribuição, que expressa como é feito o *"deploy"* de uma composição, onde cada participante teria exposto (como *Web Service*) um serviço para execução de composição remota. Finalmente em [2] é proposta o uso de web semântica para descrever *Web Services* visando à geração semi-automática da composição executável.

D'Ambrogio [8] descreve uma abordagem para medir desempenho de uma composição expressa em BPEL através de P-WSDL, uma extensão do WSDL orientada a desempenho. Os processos BPEL são descritos em uma extensão de UML (*UML Profile for Automated Business Processes* [27]).

Narendra [13] Apresenta uma abordagem dirigida a requisitos. Os requisitos são separados em duas partes – funcionais e extra-funcionais. Os requisitos funcionais são expressos pelos compromissos individuais de cada *Web Service* para compor outro *Web Service,* requisitos extra-funcionais são as regras que condicionam o comportamento de cada um dos web services. É utilizado um modelo *workflow* para expressar o processo a ser modelado. Como existe diversos *Web Service* com a mesma funcionalidade, porém operando sobre diversos contextos é utilizado Ontologias durante a composição e a execução para dar ciência do contexto ai ambiente de execução de cada *Web Service*.

Charfi [7] apresenta uma solução baseada em *middleware* para dar suporte a requisitos não funcional em orquestrações com linguagens baseadas em processos, como BPEL, já que a mesma não possui tal suporte. A configuração das propriedades não funcionais como segurança, persistência, confiabilidade, etc., são expressas através de arquivos XML. A composição e os serviços de *middleware* são integrados através de uma extensão de orientada a aspectos de BPEL, AO4BPEL [28].

Chi [29] propõe uma plataforma de composição baseada em Redes de Petri para a composição de *Web Service* dividida em três módulos: um editor visual para definição conceitual do serviço a ser composto utilizando um mecanismo baseado em Redes de Petri; um módulo de linguagem de *"script"* para transformação do modelo conceitual em outras linguagens de composição; um módulo para validar o modelo expresso em Redes de Petri. Esta abordagem não possui "*engine"* de execução.

4.1 Ferramentas para composição de serviço

Nesta seção será descrito todo o ferramental existente na atualidade que trabalha com composição de serviços e com isso saberemos o que já tem sido feito e o que pode ser melhorado com relação a essas técnicas.

JOpera

Dando inicio a análise das ferramentas sobre composição de serviços vamos falar sobre o JOpera [30]. O JOpera é uma ferramenta de modelagem e execução de composição de serviços feito em cima do Eclipse, que é uma plataforma de aplicações em geral.

O JOpera usa uma linguagem própria para representar uma composição. A JVCL é uma linguagem baseada em XML e é uma linguagem de mais alto nível em comparação com outras linguagens de descrição/execução de composições de serviço, como BPEL. Ela foi projetada para não compor apenas *Web Services*, mas também serviços em outras tecnologias.

A modelagem das composições com o JOpera é baseada em Diagramas de fluxo de dados. O sistema de controle de cada atividade da Composição é feito através de duas construções muito simples mas igualmente poderosas presentes em cada atividade: Activation e Condition. O Activation define quais atividades são pré-requisitos para que uma atividade possa executar. Essa construção é usada para definir, entre outras coisas, condições de seqüencialidade e paralelismo entre as atividades da composição. Já a construção Condition é usada para definir quando uma atividade deve executar ou não levando em consideração o estado atual das variáveis da composição visto que os pré-requisitos definidos no Activation já foram alcançados.

O *Deploy* e a execução de uma composição são feitos internamente à ferramenta. O JOpera não necessita de nenhum servidor *Web* ou de qualquer outra ferramenta para executar as suas composições. Antes de executar a composição, o JVCL é transformada em código Java para depois ser efetivamente executada pela *engine* do JOpera. A Engine é interna ao JOpera. Além disso, a *engine* do JOpera possui um *Web Service* de controle onde se pode visualizar os status das composições que estão executando no momento, bem como implantar ou remover composições remotamente.

As interfaces que são usadas pelo JOpera são descritas em WSDL padrão. Essa característica indica uma preocupação dos projetistas com a compatibilidade das composições criadas pelo JOpera.

O JOpera possui a funcionalidade de depuração onde cada passo da composição, cada transição entre as atividades são visualizadas e a ferramenta mantém para o desenvolvedor informações sobre as variáveis envolvidas em cada operação.

