Pontifícia Universidade Católica
do Rio de Janeiro

Guilherme Augusto Ferreira Lima

Eliminando Redundâncias no Perfil NCL EDTV

Dissertação de Mestrado

Dissertação apresentada ao programa de Pós–graduação em Informática da PUC–Rio como requisito parcial para obtenção do título de Mestre em Informática.

Orientador: Prof. Luiz Fernando Gomes Soares

Rio de Janeiro
Abril de 2011

Pontifícia Universidade Católica
do Rio de Janeiro

Guilherme Augusto Ferreira Lima

Eliminando Redundâncias no Perfil NCL EDTV

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-graduação em Informática da PUC–Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Luiz Fernando Gomes Soares
Orientador
Departamento de Informática — PUC–Rio

Prof. Renato Fontoura de Gusmão Cerqueira
Departamento de Informática — PUC–Rio

Prof. Marcelo Ferreira Moreno
Universidade Federal de Juiz de Fora — UFJF

Prof. José Eugenio Leal
Coordenador Setorial do Centro
Técnico Científico — PUC–Rio

Rio de Janeiro, 13 de abril de 2011

Guilherme Augusto Ferreira Lima

Graduou-se em Sistemas de Informação pela PUC–Rio em 2008. Atualmente integra o grupo de pesquisadores do Laboratório Telemídia, onde desenvolve pesquisas na área de sistemas hipermídia e televisão digital.

Ficha Catalográfica

Lima, Guilherme Augusto Ferreira

Eliminando Redundâncias no Perfil NCL EDTV / Guilherme Augusto Ferreira Lima; orientador: Luiz Fernando Gomes Soares. — Rio de Janeiro: PUC–Rio, Departamento de Informática, 2011.

v., 70 f.: il. ; 29,7cm

1. Dissertação (mestrado) — Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1. Informática — Teses. 2. NCL. 3. Nested Context Language. 4. NCL EDTV profile. 5. NCL Raw profile. 6. Eliminação de redundâncias. 7. Conversão de documentos. I. Soares, Luiz Fernando Gomes (Luiz Fernando Gomes Soares). II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CCD: 004

Aos meus pais pela confiança.
A Bianca pelo carinho e paciência.

Agradecimentos

Agradeço ao meu orientador Prof. Luiz Fernando Gomes Soares pelo exemplo, atenção aos detalhes e sabedoria; e ao Prof. Edward Hermann Haeusler pelo apoio. Agradeço também a todos os colegas do Lab. TeleMídia. Em especial, a Marcelo Moreno, Márcio Moreno, Romualdo Costa, Carlos Salles, Rafael Savignon, Francisco Sant'Anna, Carlos Batista, Roberto Azevedo, José Geraldo de Souza e Eduardo Araújo, pelas críticas e sugestões. E, ao restante da equipe, Álvaro Veiga, Bruno Lima, Felipe Nogueira, Felipe Nagato, Ricardo Rios, Luciana Redlich, Vinicius Lago pelo apoio. Finalmente, agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelo suporte financeiro sem o qual este trabalho não seria possível.

Resumo

Lima, Guilherme Augusto Ferreira; Soares, Luiz Fernando Gomes. Eliminando Redundâncias no Perfil NCL EDTV. Rio de Janeiro, 2011. 70p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A implementação de uma máquina de apresentação NCL, ou formatador, é uma tarefa complexa. Essa complexidade decorre, principalmente, da distância semântica que existe entre os documentos NCL, especificações declarativas de alto-nível, e as API que o formatador utiliza para apresentá-los, em geral imperativas e de baixo-nível. Quanto maior a distância, maior a complexidade do mapeamento e, conseqüentemente, da sua implementação que tende a ser ineficiente e não-confiável. Este trabalho apresenta um novo perfil para a linguagem NCL, chamado NCL Raw, que elimina as redundâncias do EDTV — o principal perfil da NCL 3.0 — e, de certa forma, aproxima os documentos da máquina. O perfil Raw captura apenas os conceitos essenciais do EDTV que por sua vez podem ser usados para simular a linguagem completa. Ou seja, podemos usar o Raw como uma linguagem intermediária mais simples para a qual documentos EDTV podem ser convertidos antes de serem apresentados. Esta dissertação discute as possíveis arquiteturas para conversores NCL e apresenta uma implementação de um conversor de documentos (EDTV para Raw).

