Um Metamodelo Para Aplicações Multimídia Distribuídas Autoadaptativas[1]

Felipe A. P. Pinto[2], Adilson Barboza Lopes[3], Daniel Cunha da Silva[4], André G. P. Silva[5]

Universidade Federal do Rio Grande do Norte (UFRN)

Departamento de Informática e Matemática Aplicada (DIMAp)

(+55 84) 3215-3813 - 59072-970 - Natal - RN

[2]felipe_app@yahoo.com.br, [3]adilson@dimap.ufrn.br, [4]danielcsilva1@yahoo.com.br, [5]andregustavo@uern.br

RESUMO

Sistemas multimídia distribuídos possuem requisitos bastante variáveis e que podem mudar dinamicamente. Estes sistemas devem prover suporte para adaptações dinâmicas, permitindo ajustar suas estruturas e comportamentos. Este trabalho propõe um metamodelo para a descrição de aplicações multimídia distribuídas autoadaptativas, que permite descrever modelos em tempo de execução usados para definir novas estruturas consistentes com os requisitos e novos elementos do sistema, mantendo uma autorrepresentação deste. O metamodelo é capaz de representar os componentes do sistema e seus relacionamentos, políticas para qualidade de serviço e ações de adaptação. Adicionalmente, este trabalho apresenta uma arquitetura para a implementação da proposta e uma ADL definida para modelar um estudo de caso.

ABSTRACT

Distributed multimedia systems have a high-variability of requirements and the most of them can be dynamically changed. So, these systems should provide support for dynamic adaptations in order to adjust their structures and behaviors at runtime. This paper proposes a meta-model for description of self-adaptive distributed multimedia applications, which allows describing models at runtime used to define new structures consistent with the requirements and other involved elements of the system, keeping a system self-representation. The meta-model can represent the system components and their relationships, policies for quality of service and adaptation actions. Further, this paper presents an architecture for the proposal implementation and an ADL defined to model a case study.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design – *Distributed systems, Hierarchical design, Real-time systems and embedded systems.*

General Terms

Management, Measurement, Performance, Design.

Keywords

Distributed multimedia systems, self-adaptive, model, meta-model, adaptation, ADL, architecture.

1. INTRODUÇÃO

Sistemas multimídia distribuídos possuem características que são bastante variáveis, podendo implicar em novos requisitos à medida que novas tecnologias são disponibilizadas ou na necessidade de adequação de acordo com a quantidade de recursos disponíveis. Dessa forma, deve-se permitir a possibilidade dessas aplicações realizarem ajustes e adaptações dinâmicas que afetam sua estrutura e comportamento.

Por adaptação dinâmica entendemos a capacidade de um sistema modificar sua estrutura e seu comportamento dinamicamente, em resposta a mudanças em seu ambiente de execução [1]. O uso de adaptação dinâmica permite melhorar a disponibilidade de recursos em sistemas críticos como, por exemplo, uma videoconferência para um procedimento cirúrgico, ou simplesmente tornar a interação do usuário com o sistema mais agradável, como o uso de um *player* em uma aplicação doméstica de entretenimento.

As três tecnologias chaves para o desenvolvimento de aplicações com suporte à adaptação dinâmica são [2]: a separação de interesses, a reflexão computacional e o desenvolvimento baseado em componentes. O uso conjunto de reflexão computacional e desenvolvimento baseado em componentes tem sido adotado por várias abordagens [3][4][5][6], sendo também o ponto de partida deste trabalho.

Objetivando prover capacidades de autoadaptação em sistemas multimídia distribuídos, este trabalho segue uma abordagem para adaptação baseada em modelos. Modelos proporcionam facilidades para representar e especificar elementos de um sistema de *software*. É comum classificar modelos como concretos ou abstratos. Os modelos abstratos são chamados de metamodelos e especificam regras que determinam como devem ser os modelos concretos. Assim, podemos dizer que um metamodelo pode ser instanciado em vários modelos concretos. Neste trabalho, usaremos a terminologia metamodelo e modelo, respectivamente, para indicar um modelo abstrato e um modelo concreto. Uma discussão mais detalhada sobre essa terminologia é apresentada em [7].

[1] A Meta-model for Self-adaptive Distributed Multimedia Applications

Nesse contexto, este trabalho propõe um metamodelo para a descrição de aplicações multimídia distribuídas autoadaptativas baseadas em componentes. Esse metamodelo pode ser usado para descrever modelos que funcionam como a autorrepresentação dos sistemas em tempo de execução. Em outras palavras, ele pode ser entendido como uma especificação para o metanível reflexivo desses sistemas, sendo capaz de representar componentes e seus relacionamentos, além de políticas para especificação de QoS (*Quality of Service*) e ações de adaptação.

Além do metamodelo proposto o trabalho apresenta uma ADL (*Architecture Description Language*) baseada em XML (*Extensible Markup Language*) que permite escrever uma descrição textual do sistema que pode ser mapeada por um *parser* para um modelo, em tempo de execução, conforme o metamodelo definido. Tanto o metamodelo quanto a ADL são independentes de aplicação específica. A ADL foi criada para apoiar o uso do metamodelo, mas este é independente dela, podendo ser usada outra ADL, desde que seja fornecido um *parser* que faça o mapeamento corretamente. O trabalho ainda propõe uma arquitetura de implementação objetivando mostrar como fazer uso da abordagem apresentada.

Assim, destacamos o metamodelo proposto como a principal contribuição deste trabalho. A definição da ADL e a arquitetura são contribuições secundárias que têm como objetivo apoiar o uso do metamodelo, dando suporte para abordagens de adaptação baseada em modelos. Este trabalho é reflexo de experiências anteriores que tiveram a adaptação dinâmica como foco [8][9], porém sem ter uma preocupação efetiva com reuso e especificação do metanível, o que pode ser alcançado através do uso de modelos.