Outra característica dessa ferramenta é o fato de ela estar sendo sempre melhorada inclusive sendo usada para outros tipos de Serviços mais específicos, como é o caso do suporte que ela fornece à *GridServices*.

NetBeans (Módulo BPEL)

O NetBeans é uma das ferramentas mais completas do mercado. Ela possui módulos para se trabalhar com uma infinidade de Linguagens de Programação (como: Java, C/C++, Ruby, entre outros), Tecnologias (Hibernate, STRUTS, ...) e com diversos tipo de aplicação (Desktop e Web). Um desses módulos é do de BPEL. Com esse módulo a ferramenta se transforma numa ferramenta para composição de *Web Services*. No restante dessa seção, iremos detalhar as principais funcionalidades desse módulo de criação de Composição de *Web Services*.

A primeira característica a ser mencionada aqui é a Interface Gráfica. Dentre todas as ferramentas analisadas durantes a confecção desse documento, a interface do NetBeans foi superior. A interface no NetBeans foi feita visando facilitar o trabalho do desenvolvedor da aplicação ao máximo. A forma em que os componentes gráficos são dispostos no Editor é bem clara e a ferramenta prima pela organização não permitindo uma maior liberdade de escolha sobre a disposição dos componentes no ambiente. Todo componente

que se pretende incluir na composição tem um conjunto de lugares pré-determinados para ficar. Essa falta de liberdade sobre a localização dos componentes foi tão bem pensada que chega a ser uma característica boa dessa ferramenta. Mas o número de detalhes que a interface tem pode atrapalhar um pouco no começo do uso.

O NetBeans se encaixa no grupo de ferramentas que, a nível de modelagem da composição, fornece apenas uma interface gráfica para a linguagem em que as composições são descritas, ou seja, essa ferramenta fornece um novo tipo de visualização para uma composição em BPEL. Esse tipo de abordagem exige que o desenvolvedor da composição precise ter um conhecimento considerável da linguagem de execução em questão, no caso BPEL.

Para executar as composições criadas, o NetBeans já vem integrado com um Servidor de Aplicação desenvolvido pela Sun *Microsystems* chamado GlassFish[?]. Assim como a *Engine Active BPEL* precisa de um servidor de aplicações para ser executado, a *engine* utilizada pelo NetBeans necessita utiliza o GlassFish.

O NetBeans ainda tem um Módulo muito interessantes de testes. Esse módulo já faz parte do NetBeans para BPEL e se baseia em ferramentas de testes de unidade, como JUnit, para realizar testes de caixa preta em Composições de *Web Services*. Esses testes funcionam de forma parecida com os testes JUnit. O NetBeans monta uma requisição SOAP com os dados arbitrários fornecidos pelo usuário, essa requisição é então enviada para o WebService que devolve como resposta um XML que será analisado pela ferramenta que vai comparar com uma espécie de gabarito e assim dizer se o resultado está correto ou não, fazendo assim o teste passar ou falhar, respectivamente.

Quanto ao foco desse trabalho, essa ferramenta não tem muito que acrescentar. Ela não trabalha com níveis de abstração mais altos que uma "simples" visualização gráfica de BPEL que poderiam gerar alguma independência da composição à linguagem de execução que ela usaria pra executar.

O NetBeans é um Freeware.

WebRatio

O WebRatio é uma ferramenta que tem um foco diferente das ferramentas que foram apresentadas até agora. Ela é uma ferramenta que essencialmente é usada para a criação de aplicações Web (em HTML, por exemplo) e até *Web Services*. Mas, como na criação de *Web Services* ela também pode usar outros *Web Services* (criando assim uma composição) essa ferramenta também passou a ser alvo desse estudo.

Assim como nas outras ferramentas, começamos pela Interface Gráfica. O WebRatio possui uma interface bem diferente das outras ferramentas. Até porque o foco dessa ferramenta é diferente. A interface sai bem da linha das ferramentas baseadas em BPEL e chega a se aproximar um pouco do JOpera por ser algo mais genérico mas, por outro lado, o grande numero de elementos que são usados apenas para a criação de Aplicações Web deixa pouco familiar para Programadores de Composições. A forma de estruturar a seqüência com que as operações são realizadas é feita graficamente usando setas que conectam operações que são feitas seqüencialmente.

A linguagem de execução utilizada pelo WebRatio é a WebML [33]. Essa linguagem já foi comentada acima como uma linguagem para modelagem visual de Aplicações Web, através dela e de uma série de transformações o WebRatio consegui gerar aplicações web tipicamente MVC (*Model-View-Control*) em várias tecnologias (Java/JSP, AJAX , HTML, XHTML).