Palavras–chave

NCL, Nested Context Language, NCL EDTV profile, NCL Raw profile, eliminação de redundâncias, conversão de documentos.

Abstract

Lima, Guilherme Augusto Ferreira; Soares, Luiz Fernando Gomes (Advisor). Eliminating Redundancies from NCL EDTV Profile. Rio de Janeiro, 2011. 70p. MSc. Dissertation – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The implementation of a NCL presentation engine, or formatter, is a complex task. This complexity is mainly due to the semantic distance between NCL documents, high-level declarative specifications, and the API used by the formatter to present them, in most cases low-level and imperative. The greater the distance, the greater is the complexity of this mapping and, consequently, of its implementation, which is more likely to become inefficient and bug-prone. This work presents a new NCL profile, called NCL Raw, which eliminates most of the redundancies present in EDTV — the main profile of NCL 3.0 — and, in a certain way, reduces the distance between the documents and the machine. Raw profile captures only EDTV's essential concepts, which in turn can be used to simulate the whole language defined by EDTV itself. In other words, we can use the Raw profile as a simpler intermediate language to which EDTV documents can be converted before being presented. This dissertation discusses alternative architectures for NCL converters and presents the implementation of a document converter (from EDTV to Raw).

Keywords

NCL, Nested Context Language, NCL EDTV profile, NCL Raw profile, redundancies elimination, document conversion.

Sumário

Lista de Figuras

Lista de Tabelas

Everything should be made as simple as possible, but no simpler.

— Albert Einstein

1
Introdução

1.1
Motivação

NCL (*Nested Context Language*) é uma linguagem declarativa para criação de documentos hipermídia. Um documento NCL, é uma aplicação XML que descreve o relacionamento entre objetos de mídia (textos, imagens, vídeos, etc.) no espaço e no tempo. A especificação de NCL é feita através de perfis, em que cada perfil define um conjunto de elementos e atributos suportados. NCL 3.0, a versão mais recente da linguagem, define dois perfis para televisão digital: o perfil avançado, EDTV (*Enhanced Digital Television*), e o perfil básico, BDTV (*Basic Digital Television*), subconjunto do primeiro [1, 2]. O perfil EDTV foi projetado para facilitar a autoria de aplicações de TV digital interativa. Além da sincronização de eventos, ele permite ao autor criar documentos com conteúdo adaptável, suporte à múltiplo dispositivos de exibição, efeitos de transição e animações.

A máquina de apresentação NCL, ou *formatador*, é o programa que lê documentos NCL e usa as bibliotecas multimídia do sistema para gerar uma apresentação correspondente. A distância semântica entre esses documentos, especificações declarativas de alto-nível, e a interface de programação dessas bibliotecas, em geral imperativa e de baixo-nível, exige a definição de uma estrutura de dados intermediária, chamada *modelo de apresentação*, que captura os dados e guia a apresentação do documento.

O processo de apresentação de um documento NCL consiste, portanto, de dois passos: conversão do documento e apresentação do modelo resultante. Na arquitetura do formatador cada passo é executado por um módulo correspondente. O primeiro módulo, chamado *conversor*, recebe como entrada um documento NCL e produz uma instância do modelo de apresentação. A saída do conversor serve como entrada para o segundo módulo, chamado *player*, que, a partir da instância do modelo (e da entrada do usuário), gera uma apresentação interativa correspondente. A Figura 1.1 apresenta a arquitetura básica do formatador NCL.

Nessa arquitetura, o modelo de apresentação funciona como interface de comunicação entre conversor e *player*. De certa forma, a complexidade e o grau de abstração do modelo determinam a distribuição das tarefas entre os módulos

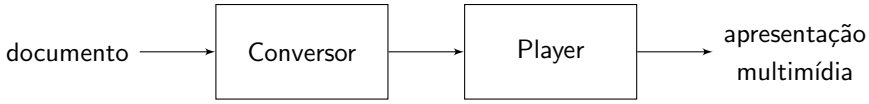

Figura 1.1 Arquitetura básica do formatador NCL.

do formatador. Um modelo próximo da linguagem simplifica o conversor, pois há pouca ou quase nenhuma manipulação estrutural, mas complica o *player*, que precisa transformá-lo em uma série instruções imperativas de baixo-nível. Por outro lado, um modelo próximo da máquina simplifica o *player*, pois aproxima o modelo das instruções, mas complica o conversor, que agora tem que converter as estruturas complexas do documento em primitivas do modelo. Essa interdependência entre conversor, modelo de apresentação e *player* sugere que a identificação do modelo adequado é um requisito fundamental para obtenção de um formatador eficiente e robusto.