As seções que seguem estão organizadas da seguinte maneira: a seção 2 discute aspectos relacionados com adaptação baseada em modelos; a seção 3 descreve alguns trabalhos relacionados; a seção 4 apresenta o metamodelo proposto; uma arquitetura de componentes para a proposta é definida na seção 5; a seção 6 apresenta uma ADL e a modelagem de um estudo de caso de acordo com a abordagem apresentada; e, finalmente, a seção 7 relata algumas considerações finais e perspectivas para trabalhos futuros.

2. ADAPTAÇÃO BASEADA EM MODELOS

Vários mecanismos usados para adaptação são incorporados no escopo da aplicação e ligados ao seu código. Nessa abordagem possíveis falhas no sistema podem ser capturadas imediatamente quando ocorrem, por exemplo, através do tratamento de exceções fornecido pelas linguagens de programação modernas. Porém, a falha é tratada de maneira localizada e o sistema pode não ser capaz de determinar a origem real do problema, além de não ter como monitorar suaves anomalias como a degradação do desempenho. Essa solução torna difícil realizar reconfigurações nas políticas de adaptação, já que geralmente elas estão entrelaçadas no código do sistema [10].

O trabalho apresentado em [10] defende uma abordagem alternativa com base no uso de modelos, de forma a "externalizar" a adaptação. O uso de modelos apresenta algumas vantagens: primeiro, do ponto de vista da aplicação, esta abordagem permite escolher qualquer modelo disponível que melhor possa representá-la; segundo, devido o nível de independência entre modelo e aplicação existe um suporte maior para reuso; terceiro, os modelos podem ser facilmente modificados; e, por último, é possível explorá-los e analisá-los, em tempo de execução, buscando melhorar requisitos ligados ao desempenho, confiabilidade e segurança, por exemplo. Uma desvantagem de trabalhar com modelos é a manutenção de um elemento extra, que é o metamodelo. Porém, essa abordagem torna possível realizar transformações automatizadas e gerar parte da implementação do sistema, ou ainda, usar o metamodelo como base para a definição de DSMLs (*Domain-Specific Modeling Languages*) [11].

A Figura 1 apresenta um esquema para adaptação baseada em modelos, inspirada na abordagem apresentada em [10] e adaptada pelo presente trabalho para incluir o metamodelo e a ADL propostos. Note que o modelo e as entidades que realizam a adaptação são externos ao sistema, por isso o uso do termo adaptação "externalizada".

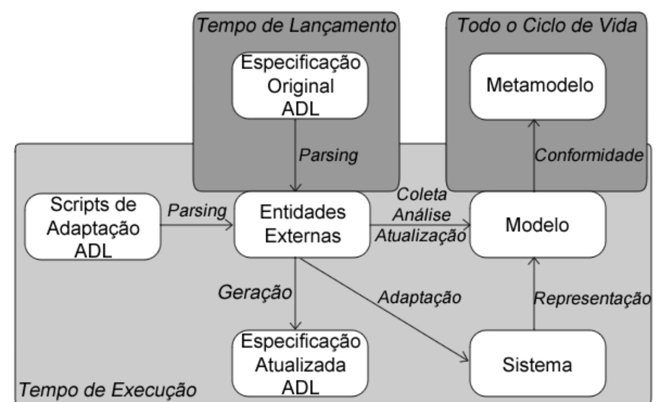

Figura 1 - Esquema para adaptação baseada em modelos.

Nesse esquema, o sistema é representado por um modelo que funciona como sua autorrepresentação (metanível), sendo o mesmo definido em conformidade com um metamodelo. Podem ser definidas entidades externas como, por exemplo, gerentes, monitores e *parsers*, que coletam e analisam informações do modelo e realizam alterações nele quando necessário. Uma entidade externa é qualquer elemento, exceto o metamodelo, externo ao sistema e independente dele. Por outro lado, deve-se assegurar que as mudanças realizadas no modelo sejam refletidas no sistema, ou seja, devem provocar adaptações.

Uma ADL pode ser usada para a especificação do sistema, de forma que no lançamento deste, um *parser* possa extrair informações dessa descrição e passá-las às entidades externas, que por sua vez, mapeiam essas informações para as estruturas do modelo. Durante a execução do sistema, *scripts* com ações de adaptação podem ser submetidos às entidades externas permitindo que elas executem as alterações necessárias no modelo e efetivem a adaptação no sistema.

Por fim, nos sistemas autoadaptativos é provável que, ao longo do tempo, devido adaptações sucessivas, a especificação original da arquitetura não mais reflita o estado atual do sistema. Nesse caso seria interessante realizar uma reconstrução dessa especificação arquitetural [12]. Nesse sentido, as entidades externas podem coletar informações do modelo com o objetivo de gerar uma especificação atualizada usando a ADL em questão.

Observando a Figura 1, deve-se destacar que o foco deste trabalho é definir um metamodelo para sistemas multimídia distribuídos autoadaptativos. A ADL e a arquitetura são introduzidas com o objetivo de possibilitar o uso da abordagem proposta. Assim, a forma como as entidades externas operam internamente não está no escopo deste trabalho.

3. TRABALHOS RELACIONADOS

Vários trabalhos têm abordado o tema de adaptação dinâmica como, por exemplo, o OpenCOM [3], Fractal [4], OpenORB [3], TOAST [5] e PLASMA [6]. Dentre os citados, apenas o TOAST e o PLASMA discutem a questão da adaptação no domínio dos sistemas multimídia, enquanto os demais tratam adaptação em sistemas distribuídos de uma forma geral. Todas essas abordagens usam os conceitos de computação reflexiva, porém nenhuma delas propõe metamodelos capazes de descrever os modelos manipulados em tempo de execução no metanível. O uso de um metamodelo traz várias vantagens, como as que foram citadas na seção 2 deste trabalho.

O trabalho proposto em [13] descreve um metamodelo para especificação de requisitos de QoS e sua realização na plataforma de componentes CORBA (*Common Object Request Broker Architecture*) [14]. Para isso o trabalho estendeu o CCM (*CORBA Component Model*) [15] seguindo dois caminhos: o primeiro permitiu ao CCM suportar os conceitos definidos no metamodelo proposto para especificação de QoS e o segundo permitiu incluir a possibilidade de modelar interações entre componentes através de portas de *stream* para o caso de troca de fluxos multimídia. A segunda extensão definiu um novo metamodelo, para ser integrado ao CCM, que modela esse novo tipo de interação. Uma limitação da abordagem é que o metamodelo de QoS não integra estratégias de adaptação, ou seja, o que fazer quando o contrato de QoS estabelecido for quebrado.

