2
Conceitos Básicos

No capítulo anterior, mencionamos que os meta serviços de negociação e sintonização de QoS foram modelados através de frameworks em Unified Modeling Language, em (Gomes, 1999), definindo decisões de projeto particulares desse domínio. Várias dificuldades foram encontradas nessa abordagem, que serão evidenciadas no decorrer deste capítulo.

Mencionamos também no Capítulo 1, que a abordagem adotada neste trabalho baseia-se no uso de uma linguagem de descrição de arquitetura (ADL) para a descrição arquitetural formal dos meta serviços de orquestração de recursos. Ainda no Capítulo 1, algumas razões para o emprego de ADLs foram comentadas, para justificar a abordagem presente.

Este capítulo inicia salientando uma série de requisitos que deveriam estar presentes em UML para que pudesse efetivamente ser usada na descrição de arquiteturas de sistemas. Depois de advogar em prol do uso de ADLs, o capítulo prossegue apresentando os principais conceitos das Linguagens de Descrição de Arquitetura. A seguir é discutido como uma descrição em UML poderia ser mapeada em uma descrição na ADL Wright. O capítulo prossegue apresentando as diversas fases de um sistema com QoS, situando os diversos frameworks desenvolvidos por (Gomes, 1999), em um provedor de serviço e nos seus vários provedores de meta serviços. O capítulo termina com um levantamento de algumas ADLs, com suporte à QoS, que poderiam ser consideradas na descrição de provedores de serviço, e justifica o uso da linguagem Wright na descrição dos meta serviços.

2.1.
Aplicação de UML na descrição arquitetural de sistemas

Estudos recentes apresentam técnicas para descrição arquitetural usando UML (Hofmeister et al., 1999) (Garlan et al., 2000) (Medvidovic et al., 2002). No

uso de UML como base para a descrição arquitetural, é preciso levar em conta pelo menos três aspectos desejáveis, como apontado em (Garlan et al., 2000):

- *Casamento de semântica*: o mapeamento deve respeitar a semântica documentada de UML e a intuição de quem modela em UML. A interpretação do modelo em UML deve ser próxima da descrição original em uma linguagem de descrição de arquitetura;
- *Clareza visual*: as descrições arquiteturais resultantes em UML devem trazer clareza visual a um projeto de sistema, evitando confusão e destacando os detalhes principais do projeto;
- *Completude*: certos conceitos arquiteturais (componente, conector, configurações arquiteturais, propriedades e estilo) devem ser representáveis no modelo em UML.

É necessário explicitar, entretanto, em que aspectos UML versão 1.1 falha em atender aos mesmos requisitos de uma ADL clássica para a descrição arquitetural de sistemas. Em (Medvidovic et al., 2002), é apresentado um conjunto mínimo de requisitos ausentes em UML para que possa efetivamente descrever arquiteturas de sistemas:

- UML deve ser mais bem servida na modelagem dos interesses estruturais, isto é, falta mais expressividade na descrição da configuração (ou da topologia) de um sistema;
- UML deve poder capturar estilos arquiteturais, como feito explicitamente e implicitamente por ADLs. Isso inclui a necessidade de um vocabulário de projeto padrão, topologias recorrentes e, possivelmente, um comportamento genérico do sistema;
- UML deve poder modelar os diferentes aspectos comportamentais de um sistema, como focalizado por várias ADLs (por exemplo, CSP, conjuntos de eventos parcialmente ordenados, π-calculus, lógica de primeira ordem). Nesse ponto, percebe-se que UML é surpreendentemente flexível em representar uma larga escala de interesses semânticos;
- UML precisa dar suporte à modelagem de uma larga escala de paradigmas de interação de componentes (específicos ou independente de um estilo particular). Essa exigência é sustentada

por uma das contribuições chave da pesquisa em arquitetura de softwares, o foco em interações de componentes, isto é, conectores de software como entidades de primeira ordem na modelagem de sistemas;

– Finalmente, uma exigência derivada das anteriores, UML deve poder capturar qualquer restrição que se levanta da estrutura, comportamento, interações e estilo(s) do sistema.

Não cabe aqui uma discussão sobre o fato de UML ser ou não uma ADL, tampouco uma profunda análise sobre a expressividade de UML em descrever arquiteturas de sistemas, até porque não há sequer um consenso sobre a definição exata de arquitetura de sistemas (vide Apêndice A). Cabe apenas salientar o mero fato de UML não atender aos requisitos estruturais e estilísticos. Essas deficiências de UML decorrem principalmente de que ela não permite a descrição precisa da semântica de frameworks, os conectores não podem ser modelados como entidades de primeira ordem e a descrição de estilos arquiteturais é muito fraca. Comparativamente, uma linguagem de descrição de arquitetura como Wright (vide Seção 2.2), que possui uma notação formal associada (CSP), permite que propriedades sejam checadas e que os padrões de estrutura e comportamento sejam definidos mais precisamente, o que ainda não é suportado por UML.

É provável que UML venha a possuir ferramentas de análise formal tão boas quanto Wright (Allen, 1997), assim que estabelecer um padrão de notação formal associado. Entre as opções para isso, OCL (Object Constraint Language) é a mais provável, por ser de fácil uso e entendimento e por já fazer parte da versão 1.1 de UML (Rational, 1997). Atualmente, sua semântica é informal, porém o principal esforço nesse aspecto da revisão 2.0 de UML, ainda em progresso, é o de torná-la formal. Outra opção, apontada em (Garlan et al., 2000), é de fazer o mapeamento dos conceitos de linguagens de descrição de arquitetura para UML Real-Time (UML-RT). No contexto da recomendação (IEEE P1471, 1999), que parece, enfim, emergir como um consenso sobre a descrição arquitetural, (Hofmeister et al., 1999) descreve uma proposta para se usar UML como base. A opção por uma abordagem usando uma ADL clássica é, no entanto, atualmente, mais vantajosa.

2.2. Linguagens de Descrição de Arquitetura

Segundo (Clements, 1996), "linguagens de descrição de arquitetura (ou apenas ADLs) são linguagens formais que podem ser usadas para representar a arquitetura de um sistema de software". Por arquitetura, entende-se os componentes que compreendem um sistema, as especificações comportamentais para esses componentes, e os padrões e mecanismos para as interações entre eles.

ADLs podem ser vistas como uma notação que permite a descrição precisa e a análise das propriedades externamente visíveis de uma arquitetura de sistema de software, oferecendo diferentes visões em diferentes níveis de abstração. Tal descrição serve como um esqueleto sobre o qual propriedades podem ser provadas e auxilia no suporte à evolução de sistemas já implementados.

Tomando por base outros aspectos, uma ADL pode também ser vista como uma linguagem voltada para especificar a estrutura de alto nível de uma aplicação (ou seja, a arquitetura conceitual), ao invés de se preocupar com detalhes de implementação de um módulo de código específico.

É interessante notar que há uma certa dificuldade em classificar uma linguagem como sendo uma ADL ou não (Medvidovic & Taylor, 1996) (Clements, 1996). Isso se deve, principalmente, ao fato que não há um consenso sobre uma definição formal para arquitetura de software, conforme mencionado anteriormente. Não havendo esse consenso, é difícil definir o que deve ser expresso por uma linguagem para que seja considerada suficiente para descrever arquiteturas de sistemas.