A implementação mais recente do formatador NCL [3, 4], criada e mantida desde 2006 pelo Laboratório TeleMídia, é uma reescrita completa do formatador anterior HyperProp. A primeira versão do HyperProp [5] data de 1997. Na época, ainda não existia o NCL, apenas o NCM (*Nested Context Model*) [6], modelo conceitual que deu origem a linguagem. Em 2003, o HyperProp ganhou uma nova implementação capaz de apresentar documentos escritos na então recém-definida NCL [7]. Mais tarde, o desejo de expandir o alcance da linguagem para além dos computadores pessoais motivou o desenvolvimento da implementação atual, cujo objetivo principal era eliminar as limitações tecnológicas e arquiteturais do HyperProp que, de certa forma, impediam essa expansão. Em 2007, a implementação atual se tornou a implementação de referência do Ginga-NCL — *middleware* declarativo do sistema brasileiro de televisão digital [1].

Apesar das inovações, o modelo de apresentação da implementação de referência é praticamente idêntico ao do HyperProp, que tem como base o NCM. O problema com o NCM é que seu propósito original era facilitar a modelagem de cenários hipermídia e não facilitar uma apresentação eficiente desses cenários. Com isso, muitas das construções redundantes do NCM, projetadas para facilitar o processo de autoria, acabaram migrando para o modelo de apresentação das implementações. Em certo sentido, é natural que as implementações anteriores tenham utilizado o NCM como base para o seu modelo de apresentação. Primeiro, porque ele reflete a estrutura do documento. Além disso, o NCM foi amplamente testado e discutido — é, portanto, uma base segura. Finalmente, as versões mais recentes do NCM foram especificadas através de uma notação orientada a objetos, paradigma utilizado por ambas

as implementações.

A evolução da linguagem NCL é outro fator que contribui para o aumento da complexidade das implementações. Na medida em que funções eram adicionadas à linguagem, classes surgiam no modelo. Muitas dessas funções eram, na verdade, apenas formas alternativas de se especificar algum comportamento pré-existente. Porém, como não havia uma separação clara entre as funções primitivas e as demais, definidas a partir das primitivas, todas acabaram virando primitivas. Com o tempo, o modelo foi ficando cada vez maior e mais complexo. Na prática, essa complexidade se traduziu em mais código — atualmente 85 classes — e, consequentemente, em mais erros de codificação (ou *bugs*).

O elemento <switch> é um exemplo típico de construção redundante que acabou "migrando" para o modelo de apresentação das implementações. O <switch> de NCL funciona de forma semelhante ao *switch* das linguagens imperativas. Dada uma condição, um conjunto de casos é especificado. Se o valor da condição for igual ao valor de um dos casos então o código associado ao caso é executado — ou, em NCL, o conteúdo associado ao caso é apresentado. Da mesma forma que em C podemos substituir um *switch* por um conjunto de instruções *if-else*, em NCL podemos simular o comportamento do <switch> usando construções mais simples. Apesar disso, o <switch> possui representação correspondente tanto no modelo de apresentação quanto na API do *player* da implementação de referência.

A forma óbvia de simplificar o modelo de apresentação atual é simplificar a linguagem, i.e. remover do perfil EDTV tudo aquilo que é redundante. Como é de se esperar, essa linguagem mais simples produz um modelo mais simples. O problema com essa abordagem é que ela prejudica o autor de documentos, pois retira da linguagem construções projetadas para facilitar o processo de autoria.

A outra alternativa é delegar ao módulo conversor o trabalho de remover o que não é essencial. Ou seja, fazer do conversor o responsável por traduzir cada construção redundante do EDTV em uma série de primitivas e gerar o modelo de apresentação a partir do documento resultante. Dessa forma, as construções redundantes podem ser vistas como macros que quando expandidas geram o documento "real". Podemos identificar duas fases nesse processo:

1. *pré-processamento*, em que o documento EDTV é transformado em outro equivalente, porém mais simples, e

2. *parsing*, em que esse documento "bruto" é convertido no modelo de apresentação.