O WebRatio possui um servidor Web integrado, o TomCat. Com o TomCat, o WebRatio pode implantar as aplicações que foram geradas através das transformações do WebML. Uma das coisas que o WebRatio gera automaticamente é o WSDL de todas os *Web Sevices* que ele cria.

Como o foco da ferramenta é a criação de Aplicações Web que não envolvem muita programação ou regras de negócio, ela não possui uma ferramenta de *Debug* muito bem elaborada como o visto nas outras ferramentas. Podemos falar o mesmo sobre alguma estrutura para testes que é inexistente.

Oracle solucion (Oracle SOA Suite + JDeveloper)

A Oracle também está interessada nesse tipo de arquitetura de software (SOA), tanto que Criou um conjunto de ferramentas e aplicativos que foram criados para suportar esse tipo de aplicação. Esse conjunto de ferramentas "*100 percent standards-based*" promete ser totalmente inter-operável com infra-estruturas legadas se qualquer empresa. Esse é o *Oracle SOA Suite*, um conjunto de aplicações que dá suporte a construção, implantação e gerenciamento.

O JDeveloper é uma ferramenta também da Oracle, totalmente integrada com o *SOA Suite*, que oferece recursos para o desenvolvimento de composições de serviço utilizando um poderoso editor gráfico e que gera código BPEL para a composição.

Como mencionado acima, o JDeveloper tem uma poderosa e rica interface gráfica. Vale reparar as semelhanças entre os editores gráficos do JDeveloper e do NetBeans, os editores gráficos, as formas em que os componentes são dispostos no editor, as formas que eles são instanciados pelo usuário, tudo isso é muito parecido. Elas diferem mais na hora de detalhar os atributos ou funcionalidades dos componentes BPEL da composição. O NetBeans consegui ser ainda mais "gráfico" que o JDeveloper, o JDeveloper ainda contém muitas coisas que não são resolvidas graficamente, ou seja, essas coisas necessitam de um conhecimento maior da tecnologia BPEL.

Outra semelhança com o NetBeans é que as duas ferramentas trabalham com BPEL e são ferramentas que geram código diretamente para BPEL e as suas interfaces gráficas são totalmente ligadas a essa linguagem de execução. Essa ferramenta reforça mais uma vez a importância de BPEL na indústria. Mas, mais uma vez, temos uma ferramenta que não aumenta o nível de abstração em relação à linguagem de execução em que ela vai ser definida posteriormente.

Todo o suporte para execução da composição criada pelo JDeveloper é dado pelo *SOA Suite* e suas aplicações. Dentro do *SOA Suite* tem, dentre outras coisas, uma *engine* desenvolvida pela Oracle para executar BPEL.

O JDeveloper possui uma ferramenta de testes bastante interessante. Os testes são divididos em dois tipos: *Unit Tests* e *Composite Test*. Todos os dois tipos de teste são baseados no JUnit e consiste em Analisar a resposta de

um método através do conhecimento prévio de suas entradas. A diferença entre o *Unit* e o *Composite Test* é que no Unit Test o comportamento dos partners é apenas simulado e fazem parte dos parâmetros do teste; no caso do Composite Test os *partners* são realmente invocados, esse teste é uma simulação da composição no mundo real.

Ainda falando de testes, o JDeveloper possui uma ferramenta que verifica quanto da composição é "coberta" pelos casos de teste, ou seja, quais as possibilidades que estão sendo previstas pelo engenheiro de teste.

ActiveVOS Designer

O ActiveVOS Designer é o sucessor da ferramenta Active BPEL Designer da empresa "Active Endpoints". O Active BPEL Designer é uma ferramenta muito conhecida na comunidade criadora de Composição de Serviços. Ela é utilizada em vários trabalhos como em Lins [43] não ó porque ela foi construída sobre uma plataforma conhecida, mas também porque ela é uma ferramenta *Free* e *OpenSource*.

Assim como na versão anterior, o ActiveVOS Designer é uma ferramenta que facilita a criação de uma composição apenas através de uma abstração de BPEL por uma interface gráfica. Mas diferentemente do que ocorre no NetBeans sua interface gráfica não é tão boa e exige do usuário um conhecimento grande a respeito de BPEL.

Diferentemente de sua antiga versão, o ActiveVOS Designer agora trabalha com uma Engine de Execução BPEL externa. O ActiveVOS Server que é a ultima versão do também muito conhecida *Engine Active BPEL* agora é totalmente externa ao ActiveVOS Designer.