Em [16] é proposto um *framework* para QDD (*Quality-Driven Delivery*) em ambientes multimídia distribuídos. QDD refere-se à capacidade do sistema entregar documentos, ou objetos, considerando as expectativas do usuário em termos de requisitos não funcionais. O *framework* segue uma abordagem dirigida por modelos com foco na modelagem de informações relacionadas à qualidade de serviço do sistema, tendo suporte para a representação de decisões de QoS. Decisões de QoS são ações que podem ser realizadas em tempo de execução na tentativa de melhorar a qualidade de serviço. A principal diferença entre o metamodelo proposto por esse *framework* e a proposta do presente trabalho, é que este apresenta uma abordagem de metamodelo mais completa, fornecendo suporte para a representação não apenas de informações relacionadas à qualidade de serviço, mas também para os componentes do sistema e seus relacionamentos, bem como para ações de adaptação. Estas seriam semelhantes à ideia de decisões de QoS.

Como podemos ver, a maior parte das abordagens que seguem uma linha de proposta semelhante ao presente trabalho estão, geralmente, focadas em pontos específicos, sendo o mais comum a modelagem de informações relativas à QoS do sistema. Ainda há a necessidade de uma proposta de metamodelo mais completo, de forma a dar suporte ao esquema de adaptação "externalizada" apresentado anteriormente.

4. O METAMODELO PROPOSTO

Nesta seção apresentamos o metamodelo proposto, que pode ser visto no diagrama *Ecore* [17] da Figura 2. Podemos destacar duas partes principais: a primeira trata da especificação estrutural do sistema; a segunda representa informações relacionadas à qualidade do serviço e ações de adaptação.

4.1 Especificação Estrutural

Inicialmente descreveremos o metamodelo considerando o suporte à representação estrutural do sistema, isto é, os seus componentes e os relacionamentos entre eles. Essa parte do metamodelo é ilustrada na Figura 2 pelos elementos preenchidos com cor branca.

O elemento *Application* pode ser considerado como a raiz do diagrama. Uma aplicação (*Application*) possui apenas um componente (*Component*), mas este pode conter muitos outros subcomponentes. O relacionamento *parent* de *Component* indica que todo componente é um subcomponente de outro, sendo a única exceção o componente raiz da aplicação.

Os componentes possuem um conjunto de interfaces (*AbstractInterface*) que são especializadas para interfaces comuns (*Interface*), que expressam serviços providos e requeridos, e interfaces de *stream* (*InterfaceStream*), que expressam pontos de entrada e saída para o fluxo multimídia. Dessa forma, um componente possui *bindings* e *connections*, respectivamente, para representar as ligações entre interfaces de serviços e interfaces de *stream*.

Os atributos chamados *name* dos elementos devem identificar as entidades de maneira unívoca. Em *Application*, além de um nome, pode-se adicionar uma descrição (*description*). O atributo *location* de *Component* indica a localização do componente na rede, enquanto *className* referência a implementação desse componente, por exemplo, se é usada uma linguagem orientada a objetos, seria o nome da classe que deve ser instanciada.

Uma interface de serviço (*Interface*) possui uma *role* que pode ser *Server* ou *Client* indicando, respectivamente, provimento ou requerimento de serviço. O atributo *optional* indica se a interface de serviço é obrigatória. No caso de uma interface provida ser obrigatória significa que o componente é obrigado a implementar a interface, enquanto que uma interface requerida obrigatória significa que o componente não irá funcionar se a interface não for fornecida. Uma interface de *stream* (*InterfaceStream*) também possui uma *role* que pode ser *Input* ou *Output*, indicando a entrada ou saída de fluxo multimídia, respectivamente.

Em *AbstractInterface*, o atributo *collection* indica se o componente possui apenas uma interface daquele tipo ou uma coleção de interfaces. Em *Interface*, *signature* representa o nome da interface na linguagem de programação usada para implementar o sistema, por exemplo, *java.lang.Runnable*.

Uma conexão (*Connection*) entre interfaces de *stream Output* e *Input* possui atributos relacionados ao *streaming* multimídia, como o protocolo (*protocol*) e porta (*port*) usados na comunicação, o tamanho do *buffer* (*bufferSize*) no receptor e o atraso mínimo (*minDelay*) para o envio de pacotes sucessivos pelo transmissor.