Apesar de ser (provavelmente) maior que o original, o novo documento é mais simples e, portanto, induz um modelo de apresentação mais simples. Do ponto de vista do autor, o perfil EDTV permanece inalterado.

Além de simplificar o modelo, a última abordagem traz alguns benefícios interessantes. Por exemplo, as fases podem ser executadas em momentos distintos. Além disso, o resultado do pré-processamento pode ser armazenado para agilizar (ou mesmo evitar) o processo de conversão. Outra vantagem é que a separação em fases mantém a linguagem da autoria isolada da linguagem da máquina de apresentação. Ou seja, mudanças no EDTV que não alteram a sua expressividade afetam apenas o conversor. Essa adaptabilidade é um requisito importante para o formatador NCL, visto que a linguagem está em constante evolução.

1.2 Objetivos

O objetivo geral desta dissertação é delimitar com precisão o menor subconjunto de elementos do perfil EDTV da NCL capaz de representar, de forma equivalente[1], qualquer documento EDTV válido. Todo documento no perfil básico BDTV é um documento EDTV válido. Portanto, tudo o que esta dissertação define sobre o EDTV vale também para o perfil BDTV. Dentre os objetivos específicos da dissertação podemos destacar:

- identificação dos elementos e atributos redundantes do EDTV,
- definição um procedimento de eliminação para cada redundância identificada,
- definição do menor perfil (em número de elementos) de expressividade equivalente, porém compatível com o EDTV — i.e. todo documento nesse perfil é também um documento EDTV,
- implementação de um conversor de documentos que permita reusar e/ou combinar os diversos procedimentos de eliminação.

A caracterização do subconjunto essencial do EDTV é um requisito fundamental para a definição de um modelo de apresentação mais simples. Apesar disso, a definição desse modelo não é objeto da dissertação, visto que essa definição envolve outros fatores que dependem da implementação do formatador.

[1] Dois documentos NCL são *equivalentes* se, e somente, ambos especificam a mesma apresentação.

1.3 Organização da Dissertação

Esta dissertação está organizada da seguinte forma. O Capítulo 2 apresenta a sintaxe e a semântica dos elementos que compõem o perfil EDTV da NCL. O Capítulo 3 identifica cada redundância presente no EDTV e define um procedimento de eliminação correspondente. Ainda nesse capítulo, é definido um perfil mínimo, compatível com o EDTV, chamado NCL *Raw*. O Capítulo 4 apresenta a arquitetura de um conversor de documentos EDTV para documentos *Raw*, e discute em detalhes a sua implementação. Finalmente, o Capítulo 5 apresenta as contribuições da dissertação e propostas de trabalhos futuros. A dissertação possui ainda dois apêndices. O Apêndice A contém a gramática completa do perfil EDTV, e o Apêndice B apresenta uma biblioteca *players* multimídia inspirada no perfil NCL *Raw*.

2
O Perfil EDTV

O perfil EDTV, *Enhanced Digital TV*, é o principal perfil da versão 3.0 de NCL. Um documento, ou aplicação, nesse perfil é um XML que descreve o relacionamento entre objetos de mídia (textos, imagens, vídeos, etc.) no espaço e no tempo. A sintaxe completa do EDTV é definida através de uma série de esquemas XML, disponíveis em `http://www.ncl.org.br` — cf. Apêndice A para um resumo do conteúdo desses esquemas. Já a semântica associada a cada construção do perfil pode ser encontrada em [2, 1]. Neste capítulo, apresentamos uma visão geral da sintaxe e da semântica dos principais elementos que compõem um documento no perfil NCL EDTV, ou simplesmente, documento EDTV. Este capítulo está organizado da seguinte forma. A Seção 2.1 define a estrutura básica de um documento EDTV. Como veremos, esta estrutura é formada por dois grandes grupos de elementos: cabeçalho e corpo. Ainda nessa seção, são definidas as convenções notacionais utilizadas pelas próximas seções. A Seção 2.2 descreve em detalhes os elementos que constituem o corpo do documento. Finalmente, a Seção 2.3 apresenta os elementos que compõem o cabeçalho do documento. De modo geral, ordem de apresentação das seções e dos elementos dentro de cada seção foi escolhida de forma a minimizar o número de referências a conceitos ainda não definidos.