No Processo de Depuração (Debug) as aplicações de Designe e o Server se conectam entre si para prover essa funcionalidade para o usuário.

Nessa nova versão foi criado um sistema de testes chamados de B-Unit (Business Unit) como uma variante do sistema de testes de unidade, JUnit. Que funciona de maneira semelhante ao JUnit onde uma entrada é fornecida à composição e a saída esperada é comparada com a saída que efetivamente é dada pela Composição.

Assim como o NetBeans, essa ferramenta não possui um nível de abstração muito alto para a construção de Composição de Serviços. Não utiliza modelos.

Eclipse (BPEL Project)

O projeto BPEL [44] da plataforma Eclipse é um projeto ainda em fase de desenvolvimento. Hoje, ele ainda não possui nem arquivos binários. Só é possível executar essa ferramenta baixando o código dela diretamente do CVS de desenvolvimento e compilá-la para depois executar.

Um dos pontos fortes dessa ferramenta pretende prover é o suporte total para as especificações de BPEL [44] que é uma versão mais recente dessa linguagem de execução.

Como já foi mencionada acima, essa ferramenta ainda está em fase de construção e ainda não possui algumas funcionalidades comuns entre ferramentas de composição, como, por exemplo, DEBUG.

A interface gráfica dessa ferramenta é comparável com outras ferramentas, como WebRatio.

Essa ferramenta também apenas trabalha como um editor para BPEL. E contém todas as problemáticas já mencionadas nas ferramentas acima que também possuem essa característica.

5 Conclusões e Trabalhos Futuros

Nesse trabalho foi proposta uma abordagem orientada a modelos para a composição de serviços. Essa abordagem possibilitou um aumento do nível de abstração durante a construção e é uma alternativa ao modo de construção mais adotado pela indústria que é baseado em construção de Processos de Negócio e em BPEL4WS. Esse aumento de nível de abstração possibilita uma construção mais rápida dessas composições e ainda serve de base para operações mais complexas, como Evolução Composicional, por exemplo.

Os novos desafios gerados por esse trabalho incluem:

- O término da implementação dessa ferramenta incluindo nela características padrão de ferramentas de modelagem de composição de serviços, como Debug, suporte a Testes de unidade e uma integração mais forte com a engine, que no caso do *MoSC Modeling Tool* é a *MoSC Auxiliary Engine*;

- Suporte do *MoSC Modeling Tool* e da MoSC-ML para descrição Semântica da composição e das partes que a constitue.
- Otimização do código MoSC-ML gerado pelo módulo *Transformer* do *MoSC Modeling Tool*;
- Implementação do restante da Arquitetura proposta, que inclui o *MoSC Repository* e o *MoSC Auxiliary Engine*;
- Melhoria dos aspectos evolucionais, como: modelos evolucionais e evoluções estruturais dos modelos das composições.

Alguns desses novos desafios já estão sendo alvo de trabalhos de mestrado e doutorado do Centro de Informática da Universidade Federal de Pernambuco. ae ae

6 Referências Bibliográficas

[1] Barrett, R. & Pahl, C. (2006), Model Driven Design of Distribution Patterns for Web Service Compositions, *in* 'Proc. International Conference on Web Services ICWS '06', pp. 887--888.

[2] Barrett, R. & Pahl, C. (2006), Semi-Automatic Distribution Pattern Modeling of Web Service Compositions using Semantics, *in* '10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06)', IEEE COmputer Society, , pp. 1-4.

[3] Barrett, R.; Patcas, L. M.; Pahl, C. & Murphy, J. (2006), Model driven distribution pattern design for dynamic web service compositions, *in* 'ICWE '06: Proceedings of the 6th international conference on Web engineering', ACM, New York, NY, USA, pp. 129--136.

[4] Belhajjame, K. & Embury, S. M. (2006), A Model-Driven Approach to Service Composition in Virtual Enterprises, *in* 'Proc. IEEE International Conference on Services Computing SCC '06', pp. 214--221.

[5] Brambilla, M.; Ceri, S.; Facca, F. M.; Celino, I.; Cerizza, D. & Valle, E. D. (2007), 'Model-driven design and development of semantic Web service applications', *ACM Trans. Interet Technol.* **8**(1), 3.