A lista abaixo, extraída de (Shaw et al., 1995), mostra algumas propriedades que ADLs devem apresentar:

- habilidade para representar componentes (primitivos ou compostos) juntamente com proposições de propriedades, interfaces e implementações;
- habilidade para representar conectores, juntamente com protocolos, proposições de propriedades e implementações;
- abstração e encapsulamento;
- tipos e checagem de tipos;
- habilidade para acomodar ferramentas de análise abertamente.

Em (Luckham & Vera, 1995) são listados os seguintes requisitos para que uma ADL seja, segundo seus próprios termos, considerada "suficientemente poderosa":

- abstração de componente;
- abstração de comunicação;
- integridade de comunicação (limitando a comunicação àqueles componentes conectados arquiteturalmente entre si);
- habilidade de modelar arquiteturas dinâmicas;
- habilidade de inferir sobre causalidade e tempo;
- suporte hierárquico de refinamento;
- relatividade, o mapeamento de comportamentos para (possivelmente diferentes) arquiteturas, como primeiro passo em direção à verificação de conformidade.

Pelo que foi exposto, percebe-se que, para uma linguagem de descrição de arquitetura ser completa, deve possuir duas características fundamentais: uma semântica precisa, resolvendo ambigüidades e auxiliando na detecção de inconsistências, e um conjunto de técnicas de suporte para o raciocínio sobre as propriedades do sistema.

ADLs devem disponibilizar ferramentas para modelar a arquitetura conceitual de sistemas de software, fornecendo tanto a sintaxe concreta como o framework conceitual para caracterizar arquiteturas. Esse framework reflete as características do domínio para o qual a ADL foi projetada e/ou o estilo arquitetônico, utilizando, tipicamente, teoria da semântica (CSP, redes de Petri, π-calculus, máquinas de estado finitas etc., ou através de uma meta-linguagem para especificar sua semântica).

2.2.1. Componentes, conectores e configurações arquiteturais

Uma descrição arquitetural é composta de elementos básicos ou blocos principais, que são os componentes, conectores e configurações arquiteturais.

Havendo o intuito de inferir informações semânticas sobre uma arquitetura, também devem ser modeladas as interfaces dos componentes e conectores.

Componentes são unidades de computação ou armazenamento de dados, geralmente com um estado associado. Dependendo do nível de abstração, componentes podem ser tanto meros procedimentos como aplicações completas.

A interface de um componente corresponde ao conjunto de seus pontos de interação, ou portas com o mundo externo, estabelecendo a semântica de um componente por meio de suas propriedades computacionais e restrições na utilização.

Conectores, por sua vez, são elementos usados para modelar interações entre componentes e as regras que as governam. Eles não realizam computação específica na aplicação e exportam com sua interface os serviços que esperam dos componentes que se ligam a eles.

A interface de um conector é um conjunto de pontos de interação, que se dá entre o respectivo conector e componentes ou entre ele e outros conectores ligados entre si. Isso permite a conectividade apropriada dos componentes, sua interação na arquitetura e fundamenta as configurações arquiteturais.

Os componentes e os conectores podem ser estruturados hierarquicamente e compostos a partir de elementos primitivos, sendo assim vistos como componentes ou conectores compostos. Essa estruturação hierárquica possibilita obter várias visões, em diferentes níveis de abstração, sobre o mesmo sistema.

Finalmente, as configurações arquiteturais, ou topologias, são grafos conexos de componentes e conectores que descrevem a estrutura arquitetural. Essa informação é ponto de apoio para verificar se os componentes apropriados estão conectados, se suas interfaces são compatíveis, se os conectores permitem a comunicação e se a semântica da configuração como um todo resulta no comportamento esperado.

É importante ressaltar que tal descrição arquitetural em blocos provê uma modelagem simples em diversos níveis de abstração. Possíveis configurações arquiteturais de um nível podem até ser mapeadas como componentes de um nível mais alto. Isso permite comunicar informações sobre um sistema de forma mais compreensível entre arquitetos e outros usuários, além de facilitar a reutilização e modularização de componentes abstratos através de tipagem, que dá oportunidade de instanciá-los muitas vezes em função das devidas necessidades.

Para aumentar o reuso e expressividade, a área de arquitetura de software também utiliza o conceito de estilo. De acordo com (Garlan & Shaw, 1993), "um estilo arquitetural define uma família de sistemas em termos de um padrão de estrutura organizacional ou, de forma mais específica, determina o *vocabulário* de componentes e conectores que podem ser usados em instâncias daquele estilo, junto com um conjunto de *restrições* na forma como eles podem ser combinados". Tais restrições podem ser topológicas (ex.: sem ciclos) ou relativas à semântica de execução. Quando uma configuração arquitetural segue o vocabulário de um estilo e respeita suas restrições, pode ser vista como uma instância daquele estilo.

2.3. Descrições arquiteturais de *Frameworks* Orientados a Objetos

O trabalho apresentado em (Arevalo, 2000) oferece uma metodologia para guiar um projetista no mapeamento entre as informações que descrevem um framework orientado a objetos e uma descrição arquitetural usando a ADL Wright (Allen, 1997). A intenção é obter uma arquitetura de software genérica que representa a família de aplicações resultante da instanciação do framework.

Em (Arevalo, 2000) são apontados, também, motivos que justificam a utilização de uma descrição formal para frameworks orientados a objetos:

• *Visão global e prévia do framework*. Um provável usuário pode saber de antemão se o framework vai atender a suas expectativas;

• *Reuso da arquitetura*. Provê uma visão de mais alto nível da arquitetura, potencializando o reuso em outros sistemas ou em mais de uma linguagem de programação;

• *Integração de frameworks*. Facilita a construção de sistemas a partir da integração de diversos frameworks existentes, em função da precisão da descrição comportamental, com as suposições arquiteturais de cada um deles descritas explicitamente;

• *Evolução e re-engenharia*. Fornece uma referência para a verificação de alterações em versões subseqüentes e permite formar hipóteses sobre a arquitetura que podem ser testadas no processo de engenharia reversa.