2.1
Estrutura Básica

Um documento EDTV é uma aplicação XML e, portanto, consiste de elementos aninhados com subelementos ordenados. Cada um desses elementos possui associados um nome (ou *tag*) e uma lista possivelmente vazia de atributos do tipo chave-valor. Para o propósito de representação do documento, podemos considerar um XML como sendo uma árvore ordenada em que cada nó possui um rótulo [8]. Em XML, os elementos também podem conter *strings* arbitrárias, porém esse tipo de construção não aparece no EDTV.

Todo documento EDTV possui um nó raiz <ncl>, que contém o cabeçalho <head> e o corpo <body> do documento. O cabeçalho contém as declarações globais do documento, e o corpo especifica os objetos e relacionamentos que o compõem. O atributo obrigatório *id* do elemento <ncl> especifica o identificador do documento. O elemento <ncl> possui ainda um atributo op-

cional[1] *xmlns* cujo valor é a URI (*Uniform Resource Identifier*) do esquema correspondente. As três *tags* <ncl>, <head> e <body> são obrigatórias — i.e. devem estar presentes em todo documento EDTV. A estrutura básica de um documento EDTV é apresentada na Figura 2.1.

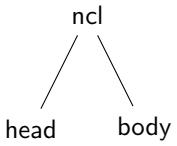

Figura 2.1 Estrutura básica de um documento EDTV.

No diagrama anterior, os nós da árvore representam os elementos da linguagem, enquanto as arestas e a disposição vertical dos nós indicam o sentido do relacionamento de inclusão — i.e. os elementos superiores incluem (ou contém) os inferiores. A cardinalidade do relacionamento é indicada através de subscritos do tipo "$1 \ldots n$." Nós que não possuem subscrito tem cardinalidade igual a um. Por exemplo, na Figura 2.1 o elemento <ncl> contém exatamente um elemento <head> e um elemento <body>. Em geral, a ordem em que os elementos aparecem na árvore é irrelevante. As únicas exceções são os elementos <bind>, filho do elemento <causalConnector>, e <bindRule>, filho dos elementos <switch> e <descriptorSwitch>.

2.2 Elementos do Corpo

O corpo do documento EDTV, delimitado pela elemento <body>, define o conteúdo, a estrutura, e a dinâmica da apresentação associada ao documento. O conteúdo — i.e. aquilo que é apresentado — é definido através dos nós de mídia e âncoras. A estrutura do documento é definida através dos nós de composição. A dinâmica da apresentação, incluindo o suporte à interatividade, é definida através dos elos e portas. De modo geral, podemos dividir os elementos que aparecem no corpo de um documento EDTV em três grupos: nós de conteúdo, nós de composição e nós de ligação (ou elos). A seguir, apresentamos os elementos que compõem cada um desses grupos.

[1] De agora em diante vamos omitir o adjetivo "opcional". A menos que indicado de outra forma, atributos definidos sem o adjetivo "obrigatório" devem ser considerados opcionais.

2.2.1 Nós de conteúdo

Em geral, os nós de conteúdo representam algum conteúdo multimídia externo — e.g. arquivo de imagem, áudio, etc. — que é referenciado pelo documento. Nós de conteúdo são definidos através de elementos <media>. De fato, a *tag* <media> define o que chamamos de objeto de mídia. Um *objeto de mídia* é um conteúdo associado a um conjunto de propriedades, elemento <property>, e âncoras, elemento <area>. Todo objeto de mídia possui um identificador único definido através do atributo obrigatório *id*. O conteúdo do objeto é especificado através do atributo *src*. A Figura 2.2 apresenta a estrutura de um objeto de mídia EDTV.

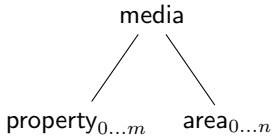

Figura 2.2 Estrutura de um objeto de mídia EDTV.

Os demais atributos do elemento <media> são: *type*, que define o tipo do conteúdo associado; *refer* e *instance* que indicam se o objeto é, na verdade, uma referência para algum outro objeto; e *descriptor*, que permite que diversos objetos compartilhem uma única declaração de valores iniciais de propriedades (cf. Seção 2.3.3).