[6] Brambilla, M.; Ceri, S.; Fraternali, P.; Acerbis, R. & Bongio, A. (2005), Model-driven design of service-enabled web applications, *in* 'SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data', ACM, New York, NY, USA, pp. 851--856.

[7] Charfi, A. & Mezini, M. (2005), Middleware services for web service compositions, *in* 'WWW '05: Special interest tracks and posters of the 14th international conference on World Wide Web', ACM, New York, NY, USA, pp. 1132--1133.

[8] D'Ambrogio, A. & Bocciarelli, P. (2007), A model-driven approach to describe and predict the performance of composite services, *in* 'WOSP '07: Proceedings of the 6th international workshop on Software and performance', ACM, New York, NY, USA, pp. 78--89.

[9] Gronmo, R. & Jaeger, M. C. (2005), Model-Driven Semantic Web Service Composition, *in* 'APSEC '05: Proceedings of the 12th Asia-Pacific Software Engineering Conference', IEEE Computer Society, Washington, DC, USA, pp. 79--86.

[10] Issa, H.; Assi, C. & Debbabi, M. (2006), QoS-Aware Middleware for Web Services Composition - A Qualitative Approach, *in* 'Proc. 11th IEEE Symposium on Computers and Communications ISCC '06', pp. 359--364.

[11] Manolescu, I.; Brambilla, M.; Ceri, S.; Comai, S. & Fraternali, P. (2005), 'Model-driven design and deployment of service-enabled web applications', *ACM Trans. Interet Technol.* **5**(3), 439--479.

[12] Matsumura, M. (2006), 'SOA Adoption Blueprint “Generico” Working Draft 0.2', Technical report, OASIS.

[13] Narendra, N. C. & Orriens, B. (2007), Modeling web service composition and execution via a requirements-driven approach, *in* 'SAC '07: Proceedings of the 2007 ACM symposium on Applied computing', ACM, New York, NY, USA, pp. 1642--1648.

[14] Oberleitner, J.; Rosenberg, F. & Dustdar, S. (2006), A Lightweight Model-Driven Orchestration Engine for e-Services, *in* 'Technologies for E-Services', pp. 48-57.

[15] OMG (2005), 'A Proposal for an MDA Foundation Model', Technical report, OMG.

[16] Orriens, B.; Yang, J. & Papazoglou, M. (2003), Model Driven Service Composition, *in* 'Service-Oriented Computing (ICSOC)', pp. 75-90.

[17] Papazoglou, M. P.; Traverso, P.; Dustdar, S. & Leymann, F. (2006), 'Service-Oriented Computing Research Roadmap, Technical report, Tilburg University, Tilburg, The Netherlands.

[18] Pfadenhauer, K.; Dustdar, S. & Kittl, B. (2005), Challenges and Solutions for Model Driven Web Service Composition, *in* 'Proc. 14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise', pp. 126--131.

[19] Schattkowsky, T. & Mueller, W. (2004), Model-based specification and execution of embedded real-time systems, *in* 'Proc. Design, Automation and Test in Europe Conference and Exhibition', pp. 1392--1393 Vol.2.

[20] Singhal, S.; Pruyne, J. & Machiraju, V. (2008), 'Picasso: A Service Oriented Architecture for Model-based Automation'(HPL-2007-50R1), Technical report, HP Laboratories Palo Alto.

[21] Skogan, D.; Gronmo, R. & Solheim, I. (2004), Web Service Composition in UML, *in* '8th IEEE Intl Enterprise Distributed Object Computing Conf (EDOC 2004)', pp. 1-11.

[22] Tsai, W.; Chen, Y.; Bitter, G. & Miron, D. (2004), Introduction to Service-Oriented Computing, *in* 'IFIP International Federation for Information Processing 2004', pp. 294-310.

[23] Zeng, L.; Benatallah, B.; Dumas, M.; Kalagnanam, J. & Sheng, Q. Z. (2003), Quality driven web services composition, *in* 'WWW '03: Proceedings of the 12th international conference on World Wide Web', ACM, New York, NY, USA, pp. 411--421.

[24] Zeng, L.; Benatallah, B.; Ngu, A.; Dumas, M.; Kalagnanam, J. & Chang, H. (2004), 'QoS-aware middleware for Web services composition', *IEEE Transactions on Software Engineering* **30**(5), 311--327.

[25] Ceri, S., Fraternali, P., Acerbis, R., Bongio, A., et al.: Architectural Issues and Solutions in the Development of Data-Intensive Web Applications, CIDR2003, Asilomar.