Os passos descritos para o mapeamento entre frameworks orientados a objeto e descrições arquiteturais em (Arevalo, 2000) são:

1) Identifique as principais classes do framework em termos do domínio do problema;
2) Cada classe é mapeada em um componente e cada possível relacionamento entre duas classes é mapeado como um conector, em termos da ADL Wright, evitando o relacionamento com classes de tipos simples (*integer*, *char*, *boolean*).
3) Os protocolos de cada classe são classificados como eventos iniciantes ou observados e todas as mensagens chamadas no corpo das mensagens são classificadas como eventos iniciantes;
4) Os protocolos para as portas e a computação dos componentes são construídos de acordo com a semântica das classes;
5) Os conectores são construídos usando as mensagens enviadas de uma classe para outra;
6) Identifique as variações de um componente (cada subclasse de uma classe raiz) e que outros componentes relacionados a ele devem ser alterados;
7) Identifique os componentes que representam pontos de flexibilização (*hot spots*), para saber quais são fixos e quais são candidatos a mudanças no caso de uma instanciação;
8) Até esse ponto já está pronta uma descrição arquitetural inicial. Estilos arquiteturais pré-definidos podem ser identificados em termos de um conjunto de classes ou apenas em função de componentes e conectores com um comportamento específico;
9) Execute as ferramentas de Wright para verificar diferentes propriedades, como ausência de *deadlock*;
10) Refine cada componente, levando em conta:
    a. Se há uma composição hierárquica de objetos cooperando entre si (definição de micro-arquitetura). A intenção é descobrir se há objetos compostos a partir de outros e se os serviços que eles oferecem são feitos usando esses objetos. Para haver uma composição hierárquica, é necessário que todos os objetos internos estejam dentro das fronteiras do objeto principal;
    b. Se há um conjunto de eventos ligados por uma escolha não-determinística, que indica a decisão do componente

considerando um estado interno. O interesse é evitar o uso da escolha não-determinística ao expressar o comportamento dos componentes.

11) Novos componentes podem ser descobertos, não necessariamente mapeados a partir dos conceitos do domínio. Esse é o caso quando se verifica um novo nível de descrição, geralmente necessitando que se estude o componente como uma micro-arquitetura e se repitam todos os passos anteriores para esses novos componentes;
12) Defina os protocolos de interação nos tipos das interfaces e a associação de pontos de flexibilização (*hot spots*) nos estilos. Isso permitirá tanto uma visão mais clara de como o framework é composto, quanto medir os impactos de possíveis mudanças na sua estrutura e no comportamento dos objetos.

A principal desvantagem dessa metodologia é que ela não pode ser automatizada. Como exemplo, o passo 1 obriga o projetista a ter um razoável conhecimento do domínio, além de ser subjetivo na identificação de classes principais para a maioria dos frameworks, o que não pode ser computacionalmente tratado.

Outro ponto crítico é que ela não foi suficientemente testada em vários frameworks e não há garantias que todos os casos particulares possam ser contemplados. Há o risco que o mapeamento de certos frameworks se torne excessivamente complicado.

Apesar dessas desvantagens, a existência de um método seqüencial e dirigido, por menos abrangente que seja, para o mapeamento entre um framework orientado a objeto e uma descrição arquitetural, no caso em Wright, acaba simplificando o trabalho manual.

Ao tentar se aplicar, na prática, essa metodologia para a descrição arquitetural dos frameworks de negociação e sintonização de QoS, foi percebida a ausência de maiores detalhes. Como exemplo disso, o passo 7 é voltado para a identificação de classes com pontos de flexibilização, porém não cita como

componentes que representam tais classes devem ser modelados em Wright. No presente trabalho, isso é feito através de esquemas de parametrização[1].

2.4. Provisão de QoS

Este trabalho segue a mesma terminologia apresentada em (Gomes, 1999), originada a partir da abordagem de serviços configuráveis. Conforme esse trabalho cita, a padronização dos termos é conveniente para promover sua independência com relação às escalas de distribuição e aos níveis de abstração em que podem ser aplicados.

Antes porém de serem apresentados os frameworks para provisão de QoS, assunto da Seção 2.4.2, a próxima seção define alguns princípios básicos para seu entendimento.

2.4.1. Princípios para Provisão de QoS

O conceito base é o de *ambiente de processamento e comunicação*, que é formado por um conjunto de componentes, por pelo menos uma *máquina de objetos*, responsável pelo processamento desses componentes, assim como por pelo menos um *provedor de serviços*, responsável pela comunicação entre eles, conforme ilustração da Figura 2.1. Por sua vez, um provedor de serviço se constitui em outro ambiente de processamento e comunicação, formado por componentes internos do serviço, por pelo menos uma máquina de objetos, responsável pelo processamento desses componentes, assim como por pelo menos um provedor de serviços, que também pode ser composto por outros componentes internos e assim sucessivamente. Desta forma, é conveniente se diferenciar os conceitos de *distribuição* e *abstração*. Enquanto a escala de distribuição tem a ver com a localização relativa dos componentes de um sistema, por sua vez, o nível de abstração refere-se à visão que se tem desses componentes, ou seja, a que

[1] O caso mais comum é aplicado no contexto do padrão de projeto Strategy (Gamma et al., 1995), que é modelado naturalmente como um parâmetro de computação em Wright (Allen, 1997), por ser comportamental, definindo uma família de algoritmos, os encapsulando e os tornando intercambiáveis, de forma independente dos clientes que os utilizam.

provedor de serviços (do aninhamento de provedores de serviços) eles estão ligados.

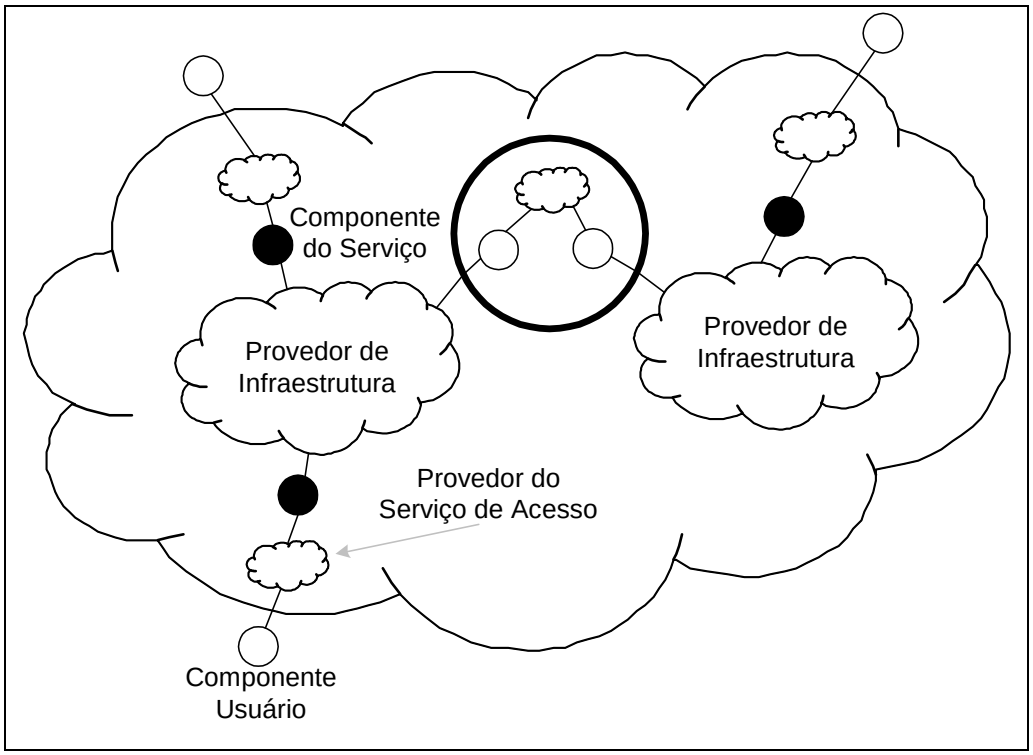

Figura 2.1 – Ambiente de processamento e comunicação

Não é suficiente descrever apenas a infra-estrutura de um provedor de serviços para que ele seja considerado operacional. É necessário definir, também, os conjuntos de estratégias, utilizadas pelos mecanismos de comunicação. Entre eles estão os mecanismos para provisão de QoS, que regulam a maneira como um provedor de serviços controla e gerencia a QoS oferecida. Esse conjunto de estratégias é chamado de *políticas de provisão de QoS* de um provedor de serviços e está intimamente associado à sua implementação real.