Cada objeto de mídia possui um conjunto de propriedades pré-definidas que, em geral, determina a aparência do objeto durante a apresentação do documento. Toda propriedade possui um nome, atributo obrigatório *name*, e um valor, atributo *value*. As propriedades pré-definidas possuem um valor *default*. Portanto, ao escrever

```
<media id="sample1" src="image.png">
  <property name="transparency" value="50%"/>
</media>
```

estamos declarando um objeto de mídia com identificador "sample1" cujo conteúdo é uma imagem que será apresentada com a metade da transparência da imagem original. Se omitirmos o elemento <property> no trecho de código anterior, a imagem será apresentada com a transparência *default* (zero). Esse tipo de declaração corresponde, portanto, à inicialização da propriedade com algum valor possivelmente diferente do *default*. A Tabela 2.1 apresenta algumas propriedades pré-definidas do EDTV e respectivos valores iniciais.

Além das propriedades pré-definidas, que variam de acordo com o tipo do objeto, o EDTV permite a declaração de propriedades adicionais, cuja se-

Tabela 2.1 Algumas propriedades pré-definidas do EDTV.

Nome	Valor Inicial	Descrição
top	0	posicionamento a partir do topo
bottom	0	posicionamento a partir da base
left	0	posicionamento a partir da esquerda
right	0	posicionamento a partir da direita
width	0	largura do objeto
heigth	0	altura do objeto
transparency	0%	transparência do objeto
visible	true	indica se o objeto está visível ou não

mântica é dada pelo usuário. Nesse caso, a propriedade funciona como uma variável geral, que pode ser utilizada pelo autor do documento para guardar *strings* arbitrárias. É o atributo obrigatório *name* que determina se uma dada declaração refere-se a uma propriedade pré-definida ou não.

Objetos de mídia podem conter âncoras, elemento <area>, em que cada âncora representa um segmento do conteúdo do objeto. Por *default*, todo objeto de mídia define uma âncora implícita, chamada de âncora *lambda* (λ), que denota o conteúdo do objeto como um todo. Toda âncora possui um atributo obrigatório *id* cujo valor é o identificador da âncora. Os demais atributos — *coords*, *begin*, *end*, *text*, *position*, *first*, *last* e *label* — determinam o tipo da âncora em questão. Por exemplo, no trecho de código

```
<media id="sample2" src="audio.mp3">
  <area id="solo" begin="30s" end="45s"/>
</media>
```

o elemento <area> com *id* "solo" define uma âncora temporal, i.e. um intervalo de tempo, no conteúdo do objeto de mídia "sample2". Quando a âncora é apresentada, apenas o intervalo especificado é reproduzido. O perfil EDTV também define âncoras espaciais que permitem selecionar trechos de textos, ou regiões de imagens ou vídeos.

Propriedades e âncoras são o que chamamos de *interfaces*. Toda interface define eventos que podem ser relacionados através de elos (ou *links*), cf. Seção 2.2.3. Observe que o objeto de mídia em si não define uma interface. Se quisermos denotar o conteúdo do objeto como um todo devemos usar a âncora λ. No entanto, quando a interface do objeto não é especificada, e.g. durante a definição de um elo ou porta, assume-se que se trata da âncora λ.

2.2.2 Nós de composição

Os nós de composição permitem agrupar, sob uma mesma entidade, objetos de mídia e elos (ou regras) correspondentes. O principal nó de composição definido pelo EDTV é o contexto, elemento <context>. Contextos podem conter objetos de mídias, elos e outros contextos que por sua vez podem conter outros objetos de mídia, elos e contextos. A única restrição é que um contexto não pode conter, recursivamente, algum contexto que o contenha. Mais precisamente, a estrutura definida por um contexto e seus filhos é um grafo dirigido acíclico (ou DAG, *Directed Acyclic Graph*). O elemento <context> possui apenas dois atributos: *id*, atributo obrigatório que define o identificador do contexto; e *refer*, que indica se o contexto é, na verdade, uma referência para algum outro objeto.

Além dos objetos de mídia, elos e outros contextos, um contexto pode conter portas, propriedades e *switches* (o outro tipo de nó de composição definido pelo EDTV). A Figura 2.3 apresenta a estrutura de um contexto EDTV. O elemento <body> possui a mesma estrutura do contexto, a única diferença é que o seu atributo *id* é opcional.