[26] Ceri, S., Fraternali, P., Bongio, A.: Web Modeling Language (WebML): a modeling language for designing Web sites. WWW9/Computer Networks 33(1-6): 137-157 (2000).

[27] T. Gardner, UML Modelling of Automated Business Processes with a Mapping to BPEL, First European Workshop on Object Orientation and Web Services (in conjunction with ECOOP'03), Darmstad, Germany, July 2003 (see also http://www.ibm.com/developerworks/webservices/library/ws-uml2bpel/).

[28] A. Charfi and M. Mezini. Aspect-Oriented Web service Composition with AO4BPEL. In Proc. ECOWS 2004, volume 3250 of LNCS, 2004.

[29] Chi, Y. & Lee, H. A formal modeling platform for composing web services Expert Syst. Appl., Pergamon Press, Inc., 2008, 34, 1500-1507.

[30] C. Pautasso and G. Alonso. JOpera visual composition language. Journal of Visual Languages and Computing (JVLC) 16 (2005) 119–152.

[31] WS-DIAMOND (2006), 'Requirements, application scenarios, overall architecture, and test/validation specification, common working environment and standards', Technical report, University of Torino.

[32] Cardelli, L. & Davies, R. (1999), 'Service Combinators for Web Computing', IEEE TRANSACTIONS ON SOFTWARE ENGINEERING 25(3), 309-316.
[33] Milanovic, N. & Malek, M. (2004), 'Current Solutions for Web Service Composition', IEEE INTERNET COMPUTING November-December, 51-59.
[34] Woodman, S.; Palmer, D. & Shrivastava, S. (2004), Notations for the Specification and Verification of Composite Web Services, in '8th IEEE Intl Enterprise Distributed Object Computing Conf (EDOC 2004)', pp. 1-12.
[35] Aalst, W. M. v. d.; Dumas, M. & Hofstede, A. H. t. (2003), Web Service Composition Languages: Old Wine in New Bottles?, in '29th EUROMICRO Conference (EUROMICRO'03)', IEEE COmputer Society, , pp. 1-8.
[36] Aksit, M. & Choukair, Z. (2003), Dynamic, Adaptive and Reconfigurable Systems Overview and Prospective Vision, in '23rd International Conference on Distributed Computing Systems Workshops (ICDCSW'03)'.
[37] Beek, M. H.; Bucchiarone, A. & Gnesi, S. (2007), Web Service Composition Approaches: From Industrial Standards to Formal Methods, in 'Second International Conference on Internet and Web Applications and Services (ICIW'07)', IEEE COmputer Society, , pp. 1-6.
[38] Benatallah, B.; Sheng, Q. Z. & Dumas, M. (2003), 'The Self-Serv Environment for Web Services Composition', IEEE INTERNET COMPUTING January-February, 40-48.
[39] Kephart, J. O. (2005), Research Challenges of Autonomic Computing, in '27th international conference on Software Engineering', pp. 15 - 22.
[40] Hibner, A. & Zielinski, K. (2007), Semantic-based Dynamic Service Composition and Adaptation, in 'IEEE Congress on Services (SERVICES 2007)', IEEE COmputer Society, , pp. 1-8.
[41] Bosse, T.; Both, F.; Hoogendoorn, M. & Treur, J. (2007), Specification of Adaptive Client-Tailored Product Models, in 'IEEE Congress on Services (SERVICES 2007)', IEEE COmputer Society, , pp. 1-8.
[42] Moo-Mena, F. & Drira, K. (2007), Reconfiguration of Web Services Architectures: A model-based approach, in 'IEEE Congress on Services (SERVICES 2007)', IEEE COmputer Society, , pp. 1-8.
[43] Lins, F.; Santos Junior, J. & Rosa, N. (2007), 'Adaptive Web Service Composition', ACM SIGSOFT Software Engineering Notes 32(4), 1-8.
[44] T. Andrews, F. Curbera, H. Dholakia, et. al. Business Process Execution Language for Web Services, Version 1.1 BPEL4WS specification, May 2003. Available at http://www.ibm.com/developerworks/library/ws-bpel/
[45] W3C. “SOAP Version 1.2”. Disponível em http://www.w3.org/TR/soap. 2003.
[46] W3C. Web Services Description Language (WSDL). Specifications published at http://www.w3.org/2002/ws/desc/.

Data e assinaturas

18 de dezembro de 2008

_____

André Ricardo da Silva Souza
(Aluno)

_____

Nelson Souto Rosa
(Orientador)