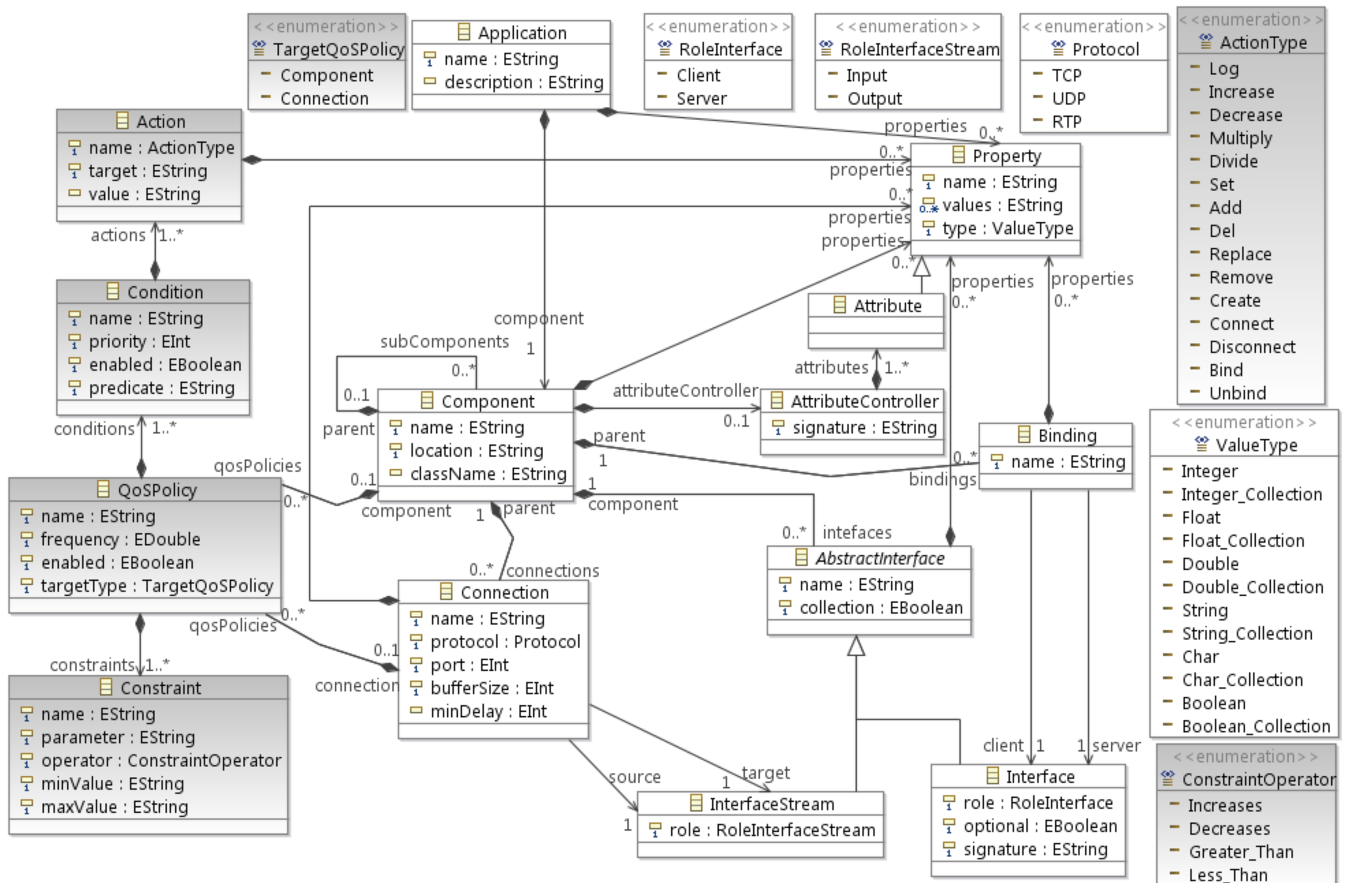

Figura 2 - Diagrama *Ecore* do metamodelo proposto. Os elementos com cor cinza modelam as informações relacionadas à especificação de QoS e ações de adaptação, enquanto os demais modelam informações estruturais do sistema.

AbstractInterface, *Application*, *Component*, *Connection*, *Binding* e *Action* (este será explicado na próxima subseção) todos possuem propriedades (*Property*). Uma propriedade possui um nome (*name*), uma lista de valores (*values*) e um tipo (*type*) para esses valores. O tipo determina se o atributo *values* representa uma coleção ou um valor atômico. Por exemplo, se *type* é *String*, então *values* deve ser tratado como um valor atômico do tipo *String*. Por outro lado, se *type* é *Integer_Collection*, então *values* deve ser tradado como uma coleção de inteiros. As coleções representadas por *values* são homogêneas em relação ao tipo.

Além das propriedades, os componentes podem definir atributos (*Attribute*) no contexto de uma interface controladora de atributos (*AttributeController*). Uma interface controladora de atributos encapsula um conjunto de atributos definidos para um dado componente e indica o nome da interface (*signature*) usada para acessar esses atributos. Um atributo herda todas as características de uma propriedade, porém é definido somente no contexto de uma interface controladora. Por exemplo, um componente codificador pode ter um atributo chamado *quality* que representa a qualidade do vídeo codificado. Esse componente permite acesso para esse atributo através de uma interface chamada *br.encoder.IEncoder* que possui métodos *get* e *set* para *quality*. Para esse exemplo, o componente codificador deve definir uma interface controladora com valor de *signature* sendo *br.encoder.IEncoder* e o atributo *quality* deve ser definido no contexto desta interface.

O conceito de atributos para componentes é semelhante ao conceito de atributos para classes. Já o termo propriedade é usado aqui para representar características que não são conhecidas na fase de projeto do componente, sendo úteis para descrever características dinâmicas de plataformas específicas. Propriedades podem ser criadas no momento em que o elemento é instanciado ou durante a execução do sistema, adicionando novas características ao elemento em questão. Os atributos são os mesmos durante todo o ciclo de vida do sistema, apenas seus valores são alterados, enquanto as propriedades podem ser criadas e removidas em tempo de execução. Resumidamente, atributos são membros estáticos dos componentes definidos no contexto de uma interface controladora de atributos, enquanto propriedades são membros dinâmicos. Um conceito semelhante de propriedades é também usado no PREMO (*Presentation Environment for Multimedia Objects*) [18].

4.2 Especificação de QoS e Autoadaptação

Nesta subseção discutiremos os elementos do metamodelo relacionados com a QoS do sistema e com as ações de adaptação. Os elementos correspondentes do metamodelo estão preenchidos com cor cinza na Figura 2.

Conexões (*Connection*) e componentes (*Component*) podem ter várias políticas de QoS associadas (*QoSPolicy*), porém apenas uma delas pode estar habilitada por vez para um dado elemento. Uma política de QoS possui um atributo nome (*name*) que deve ser único entre as políticas definidas, um atributo que indica a frequência (*frequency*) de verificação das restrições (*Constraint*) e

condições (*Condition*), um atributo que diz se a política está habilitada (*enabled*) e um atributo que indica o alvo da política (*targetType*) que pode ser um componente ou uma conexão.