O termo QoS é definido em (Gomes, 1999) como "o conjunto das características (qualitativas e quantitativas) de processamento e comunicação suportadas por um serviço que permite a provisão da funcionalidade desejada por usuários do ambiente". A QoS pode, então, ser entendida como uma relação entre os requisitos dos usuários, o estado interno atual do provedor de serviços e os eventos na interface entre ambos.

Cada provedor de serviços em um aninhamento de provedores traz consigo sua *visão de QoS*. A "visão" (funcionalidade desejada) de QoS depende dos

usuários do serviço. Por exemplo, em uma transmissão de vídeo, se estamos no nível mais alto de abstração de serviços de um provedor, o usuário é geralmente o ser humano, que vai especificar a qualidade de serviço conforme sua percepção da mídia, como por exemplo: "vídeo colorido com qualidade de TV". À medida que vamos entrando nos provedores mais internos de um provedor de serviços, ou seja, à medida que o nível de abstração vai diminuindo, a qualidade de serviço passa a ser especificada em outros termos, como para o exemplo anterior: "vídeo em tempo real a uma taxa de 30 quadros por segundo e com uma resolução de aspecto 4:2:2". Mais internamente ainda, a visão de QoS pode mudar: "taxa de 162 Mbps, retardo máximo menor ou igual a 100 ms, taxa de perdas menor que 1%". Assim, ao passarmos para os níveis mais internos de um provedor, um *mapeamento* da QoS se fará necessário, traduzindo a "visão" de QoS de um provedor para as "visões" de seus provedores mais internos.

Ao analisar o papel de um provedor de serviço com QoS, dois princípios básicos são identificados:

- A especificação dos requisitos dos usuários;
- A provisão de mecanismos que garantam os requisitos dos usuários através do compartilhamento e da orquestração de recursos.

É desejável que a especificação dos requisitos dos usuários seja o mais abrangente possível porque a flexibilidade na definição de novos serviços está intimamente ligada à capacidade de expressar seus requisitos. De nada adianta, porém, um esquema abrangente se não há mecanismos que possam oferecer garantias para esses requisitos. Já que a complexidade em oferecer mecanismos de garantia é dependente de quais requisitos precisam ser atendidos, esses dois princípios estão muito relacionados. Por exemplo, a linguagem Xelha (Duran-Limon & Blair, 2000) permite que se especifiquem os requisitos de QoS de uma tarefa em termos de um conjunto finito e imutável de parâmetros, como retardo, vazão, taxa de perda de pacotes, *jitter* etc. Como conseqüência, os serviços que podem ser providos por essa plataforma são limitados, porém um modelo que se propõe a atender apenas a esses requisitos é muito mais simples de implementar do que se houvesse um esquema de especificação de requisitos mais flexível.

2.4.1.1. Parametrização de Serviços

Há três tipos de parâmetros diferentes relacionados com a provisão de QoS :

- Parâmetros de desempenho do provedor.
- Parâmetros de caracterização de carga.
- Parâmetros de especificação da QoS.

Os *parâmetros de desempenho do provedor* são aqueles que definem seu estado interno num dado momento. Esses parâmetros são especialmente importantes para o controle de admissão, já que são eles que serão levados em conta para verificar a possibilidade de aceitação, ou não, de uma requisição de serviço.

Parâmetros de caracterização de carga definem a dinâmica dos fluxos do usuário, enquanto *parâmetros de especificação da QoS* descrevem os requisitos de processamento e comunicação desejados pelos usuários. Juntos, eles são usados para definir os requisitos de um serviço.

2.4.1.2. Fases de Provisão de QoS

A provisão de QoS por parte de um provedor de serviço pode ser dividida em quatro fases: i) iniciação do provedor de serviços; ii) requisição de serviços; iii) estabelecimento de contratos de serviço; e iv) manutenção de contratos de serviço.

A descrição do estado interno do provedor de serviços é feita durante a fase de iniciação. Isso envolve a definição tanto da infra-estrutura de suporte aos serviços oferecidos quanto das políticas de provisão de QoS adotadas por aquele provedor.

Uma vez que um provedor de serviços tenha sido iniciado, usuários podem requisitar serviços a esse provedor. Isso é feito através da caracterização da carga e especificação da QoS desejada, levando em conta o nível de visão de QoS do usuário, que tem a ver, principalmente, com a qualidade de percepção da mídia e seu custo associado.

Durante a fase de estabelecimento de contratos de serviço, é feito um controle de admissão no provedor, que verifica se há recursos disponíveis para satisfazerem os parâmetros daquela requisição. Em caso positivo, cabe ao provedor de serviços alocar os recursos de forma a otimizar a quantidade de serviços que podem ser oferecidos. O provedor pode, também, rejeitar esse fluxo caso não possua recursos para atendê-lo ou, ainda, sugerir ao usuário uma outra especificação de QoS passível de ser satisfeita para que seja feita uma nova requisição. Esse mecanismo é chamado de *negociação de QoS*.

Na fase de manutenção de contratos de serviço, o provedor deve cuidar para que a QoS negociada seja mantida durante todo o tempo de uso do serviço, desde que o usuário esteja em conformidade com a carga caracterizada durante o estabelecimento do contrato. O provedor deve monitorar os diversos fluxos e atuar em caso de violações do contrato de serviço. Tais violações podem se dar por ambas as partes (usuário e provedor), e podem causar uma realocação de recursos, alertas ao usuário e até mesmo a interrupção no fornecimento do serviço. Esse conjunto de mecanismos que atuam durante a manutenção do serviço para garantir a QoS negociada no contrato é chamado de *sintonização de QoS*.

2.4.2. Frameworks para Provisão de QoS

O trabalho apresentado em (Gomes, 1999) descreve Frameworks para Provisão de Qualidade de Serviço (ou apenas Frameworks de agora em diante), onde a abordagem se centraliza na criação de sistemas configuráveis no tocante à introdução de novos serviços.

Os Frameworks compõem-se de: i) um Framework para Parametrização de Serviços; ii) Frameworks para Compartilhamento de Recursos, sendo um para Escalonamento e outro para Alocação de Recursos; e iii) Frameworks para Orquestração de Recursos, contendo um Framework para Negociação de QoS e outro para Sintonização de QoS.

Apesar de apresentar os *Frameworks* em detalhes, o trabalho exposto em (Gomes, 1999) não os descreve formalmente, reconhecendo nisso um aspecto crítico, especialmente em relação à imprecisão ao definir famílias de arquiteturas em *Unified Modeling Language* (Rational, 1997). Assim, a especificação formal

dos Frameworks para Orquestração de Recursos se tornou o foco desta dissertação.