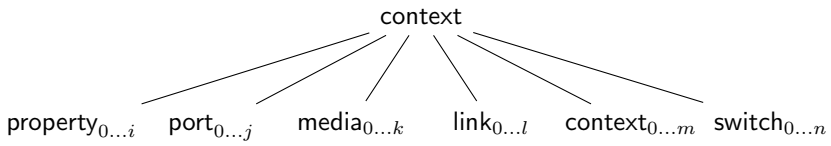

Figura 2.3 Estrutura de um contexto EDTV.

O contexto possui dois tipos de interface: propriedades e portas. As propriedades, elemento <property>, funcionam como variáveis locais do contexto. O comportamento dessas propriedades é análogo ao das propriedades sem-semântica dos objetos de mídia. Já as portas, elemento <port>, permitem que interfaces definidas dentro do contexto — e.g. mídias, outros contextos, etc. — sejam referenciadas por elos e portas de fora do contexto. Mais precisamente, por elos e portas definidos no contexto imediatamente acima na hierarquia. Dessa forma, dizemos que uma porta "mapeia" ou "torna visível", através do seu atributo obrigatório *component*, um objeto definido dentro contexto. Por exemplo, no trecho de código

```
<context id="outter">
  <context id="inner">
    <port id="p1" component="sample3"/>
    <media id="sample3"/>
    <media id="sample4"/>
```

```
    </context>
    ...
  </context>
```

os elos e portas definidos no contexto "outter" não podem referenciar diretamente os objetos de mídia "sample3" e "sample4". Porém, esses mesmos elos e portas podem referenciar a porta "p1", que torna visível a interface λ do objeto de mídia "sample3". Além do *component*, o outro atributo obrigatório do elemento <port> é o *id*, que especifica o identificador da porta. O elemento <port> possui ainda um atributo *interface*, usado nos casos em que é preciso mapear uma interface específica de algum objeto, e.g. uma propriedade de um objeto de mídia.

Além de mapear interface, as portas também definem a semântica da apresentação do contexto. Apresentar um contexto significa apresentar todas as interfaces mapeadas por suas portas. Dessa forma, ao definir uma porta estamos: (i) tornando a interface visível; e, ao mesmo tempo, (ii) associando a apresentação da interface à apresentação do contexto. Observe que é impossível apresentar um contexto que não possui portas, visto que o estado da apresentação do contexto depende exclusivamente do estado da apresentação de seus componentes. Se esses componentes não puderem ser acessados, i.e. se o estado da sua apresentação não puder ser modificado, então não há como interferir no estado da apresentação do contexto. Na Seção 2.2.3, os mecanismos que controlam a apresentação de interfaces são discutidos em detalhes.

O outro tipo de nó de composição definido pelo EDTV é o *switch*, elemento <switch>. Em termos de estrutura, podemos dizer que o *switch* é similar ao contexto. Ambos possuem os mesmos atributos e compartilham a mesma restrição de composicionalidade, i.e. o fato de ser um DAG. A grande diferença entre ambos está na semântica. No lugar de portas e elos, o *switch* utiliza regras para controlar a apresentação dos seus componentes. As regras são expressões Booleanas definidas no cabeçalho do documento (cf. Seção 2.3.6). Essas expressões fazem referência à variáveis globais mantidas num objeto de mídia especial chamado *settings* (cf. Seção 2.2.4). A Figura 2.4 apresenta a estrutura de um *switch* EDTV.

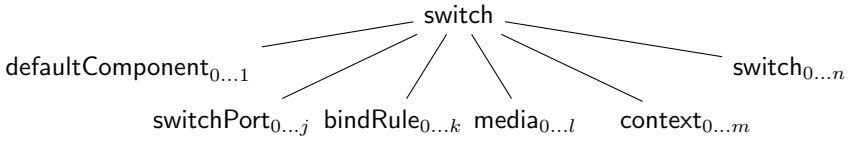

Figura 2.4 Estrutura de um *switch* EDTV.