Uma política define restrições (*Constraint*). Uma restrição possui um nome (*name*) que deve ser único entre as restrições, um parâmetro (*parameter*) para o qual a restrição é aplicada, um valor mínimo (*minValue*), um valor máximo (*maxValue*) e um operador (*operator*). O operador é definido por *ConstraintOperator* e indica se o parâmetro diminui (*Descreases*), aumenta (*Increases*), é maior que (*Greater_Than*), ou é menor que (*Less_Than*) o valor informado. Dessa forma é possível indicar faixas de valores dizendo, por exemplo, que a taxa de perda de pacotes é entre 5% e 10% (*minValue=5 e maxValue=10*). Também é possível indicar valores atômicos, usando a convenção de que o valor mínimo e o valor máximo são iguais, por exemplo, se a taxa de perda de pacotes aumenta 5% (*minValue=maxValue=5* e *operator=Increases*). Quando se trabalha com faixas de valores o operador não precisa ser indicado.

Restrições estabelecem regras testadas através de condições (*Condition*) definidas pela política. Condições possuem um atributo nome (*name*) que as identificam univocamente, além de um atributo prioridade (*priority*) que diz qual condição considerar no caso de mais de uma delas ser verdadeira, já que a política pode definir várias condições. Uma condição pode ser habilitada através do atributo *enabled*. No caso de estar desabilitada ela deverá ser ignorada mesmo sendo verdadeira. O predicado (*predicate*) da condição representa uma expressão *booleana* envolvendo as restrições. Por exemplo, se uma restrição, chamada *R1*, diz "a taxa de perda de pacotes está entre 5% e 10%" e outra restrição, chamada *R2*, diz "*jitter* aumenta 5ms", então um predicado para uma condição *C1* poderia ser *R1* ^ *R2*, sendo "^" o *AND* lógico.

As condições possuem ações de adaptação (*Action*). Quando uma determinada condição for verdadeira as ações de adaptação associadas devem ser executadas. A ação possui um alvo (*target*) e um valor (*value*). A semântica desses atributos depende do nome (*name*) da ação, definida por *ActionType*, que indica o tipo de adaptação que será executada, podendo ser:

- Operação de *log* (*Log*). Nesse caso o *target* indica o nome de um arquivo para *log*. O atributo *value* não é usado;
- Operações sobre atributos e propriedades dos elementos (*Increase*, *Decrease*, *Multiply*, *Divide, Set, Add, Del*). Os quatro primeiros são as operações aritméticas, por exemplo, poderíamos ter "*name=Increase, target=Component:root.Component:enc.Attribute:quality, value=5*". Isso diz para aumentar em cinco, o valor do atributo *quality* do componente chamado *enc*, que é um subcomponente do componente chamado *root*. *Set* altera o valor de um atributo ou propriedade, enquanto *Add* e *Del* são usados para adicionar e remover propriedades. No caso da ação *Add*, o *target* indica o elemento que terá a propriedade e *value* indica o valor da propriedade. Para a ação *Del*, *target* indica a propriedade que será removida e *value* não é usado nesse caso;
- Operações sobre os componentes (*Replace*, *Remove*, *Create*). *Replace* diz que um componente deve ser substituído por outro; nesse caso o *target* indica o componente que será trocado e *value* indica o componente substituto. *Remove* retira um componente que não é mais usado do ambiente de execução do sistema; nesse caso, apenas *target* é usado e indica o componente que deve ser removido. *Create* cria um componente como um subcomponente de outro qualquer; nesse caso, o *target* indica o componente que irá encapsular o novo componente criado e *value* indica uma referência para a definição do componente que será instanciado;
- Operações sobre *bindings* e conexões (*Connect*, *Disconnect*, *Bind*, *Unbind*). No caso das ações *Disconnect* e *Unbind* o atributo *value* não é usado, mas *target* indica a conexão ou o *binding* que será desligado. Para as ações *Bind* e *Connect*, o *target* indica a interface de *role Client* ou *Output* e o *value* indica a interface de *role Server* ou *Input*.

De acordo com a descrição do elemento *Action* (Figura 2), ações como *Add*, *Connect* e *Create* não teriam todas as suas informações expressas. Por exemplo, uma conexão possui vários atributos que devem ser configurados como o protocolo e a porta. O mesmo acontece para um componente que será instanciado e precisa informar um nome e sua localização, ou para uma nova propriedade que precisa definir seu nome e tipo. A solução consiste em permitir que ações de adaptação definam propriedades.

Assim, em uma ação de conexão teríamos "*name=Connect, target=Component:root.Component:prod.InterfaceStream:out, value=Component:root.Component:cons.InterfaceStream:in*". Essa ação diz para conectar a interface de *role Output*, chamada *out*, do componente *prod*, com a interface de *role Input*, chamada *in*, do componente *cons*. Essa ação deve definir propriedades para os atributos da conexão, como porta, protocolo, tamanho do *buffer*, etc. Assim teríamos propriedades do tipo: "*name=bufferSize, values=1024, type=Integer*"; "*name=protocol, values=RTP, type=String*" e assim por diante.

5. ARQUITETURA PROPOSTA

Esta seção apresenta uma arquitetura que permita fazer uso da abordagem proposta, a qual pode ser vista na Figura 3.

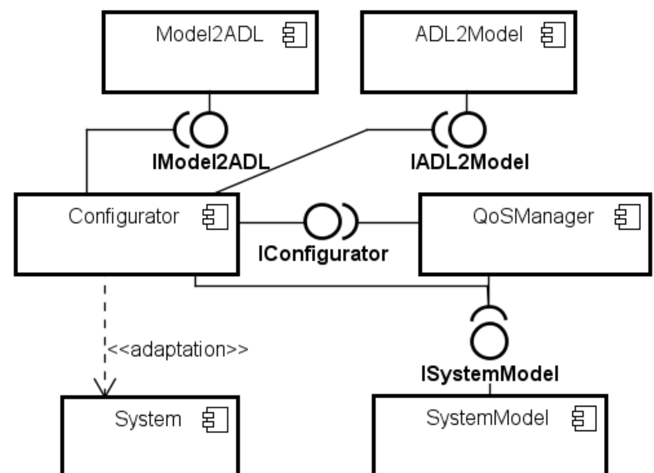

Figura 3 - Arquitetura de implementação (UML 2.1).