2.4.2.1. Framework para Parametrização de Serviços

A abordagem genérica dos Frameworks de Provisão de QoS exige que eles não sejam limitados a categorias de serviço específicas. Isso implica diretamente a necessidade de um esquema de parametrização de serviços que seja plenamente flexível quanto à introdução de novos parâmetros de desempenho do provedor, de caracterização de carga ou de especificação de QoS. Importante, também, é que o esquema de parametrização de serviços seja modelado de tal forma que mantenha suficientemente flexíveis as estruturas de registro de medições e de mapeamento de parâmetros, para que as estratégias de monitoramento e mapeamento possam ser facilmente substituídas. É necessário, portanto, pensar no esquema de parametrização como uma forma de facilitar o mapeamento entre níveis de visão de QoS e o monitoramento de fluxo.

O Framework de Parametrização de QoS é "responsável por definir um esquema de parametrização que seja independente dos possíveis serviços a serem oferecidos" (Gomes, 1999). Para isso, o framework define uma estruturação hierárquica para a definição abstrata de parâmetros. O conceito de *categorias de serviço* define que políticas de provisão de QoS podem ser usadas, bem como um contexto através do qual toda manipulação de parâmetros é feita.

2.4.2.2. Frameworks para Compartilhamento de Recursos

A fim de facilitar a aplicação de diversos algoritmos de escalonamento e controle de admissão no mesmo recurso e assim oferecer um conjunto amplo e flexível de serviços em um único sistema, recursos são organizados em uma estrutura chamada *árvore de recursos virtuais*, como ilustrado pelo exemplo da Figura 2.2. Os frameworks de Compartilhamento de Recursos permitem a criação e gerenciamento de tais árvores.

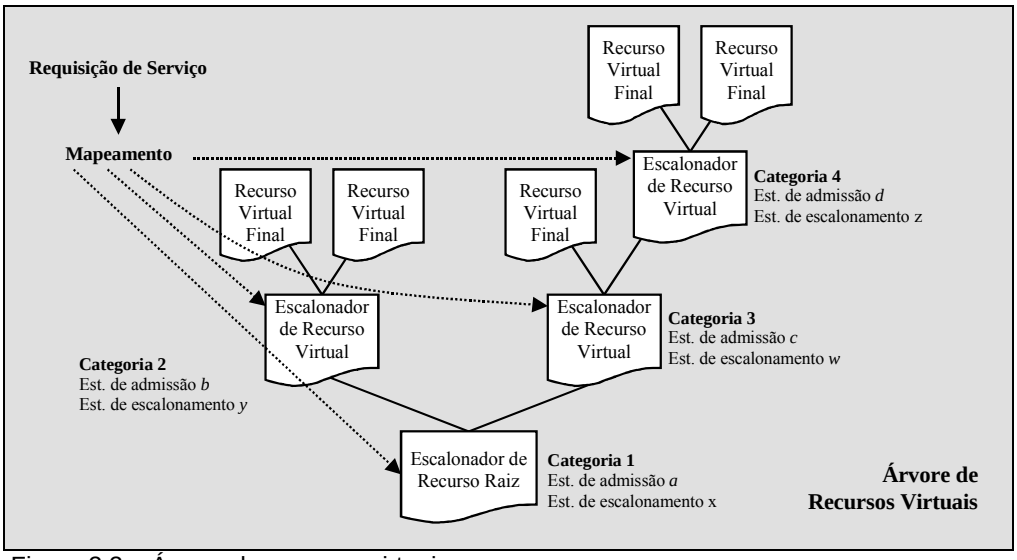

Figura 2.2 – Árvore de recursos virtuais

O conceito de *árvore de recursos virtuais* denota abstratamente a divisão hierárquica de parcelas de utilização de um ou mais recursos. *Recursos virtuais* representam parcelas de utilização de um recurso real, de um outro recurso virtual, ou ainda de um conjunto de recursos. O nó raiz de uma árvore é chamado de *escalonador do recurso raiz*, podendo esse *recurso raiz* corresponder a: i) um recurso virtual ou real (como banda passante, CPU, memória etc.); ii) um conjunto de recursos reais ou virtuais visto como um recurso único; iii) um conjunto de recursos reais ou virtuais vistos isoladamente. Em qualquer caso, o papel do escalonador do recurso raiz é distribuir parcelas de utilização daquele recurso raiz entre seus nós filhos. Cada um desses nós, por sua vez, distribui sua própria parcela de utilização entre seus nós filhos e assim sucessivamente até os nós folhas, que são chamados *recursos virtuais finais*. Os nós intermediários da árvore são chamados de *escalonadores de recursos virtuais*.

Cada nó na árvore, exceto as folhas, está associado a uma categoria de serviço (com um conjunto de parâmetros que a define) e algumas políticas de provisão de QoS correspondentes. As políticas incluem estratégias para escalonamento e admissão de fluxos, além de um componente *fábrica de recursos virtuais*. Nos Frameworks de Compartilhamento de Recursos, a *fábrica de recursos virtuais* permite a inclusão de um recurso virtual na lista de responsabilidades do escalonador e a configuração dos mecanismos de classificação e policiamento (controle de parâmetros usuário/sistema).

Mecanismos de classificação são responsáveis pelo redirecionamento de um fluxo de dados para o recurso filho apropriado. Mecanismos de policiamento devem verificar se fluxos do usuário estão de acordo com o tráfego caracterizado e, baseado nas estratégias de controle de parâmetros usuário/sistema, realizar certas ações para limitá-lo (como *traffic shaping*, descarte de parte do fluxo, etc).

O controle de admissão de um novo fluxo é baseado em informações sobre a utilização de recursos do escalonador associado à categoria de serviço requisitada. Se a admissão é possível, o recurso virtual pode ser criado através das *fábricas de recursos virtuais*. Quando o recurso raiz é um único recurso ou um conjunto de recursos visto como único, a admissão de novos fluxos do usuário é chamada primitiva, porque não há necessidade de negociação entre outros mecanismos. Em outras palavras, nesse caso, testes de admissão podem ser feitos diretamente sobre uma única árvore de recursos virtuais. De outra forma, quando o conjunto de recursos não pode ser visto como único, a admissão do fluxo é recursivamente delegada a árvores de recursos virtuais de níveis de abstração inferiores. Esse processo termina quando um recurso virtual com admissão primitiva é alcançado. Esse conceito ficará mais claro na próxima seção.

2.4.2.3. Frameworks para Orquestração de Recursos

Os Frameworks para Orquestração de Recursos são responsáveis por gerenciar os mecanismos de alocação e escalonamento. A Negociação de QoS modela os mecanismos de negociação e mapeamento durante as fases de requisição e estabelecimento, enquanto a Sintonização de QoS modela os mecanismos de monitoração e da própria sintonização durante a fase de manutenção.

2.4.2.3.1. Framework de Negociação de QoS

O Framework de Negociação de QoS é constituído de três tipos de elementos centrais: *Controladores de Admissão*; *Negociadores de QoS*; e *Mapeadores de QoS*. Ele foi construído de tal forma que possa ser recursivamente

aplicado nos vários níveis de abstração presentes em um ambiente, independente do modelo de integração com os níveis adjacentes (centralizado ou distribuído).