Os componentes *System* e *SystemModel* representam, respectivamente, o sistema e as estruturas definidas pelo

metamodelo. O *SystemModel* implementa as estruturas definidas pelo metamodelo e fornece uma interface (*ISystemModel*) para acesso aos metaelementos. Um gerente de QoS (*QoSManager*) é responsável por monitorar as restrições e condições definidas na especificação da aplicação para a qualidade de serviço. Caso o gerente de QoS perceba que alguma ação de adaptação é necessária ele notifica o *Configurator*, via interface *IConfigurator*, que por sua vez atualiza os metaelementos necessários, via *ISystemModel*, e executa a adaptação no sistema base (*System*).

O *Configurator* faz uso dos componentes *ADL2Model* e *Model2ADL*. O primeiro mapeia definições de uma ADL para as estruturas definidas pelo metamodelo e o segundo faz o inverso, gerando uma especificação atualizada, em alguma ADL, com base nas informações do modelo.

Toda ação de adaptação é encaminhada ao *Configurator* pela interface *IConfigurator*. Esta interface deve ser disponibilizada ao ambiente externo de forma a permitir solicitações de adaptação, inclusive remotamente. A qualquer momento deve ser possível solicitar ao *Configurator* para gerar uma especificação atualizada do sistema, já que a especificação original talvez não represente a realidade atual, devido à execução de adaptações.

A arquitetura é inspirada no esquema de adaptação baseada em modelos apresentado na Figura 1 e não é dependente do metamodelo proposto. O componente *SystemModel* pode implementar qualquer outro metamodelo definido.

6. MODELANDO UM ESTUDO DE CASO

Esta seção apresenta como estudo de caso a modelagem de uma aplicação para *streaming* de vídeo, de acordo com a abordagem apresentada por este trabalho. Para permitir essa modelagem, ao longo da seção serão apresentadas as principais partes de uma ADL baseada em XML definida considerando o metamodelo descrito e proposto pelo trabalho. Essa modelagem, especificamente, considera o uso da linguagem Java para implementação desse estudo de caso.

A maior parte do mapeamento entre metamodelo e ADL é direto, de modo que, apenas passos menos intuitivos necessitam de uma explicação mais detalhada. A aplicação modelada nesse estudo de caso é formada por um componente principal composto de dois outros subcomponentes, sendo um codificador, responsável por codificar e enviar o fluxo de vídeo, e um decodificador, que recebe o fluxo e decodifica-o.

Na ADL proposta, os componentes da aplicação podem ser definidos no arquivo da aplicação ou em arquivos separados. O atributo *definition* na *tag Component* é usado para referenciar definições externas de componentes, como mostrado na declaração do componente chamado *encoder* na Figura 4. Nesta mesma figura também é mostrado um componente chamado *root* que contém dois outros componentes chamados *encoder* e *decoder*, localizados em máquinas com endereços IP diferentes (*location*). O componente *encoder* segue a definição especificada em *adl.EncoderADL* que é mostrada na Figura 5, enquanto o componente *decoder* define diretamente sua própria especificação. Dessa forma, *encoder* possui uma interface de *stream* com *role Output* chamada *out*, enquanto *decoder* possui uma interface de *stream* com *role Input* chamada *in*.

```
<Application name="example"
    description="Application example 1">
<Component name="root">
    <Component name="encoder" location="192.168.0.2"
        definition="adl.EncoderADL" />
    <Component name="decoder" location="192.168.0.3"
        class="example.DecoderImpl">
        <AttributeController
            signature="example.IDencoderAttControler">
            <Attribute name="media-type" value="h264"
                type="String" />
        </AttributeController>
        <InterfaceStream name="in" role="Input"
            collection="false" />
    </Component>
    <Connection name="conn" source="encoder.out"
        target="decoder.in" protocol="RTP" port="8888"
        bufferSize="4096" minDelay="0" />
</Component>
<QoSPolicy name="qos-policy"
    targetType="Connection" target="root.conn"
    frequency="3000" enabled="true">
    <Constraint name="c1" parameter="packet-loss-rate"
        operator="Increases" value="2" />
    <Constraint name="c2" parameter="rtt"
        operator="Greater_Than" value="100" />
    <Condition name="condition-1" priority="1"
        enabled="true" predicate="c1 | c2">
        <Action name="Decrease" target="Component:root.
            Component:encoder.Attribute:quality"
            value="2" />
    </Condition>
    <Condition name="condition-2" priority="2"
        enabled="true" predicate="!c1 ^ !c2">
        <Action name="Increase" target="Component:root.
            Component:encoder.Attribute:quality"
            value="2" />
    </Condition>
</QoSPolicy>
</Application>
```

Figura 4 - Uma aplicação definida com a ADL proposta.

```
<ComponentDefinition name="adl.EncoderADL"
    class="example.EncoderImpl">
    <AttributeController
        signature="example.IEncoderAttControler">
        <Attribute name="media-type"
            value="h264" type="String" />
        <Attribute name="quality"
            value="90" type="Integer" />
    </AttributeController>
    <InterfaceStream name="out"
        role="Output" collection="false" />
</ComponentDefinition>
```

Figura 5 - Definição para *EncoderADL*.

O atributo *class* da *tag Component* indica o nome da classe (estamos considerando a linguagem Java nesse exemplo) que deve ser instanciada para fazer uso do componente, sendo mapeado para o atributo *className* de *Component* como definido no metamodelo. Caso o componente use uma definição externa através do atributo *definition*, a classe deve ser informada na definição externa por meio do atributo *class* da *tag ComponentDefinition*.

O componente *encoder* possui dois atributos, um inteiro com valor 90, chamado *quality*, que representa a porcentagem da qualidade do vídeo codificado e uma *String*, com valor *h264*, chamada *media-type*, que indica a codificação usada. Esses atributos são gerenciados através de uma interface controladora de atributos com assinatura *example.IEncoderAttController*. Note que este será o valor mapeado para o atributo *signature* de *AttributeController*, como indicado no metamodelo.

O componente *decoder* define apenas um atributo do tipo *String*, chamado *media-type*, com valor também *h264* indicando o tipo de codificação suportada. O atributo possui uma interface controladora com assinatura *example.IDecoderAttController*. Novamente, este será o valor mapeado para *signature* de *AttributeController*.

No final da definição do componente *root* (Figura 4) podemos ver a conexão entre os dois subcomponentes definidos, tendo como origem a interface *out* do *encoder* e como alvo a interface *in* do *decoder*. Adicionalmente, na *tag Connection* são configurados os demais atributos de uma conexão como o protocolo, a porta, o tamanho do *buffer*, etc.

O último passo consiste em definir as políticas de QoS. Na Figura 4, temos uma política, chamada *qos-policy*, com alvo na conexão, chamada *conn*, estabelecida anteriormente. O alvo de uma política define seu ponto de atuação e é representado pelo atributo *target* da *tag QoSPolicy*, enquanto que o atributo *targetType* indica se esse alvo é um componente ou uma conexão. A decisão de definir as políticas diretamente dentro da *tag Application* e referenciar um alvo a partir delas, ao invés de defini-las dentro da *tag Connection* ou *Component* correspondente, foi tomada visando aumentar a legibilidade da ADL.

A política do exemplo define duas restrições chamadas *c1* e *c2*. A primeira será satisfeita quando a taxa de perda de pacotes aumenta em duas unidades e a segunda quando o RTT (*Round Trip Time*) for maior que 100. Na prática, isso representa 2% e 100ms, pois a taxa de perda de pacotes é dada em porcentagem e o RTT em milissegundos.

Ainda na Figura 4, as restrições são usadas em condições (*condition-1* e *condition-2*). O predicado da primeira é um simples *OR* lógico (*c1 | c2*), enquanto a segunda diz se não acontecer *c1* e não acontecer *c2* (*!c1 ^ !c2*). Uma vez que o predicado seja verdadeiro e o atributo *enabled* indique *true*, as ações de adaptação correspondentes são executadas. A ação de *condition-1* diminui a qualidade da codificação do vídeo em duas unidades, enquanto a ação de *condition-2* aumenta a qualidade em duas unidades. Na prática isso representa diminuir ou aumentar a qualidade em 2%, pois *quality* é dado em porcentagem.

Considerando a arquitetura proposta na seção anterior, é possível submeter ao *Configurator* um conjunto de ações de adaptação. Por exemplo, poderíamos usar uma versão gratuita, ou mais barata, do decodificador, bastando ordenar a troca desse componente.

Um *script* de adaptação para a aplicação *example*, desse estudo de caso, é ilustrado na Figura 6. A primeira ação desse *script* determina a criação de um componente *(name=create)* como subcomponente do componente *root* (*target=Component:root*). Esse componente segue a especificação definida em *adl.FreeDecoderADL* (*value=adl.FreeDecoderADL*). Essa instância possui nome *free-decoder* e localização *192.168.0.3*, ambos definidos usando propriedades. A segunda ação consiste em substituir o subcomponente de *root* chamado *decoder* pelo novo componente chamado *free-decoder*.

```
<ScriptAdaptation application="example">
<Action name="Create" target="Component:root"
    value="adl.FreeDecoderADL">
    <Property name="name"
        value="free-decoder" type="String" />
    <Property name="location"
        value="192.168.0.3" type="String" />
</Action>
<Action name="Replace"
    target="Component:root.Component:decoder"
    value="Component:root.Component:free-decoder" />
</ScriptAdaptation>
```

Figura 6 - *Script* com ações para adaptação.

O *Configurator* deve encarregar-se de executar as adaptações, realizando as alterações necessárias de acordo com o tipo de ação solicitada. Por exemplo, em um *replace* ele irá desconectar todas as conexões e *bindings* do componente alvo, conectá-las ao novo componente e, finalmente, remover o componente antigo do ambiente de execução.

Note que para manter a consistência do sistema, no caso da remoção de um componente, por exemplo, todos os seus subcomponentes são também removidos, bem como as políticas de QoS definidas para ele. Assim, além de ações, o *script* também pode definir novas políticas de QoS para os novos elementos, usando exatamente a mesma sintaxe do exemplo da Figura 4.

7. CONCLUSÃO

Este trabalho apresentou um metamodelo para sistemas multimídia distribuídos autoadaptativos. A definição de um metamodelo para especificar o metanível de sistemas autoadaptativos permite que seja usada uma abordagem baseada em modelos para adaptação, possibilitando uma maior separação entre sistema e sua autorrepresentação, o que aumenta o suporte para reuso, além de tornar a exploração e análise do sistema como um todo mais simples, buscando melhorar requisitos ligados ao desempenho, confiabilidade e segurança.

O metamodelo proposto é capaz de representar os componentes do sistema e seus relacionamentos, além de políticas de QoS e ações de adaptação. Adicionalmente, foi apresentada uma arquitetura de implementação para a proposta e uma ADL definida para a modelagem de um estudo de caso.

Como trabalho futuro, pensa-se em incluir ao metamodelo suporte para especificações de políticas de seleção arquitetural. Esse tipo de processo é útil quando o sistema precisa executar uma autoadaptação que modifica sua estrutura e existe mais de uma possibilidade para a nova arquitetura. As políticas de seleção podem ser usadas por selecionadores que ajudam o sistema a decidir qual será a nova arquitetura resultante após a adaptação. Em uma abordagem desse tipo, não seria necessário, por exemplo, indicar o componente substituto em uma ação de adaptação do tipo *replace*, pois o próprio selecionador iria realizar uma busca pela melhor, ou por uma das melhores opções possíveis em um repositório de componentes.

Ainda na perspectiva de trabalhos futuros, pretende-se integrar o metamodelo definido a um *framework* que forneça suporte ao desenvolvimento de aplicações multimídia autoadaptativas, apoiando a abordagem de adaptação baseada em modelos, e incluindo facilidades para o desenvolvimento do sistema base, como uma API (*Application Programming Interface*) e um modelo de componentes com suporte à adaptação dinâmica.