A requisição de um contrato de serviço é feita através de uma chamada a um controlador de admissão. Os parâmetros dessa chamada devem possuir informações que descrevam o comportamento do fluxo do usuário, através da caracterização de carga e da especificação de QoS.

Caso esse controlador de admissão seja relacionado a um recurso (provedor) composto por outros internos, sua função será acionar o mecanismo de negociação, seja centralizado ou distribuído, através de uma chamada ao negociador de QoS associado. Cabe a esse negociador de QoS a tarefa de identificar todos os recursos internos envolvidos no fornecimento do serviço requisitado[2] e dividir a parcela de responsabilidade pela provisão de QoS a cada um deles. Tal processo é feito em conjunto com um mapeador de QoS agregado ao negociador, que traduz a requisição de serviço do nível de visão de QoS do recurso em parâmetros condizentes com o nível de visão de QoS dos recursos internos envolvidos, levando em conta a distribuição de parcelas de responsabilidade de cada um deles.

Após a escolha dos recursos internos e das respectivas parcelas de QoS com os parâmetros mapeados, uma nova requisição de estabelecimento de contrato é feita a cada um dos recursos via os respectivos controladores de admissão. O processo se repete até chegar a um nível em que um controlador de admissão efetua os testes para aceitação, ou não, daquele fluxo do usuário diretamente sobre uma única árvore de recursos virtuais com admissão primitiva, sem exigir a negociação entre outros mecanismos.

A Figura 2.3 apresenta essa modelagem do Framework de Negociação de QoS, usando UML, conforme recentes atualizações (Gomes et al., 2001) do trabalho proposto em (Gomes, 1999).

A classe *AdmissionController* representa abstratamente o controlador de admissão. A requisição de estabelecimento de contrato de serviço é feita através do método *admit()*. As instâncias de *HighLevelAdmissionController* atuam em

[2] O responsável pela escolha de quais recursos internos serão envolvidos pode levar em conta mecanismos adicionais, como o roteamento, mas isso está sendo deixado fora do escopo deste trabalho.

níveis de abstração em que o recurso é composto de outros internos e, durante o estabelecimento de contratos de serviço, delegam parcelas de responsabilidade aos subsistemas envolvidos.

Por sua vez, as instâncias de *LowLevelAdmissionController* agem diretamente sobre árvores de recursos virtuais com admissão primitiva. Nesse caso, quando uma requisição de contrato de serviço é feita, utilizam uma instância de uma subclasse de *AdmissionStrategy* para testar a disponibilidade de recursos através do método *check()*. Elas estão, também, associadas a um *ResourceScheduler*, que gerencia recursos segundo a categoria de serviço relacionada. Essas duas associações reforçam o intuito de maximizar a utilização de recursos através do relacionamento entre escalonamento e estratégias de admissão com uma determinada categoria de serviço.

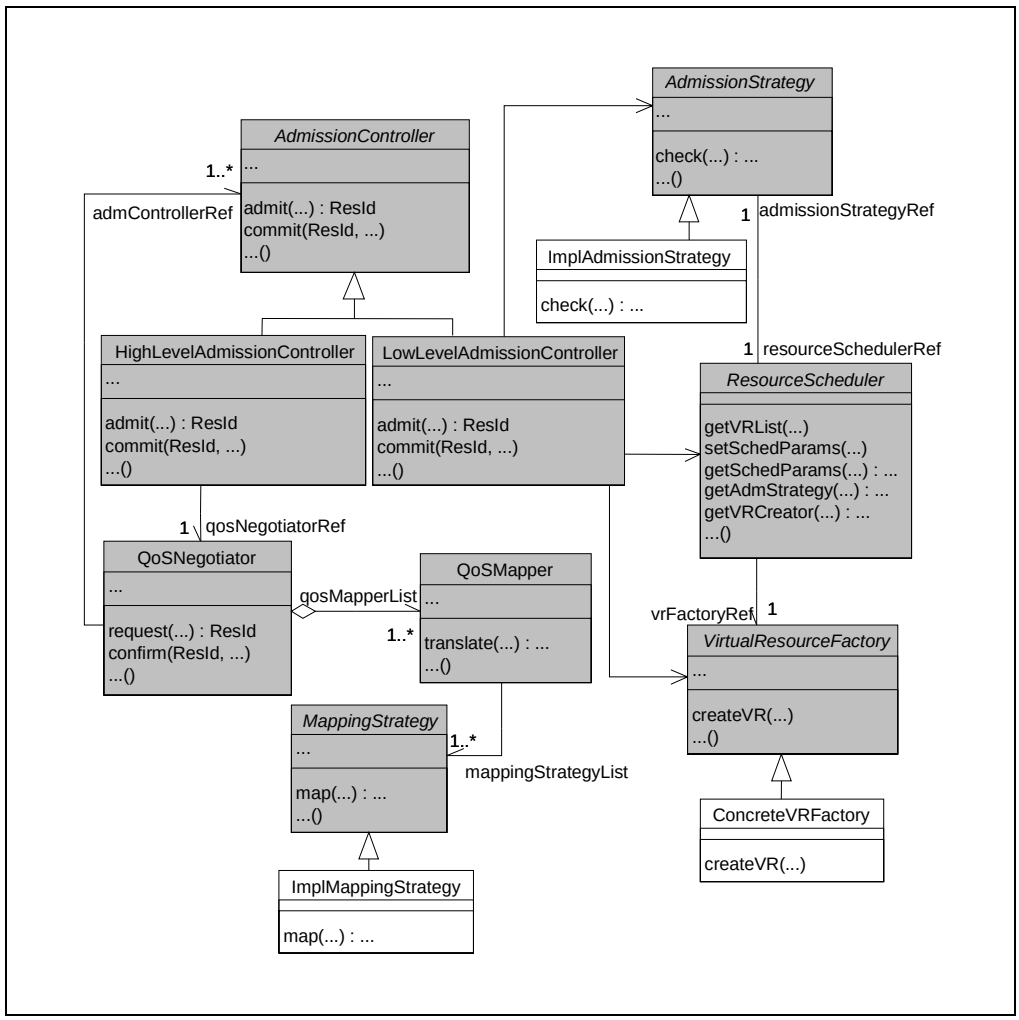

Figura 2.3 – Framework de Negociação de QoS

As classes *QoSNegotiator* e *QoSMapper* representam, respectivamente, os conceitos de negociadores e mapeadores de QoS. O método *request()*, definido na classe *QoSNegotiator*, é o responsável pelo mecanismo de negociação de QoS, conforme descrito anteriormente. Cada instância de uma subclasse de *QoSNegotiator* possui instâncias de *QoSMapper* agregadas (uma para cada controlador de admissão associado nos subsistemas internos), o que é representado pelo relacionamento *qosMapperList*.

A tradução entre níveis de visão de QoS inicia com uma chamada ao método *translate()* em uma instância correspondente de uma subclasse de *QoSMapper*, que retorna o resultado após disparar o método *map()* de uma estratégia de mapeamento, associada pelo relacionamento *mappingStrategyList*. Tal relacionamento visa uma maior flexibilidade quanto às categorias de serviços oferecidas por um provedor.

Os métodos *commit()* e *confirm()*, presentes respectivamente nas classes *AdmissionController* e *QoSNegotiator*, são utilizados para confirmar a reserva de recursos, descendo recursivamente os níveis de abstração até alcançar controladores de admissão primitivos. Quando isso ocorre, é executado o método *getVRCreator()* na classe *ResourceScheduler*, retornando uma referência a uma instância de *ConcreteVRFactory()*, onde deve ser criado o recurso virtual, por intermédio do método *createVR()*.

2.4.2.3.2. Framework de Sintonização de QoS

O Framework de Sintonização de QoS é composto principalmente de quatro elementos: *Controladores de Ajuste*; *Sintonizadores de QoS*; *Mapeadores de QoS* e *Monitores de Fluxo*. Ele não só modela a sintonização de QoS, ou seja, o mecanismo responsável por reorquestrar recursos em caso de iminência de violação de contratos de serviço durante seu fornecimento, como também os mecanismos de monitoramento da real QoS que se oferece ao usuário.

Depois que um serviço é admitido, cabe aos monitores de fluxo medirem a QoS sendo fornecida. Controladores de ajuste, relacionado a uma árvore de recursos virtuais com admissão primitiva, e sintonizadores podem estar associados

a monitores de fluxo. Quando isso acontece, eles avaliam as medições efetuadas pelos monitores associados em busca de indícios que apontem se um contrato de serviço está prestes a ser rompido. Esse processo pode ocorrer de duas formas: efetuando chamadas periodicamente às medições de estatísticas no monitor, ou no sentido inverso, com o monitor notificando tais medições sem ser acionado.

Quando uma violação da QoS contratada ocorrer, ou estiver prestes a ocorrer, o controlador de ajuste primitivo ou sintonizador de QoS que a percebeu, através de indicação do monitor associado, tenta corrigir o problema, através de uma nova admissão na árvore de recursos virtuais, no caso do controlador de ajuste, ou através de uma nova orquestração de recursos, no caso do sintonizador. No caso de uma nova orquestração, tal qual a fase de negociação, o processo é feito em conjunto com um mapeador de QoS agregado ao sintonizador, que traduz a requisição de serviço do nível de visão de QoS do recurso composto em parâmetros condizentes com o nível de visão de QoS dos recursos internos envolvidos. Uma nova requisição de estabelecimento de contrato é feita a cada um dos recursos via seus respectivos controladores de ajuste. O processo se repete até chegar a um nível em que um controlador de ajuste efetua os testes para aceitação sobre uma única árvore de recursos virtuais com admissão primitiva.

Caso a degradação da QoS não possa ser corrigida, o controlador de ajuste primitivo ou o sintonizador de QoS que a percebeu deve enviar um alerta a um agente de mais alto nível de abstração. O agente de mais alto nível poderia ser o próprio usuário, que estaria sendo informado que o nível contratado de QoS não pode mais ser mantido; ou um sintonizador de QoS de nível de abstração superior. Nesse último caso, o sintonizador tenta uma nova orquestração, da mesma forma como se tivesse percebido o problema através de um monitor.

A Figura 2.4 apresenta a modelagem do Framework de Sintonização de QoS usando UML. Novamente, a figura está baseada em recentes atualizações (Gomes et al., 2001) do trabalho proposto em (Gomes, 1999).

A classe *AdjustmentController* representa os controladores de ajuste. O método *adjust()* é disparado sempre que for preciso reorquestrar os recursos para manter o nível contratado de serviço. Em instâncias de *HighLevelAdjustmentController*, esse método apenas executa o *tune()* na instância de *QoSTuner* associada pelo relacionamento *qosTunerRef*. Em instâncias de *LowLevelAdjustmentController*, por sua vez, o método *adjust()* atua diretamente

sobre os escalonadores de recursos virtuais, de acordo com a estratégia de ajuste adotada. Em tais instâncias de mais baixo nível, um monitor pode estar associado ao controlador de ajuste, permitindo a verificação de violações no contrato de serviço negociado diretamente no nível primitivo.

O método *getStatistics()*, na classe *Monitor*, calcula as estatísticas de medição da real QoS fornecida, com apoio do método *calcStats()* na classe *MonitoringStrategy*. Isso permite uma maior flexibilidade de estratégias de monitoramento em relação ao modo pelo qual os parâmetros de caracterização de serviços, usados nas medições, são estruturados e distribuídos pelo ambiente.

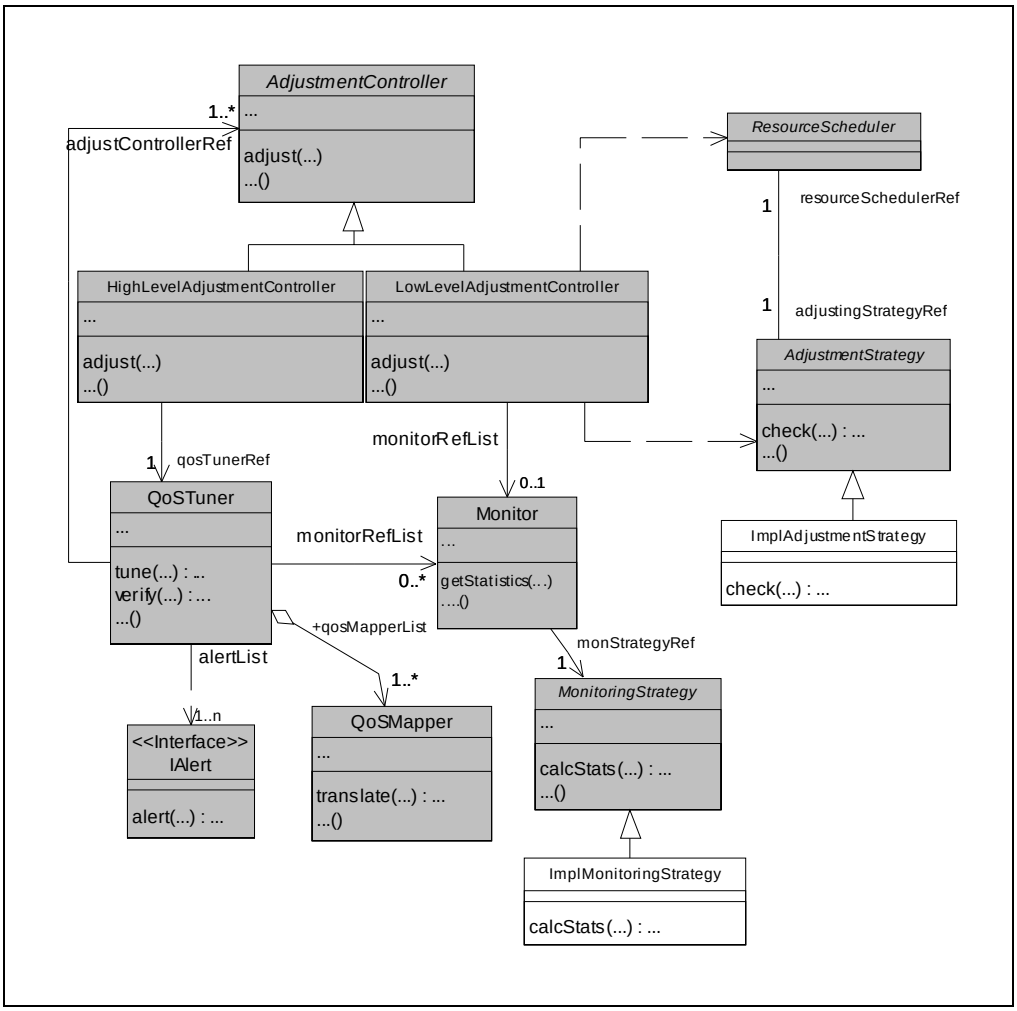

Figura 2.4 – Framework de Sintonização de QoS

A classe *QoSTuner* representa o conceito de sintonizador de QoS. O método *alert()* simboliza um alerta de violação do nível de QoS inicialmente negociado e é disparado como alternativa no caso de um sintonizador de QoS não puder mais