8. AGRADECIMENTOS

Este trabalho foi parcialmente financiado pela CAPES através do programa DS e do projeto RH-TVD (Projeto 236/2008).

9. REFERÊNCIAS

[1] Taylor, R. N. T., Medvidovic N., Oreizy P. Architectural Styles for Runtime Software Adaptation. Proceedings of the 8th Joint Working IEEE/IFIP Conference on Software Architecture 2009 & the 3rd European Conference on Software Architecture 2009. Cambridge, England, September 2009. 171- 180.

[2] McKinley, P. K., Sadjadi, S. M., Kasten, E. P., and Cheng, B. H. 2004. Composing Adaptive Software. Computer 37, 7 (Jul. 2004), 56-64. DOI= http://dx.doi.org/10.1109/MC.2004.48.

[3] Clarke, M., Blair, G. S., Coulson, G., and Parlavantzas, N. 2001. An Efficient Component Model for the Construction of Adaptive Middleware. In Proceedings of the IFIP/ACM international Conference on Distributed Systems Platforms Heidelberg (November 12 - 16, 2001). R. Guerraoui, Ed. Lecture Notes In Computer Science, vol. 2218. Springer-Verlag, London, 160-178.

[4] Bruneton, E., Coupaye, T., and Stefani, J.-B. 2002. Recursive and Dynamic Software Composition with Sharing. 7th International Workshop on Component-Oriented Programming (WCOP02), Monday, June 10, 2002 - At ECOOP 2002, Malaga, Spain, June 10-14, 2002.

[5] Fitzpatrick, T. 1999. Open Component-Oriented Multimedia Middleware for Adaptive Distributed Applications. 1999. Tese (PhD). Computing Department, Lancaster University, Lancaster, 1999.

[6] Layaïda, O., Hagimont, D. 2005. PLASMA: A Component-based Framework for Building Self-Adaptive Applications. In Proc. SPIE/IS&T Symposium On Electronic Imaging, Conference on Embedded Multimedia Processing and Communications, San Jose, CA, USA, January 2005.

[7] Zhou, J., Rautiainen, M., and Ylianttila, M. 2008. Metamodeling for Community Coordinated Multimedia and Experience on Metamodel-Driven Content Annotation Service Prototype. In Proceedings of the 2008 IEEE Congress on Services Part II (September 23 - 26, 2008). SERVICES-2. IEEE Computer Society, Washington, DC, 88-95. DOI=http://dx.doi.org/10.1109/SERVICES-2.2008.31.

[8] Lopes, A. B. Um Framework para Configuração e Gerenciamento de Recursos e Componentes em Sistemas Multimídia Distribuídos Abertos. 2006. Tese (Doutorado). Faculdade de Engenharia Elétrica e de Computação, Universidade Estadual de Campinas, Campinas, 2006.

[9] Pinto, F. A. P., Lopes, A. B., Silva, A. G. P., and Silva, D. C. 2009. Um Modelo de Interconexão de Componentes Multimídia com Suporte à Seleção e Reconfiguração Dinâmica de Mecanismo de Comunicação. In Proceedings of the 15th Brazilian Symposium on Multimedia and the Web (Fortaleza, Brazil, 2009). vol. 1, 27-34.

[10] Garlan, D. and Schmerl, B. 2002. Model-based adaptation for self-healing systems. In Proceedings of the First Workshop on Self-Healing Systems (Charleston, South Carolina, November 18 - 19, 2002). D. Garlan, J. Kramer, and A. Wolf, Eds. WOSS '02. ACM, New York, NY, 27-32. DOI=http://doi.acm.org/10.1145/582128.582134.

[11] Paunov, S., Hill, J., Schmidt, D., Baker, S. D., and Slaby, J. M. 2006. Domain-Specific Modeling Languages for Configuring and Evaluating Enterprise DRE System Quality of Service. In Proceedings of the 13th Annual IEEE international Symposium and Workshop on Engineering of Computer Based Systems (March 27 - 30, 2006). ECBS. IEEE Computer Society, Washington, DC, 196-208. DOI=http://dx.doi.org/10.1109/ECBS.2006.39.

[12] Pollet, D., Ducasse, S., Poyet, L., Alloui, I., Cimpan, S., and Verjus, H. 2007. Towards A Process-Oriented Software Architecture Reconstruction Taxonomy. In Proceedings of the 11th European Conference on Software Maintenance and Reengineering (March 21 - 23, 2007). CSMR. IEEE Computer Society, Washington, DC, 137-148. DOI=http://dx.doi.org/10.1109/CSMR.2007.50.

[13] Ritter, T., Born, M., Unterschütz, T., and Weis, T. 2003. A QoS Metamodel and its Realization in a CORBA Component Infrastructure. In Proceedings of the 36th Annual Hawaii international Conference on System Sciences (Hicss'03) - Track 9 - Volume 9 (January 06 - 09, 2003). HICSS. IEEE Computer Society, Washington, DC, 318.1.

[14] OMG. Catalog of OMG Specifications: Middleware Specifications. Last access April 2010. Available at <http://www.omg.org/technology/documents/spec_catalog.htm#Middleware>.

[15] OMG. CORBA Component Model (CCM), V4.0. Last access April 2010. Available at <http://www.omg.org/spec/CCM/4.0>.

[16] Kerherve, B., Nguyen, K. K., Gerbe, O., and Jaumard, B. 2006. A Framework for Quality-Driven Delivery in Distributed Multimedia Systems. In Proceedings of the Advanced int'L Conference on Telecommunications and int'L Conference on internet and Web Applications and Services (February 19 - 25, 2006). AICT-ICIW. IEEE Computer Society, Washington, DC, 195.

[17] Eclipse. Ecore Tools. Last access April 2010. Available at <http://wiki.eclipse.org/Ecore_Tools>.

[18] Duke, D. J., Herman, I., Marshall, M. S. 1999. PREMO: A Framework for Multimedia Middleware: A Java description of the ISO/IEC Standard. Springer Verlag, February 1999.