manter o nível contratado. Ele é definido por uma interface *IAlert* disponibilizada pela classe *QoSTuner*. A classe que implementar tal interface atua como um agente que executa certas ações em caso de violação de QoS.

O método *tune()* é acionado pelo *verify()* sempre que a avaliação dos cálculos estatísticos sobre o fluxo indicar uma (iminência de) violação de contrato de QoS. Ele é responsável pelos mecanismos de sintonização explicados anteriormente. Como parte desses mecanismos, o método *tune()* pode: i) acionar o método *tune()* em outros sintonizadores naquele mesmo subsistema; ii) acionar o método *adjust()* de controladores de ajuste em subsistemas internos; ou iii) invocar um alerta de violação de contrato de serviço, por meio do método *alert()*.

2.4.3. Levantamento de ADLs para a descrição arquitetural dos Frameworks de Provisão de QoS

A descrição da semântica de negociação e sintonização usando UML não é simples, nem formal (portanto não é livre de ambigüidades). Como exemplo, o aninhamento de subsistemas não é naturalmente entendido nos diagramas de classes da Figura 2.3 e da Figura 2.4 (nem em outros diagramas).

O conjunto das deficiências estruturais e estilísticas de UML (apresentadas na Seção 2.1) conduz à necessidade de escolha de uma ADL clássica para a descrição arquitetural dos Frameworks de Provisão de QoS. Para isso, diversas ADLs foram avaliadas: ACME, Aesop, C2, Darwin, MetaH, Rapide, SADL, Unicon e Wright (Clements, 1996). Duas outras ADLs foram incluídas no levantamento, por oferecerem suporte à QoS: Xelha e CBabel (maiores detalhes no Apêndice B). A intenção foi escolher a que melhor atende aos interesses dessa descrição arquitetural específica, desprezando um contexto mais geral. Torna-se importante ressaltar que não se teve como objetivo levantar motivos que impedissem tal descrição em cada ADL, mas apenas razões para evitar a escolha de uma ADL em função da melhor expressividade das demais.

ACME tem o seu uso voltado para a permutação de especificações arquiteturais entre ADLs, não sendo, por vezes, classificada como tal. A utilidade do mapeamento entre ADLs está justamente em permitir um maior aproveitamento de ferramentas específicas de suporte. Como conseqüência, a

utilização de ACME para a descrição arquitetural dos Frameworks de Provisão de QoS foi obviamente descartada.

C2 apresenta um modelo semântico primitivo, tanto para os componentes, quanto para os conectores (via filtros de mensagem). A interface dos componentes em C2 é definida por uma única porta, enquanto os elementos individuais da interface são *messages*. Esse tipo de convenção torna o entendimento muito menos natural, apesar da modelagem ser viável. Notou-se, por exemplo, a falta de clareza numa associação através de uma única porta entre Negociadores de QoS e, ao mesmo tempo, entre Negociadores de QoS e Controladores de Admissão.

Darwin, MetaH e Rapide não modelam conectores como entidades de primeira ordem. Seus conectores são sempre especificados como instâncias, não podendo ser manipulados em tempo de projeto ou reusados. Essa restrição não é bem-vinda e, por si só, colocou tais ADLs num patamar de exclusão. SADL e UniCon permitem apenas conectores de tipos pré-especificados. Atualmente UniCon suporta Pipe, FileIO, ProcedureCall, DataAcess, PLBundler, RemoteProcCall e RTScheduler. SADL permite um único tipo de conector para cada um de seus estilos. É interessante que os conectores possam ser livremente tipados para oferecer maior possibilidade de reuso e flexibilidade.

Algumas ADLs analisadas oferecem suporte próprio à Qualidade de Serviço, como Xelha (Duran-Limon & Blair, 2000) e CBabel (Lobosco, 1999) (Sztajnberg & Loques, 2000), essa última ainda em estágio de desenvolvimento. Todas estão, porém, restritas a um ambiente e categorias de serviço específicas. Xelha, por exemplo, faz parte do ambiente OpenORB e limita-se a um certo número e tipos de parâmetros de QoS. O enfoque para o suporte à QoS, adotado neste trabalho, será o mesmo de (Gomes, 1999), que abrange um conjunto amplo e flexível de categorias de serviço, o que cria um empecilho para a utilização dessas linguagens desde a parametrização.

O uso de ADLs com suporte à QoS pode se tornar a escolha natural no desenvolvimento de um serviço com QoS, pois facilitam a gerência de recursos, mas não em um meta serviço de provisão de QoS, que descreve a orquestração e negociação que comandarão essa gerência de recursos. Como ilustração, o uso de

Xelha para a descrição dos Frameworks de Provisão de QoS exigiria que a plataforma OpenORB fosse reimplementada para que, por exemplo, os mecanismos de admissão, negociação, monitoração e adaptação pudessem ser parametrizáveis e expressos de alguma forma. O custo dessa alteração é muito alto.

Aesop não apresenta qualquer mecanismo da linguagem para a especificação semântica dos componentes, apesar de permitir o uso de linguagens próprias de cada estilo definido. A semântica para conectores é diferente da de componentes, podendo opcionalmente empregar Wright em sua especificação.

Em (Allen, 1997), a ADL Wright é usada para demonstrar que "uma Linguagem de Descrição de Arquitetura baseada em um modelo formal e abstrato de comportamento do sistema pode prover meios práticos de descrever e analisar arquiteturas de software e estilos arquiteturais". A escolha de Wright tornou-se mais natural, já que esse é exatamente o principal interesse deste trabalho: usar os conceitos de componentes, conectores e configurações arquiteturais, associados a uma notação formal (CSP, no caso de Wright), para descrever a provisão de QoS. Além disso, Wright apresenta ferramentas de suporte focalizadas na análise, o que inclui a verificação de *deadlocks* em conectores individuais e conformidade de tipo entre *portas* e *papéis* (conforme definições adiante).

Enfim, Wright é a ADL escolhida para a descrição arquitetural dos Frameworks de Provisão de QoS descritos em (Gomes, 1999). O Capítulo 3, a seguir, define os estilos desenvolvidos em Wright para servirem como base para a descrição arquitetural. Tais estilos definem o comportamento dos principais componentes dos meta serviços de negociação e sintonização de QoS e as regras de interação entre eles.