O elemento <bindRule> associa uma regra, definida no cabeçalho, uma

interface — e.g. objeto de mídia, contexto, etc. — definida no corpo do <switch>. Os atributos obrigatórios *rule* e *constituent* do elemento <bindRule> contém, respectivamente, o identificador da regra e o identificador do objeto associado. Quando o *switch* é apresentado, as regras são avaliadas em sequência, uma a uma. Assim que uma delas é avaliada como verdadeira, o processo de avaliação é interrompido e o componente associado a regra é apresentado. Por exemplo, no trecho de código

```
<switch id="switch0">
  <bindRule rule="r1" constituent="sample5"/>
  <bindRule rule="r2" constituent="sample6"/>
  <media id="sample5"/>
  <media id="sample6"/>
</switch>
```

assim que o *switch* "switch0" é apresentado, a regra "r1" é avaliada. Se "r1" for avaliada como verdadeira, a interface λ do objeto de mídia "sample5" é apresentada. Caso contrário, a regra "r2" é avaliada. Se "r2" for avaliada como verdadeira, a interface λ do objeto de mídia "sample6" é apresentada, senão nada é apresentado. Observe que ordem de especificação das regras determina o comportamento do *switch*. De fato, esse é o único caso em que a ordem dos elementos influencia no resultado da apresentação. O perfil EDTV define ainda o elemento <defaultComponent> que pode ser usado para para definir um componente *default*, apresentado caso nenhuma regra seja avaliada como verdadeira.

Assim como o contexto, o *switch* define portas, elemento <switchPort>, que tornam visíveis para elementos externos as interfaces definidas no corpo do *switch*. A diferença entre o <switchPort> e o <port> do contexto é que as interfaces expostas pelo <switchPort> estão atreladas às regras do *switch*. Ou seja, uma interface mapeada por um *switchPort* é apresentada apenas quando a regra associada é avaliada como verdadeira.

2.2.3 Nós de ligação (elos)

Os elos (ou *links*) são o principal mecanismo para criação de apresentações dinâmicas e interativas no NCL EDTV. Através deles, é possível especificar, de forma declarativa, um conjunto de condições e ações associadas, que são disparadas sempre que as essas condições forem satisfeitas. Esse tipo de relacionamento entre condições e ações é chamado de relacionamento de causa e efeito.

No EDTV, ações e condições são, na verdade, eventos associados a interfaces (i.e. âncoras ou propriedades) dos objetos que compõem o documento. Cada âncora define exatamente dois eventos[2]: um *evento de apresentação*, que indica (ou controla) o andamento da a exibição do objeto; e um *evento de seleção*, que indica o andamento da seleção do objeto. Além disso, cada propriedade define um *evento de atribuição*, que indica (ou controla) o andamento da atribuição de um valor a uma propriedade. Todo evento define uma máquina de estados similar a apresentada na Figura 2.5. Na figura, os vértices representam os possíveis estados do evento: *occuring*, o evento está ocorrendo; *paused*, o evento está pausado; e *sleeping* o evento está dormindo ou (parado). E as arestas representam as transições entre os estados da máquina. O símbolo '|' no rótulo da aresta indica um conjunto de alternativas.

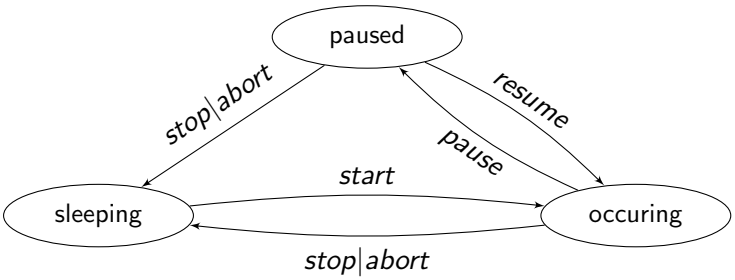

Figura 2.5 Máquina de estados de evento.

Condições e ações nada mais são do que as transições entre os estados. Uma condição é uma transição aguardada, e uma ação é uma transição induzida na máquina de alguma interface. Dessa forma, os elos conectam transições de máquinas de estado de eventos. No EDTV, para construir um elo entre duas interfaces o autor precisa especificar quatro parâmetros: (i) a interface observada, (ii) a interface a ser acionada, (iii) a transição aguardada, e (iv) a transição a ser induzida. De fato, os dois últimos parâmetros são especificados no conector, elemento <connector>, e não no elo, elemento <link>. O conector permite criar combinações entre transições aguardadas e induzidas sem especificar a quais interfaces essas transições pertencem. Dessa forma, todo elo referencia algum conector. A principal motivação por trás da separação entre <link> e <conector> é permitir que um mesmo conector possa ser reusado por diversos elos.

O motivo da separação entre a especificação das interfaces e a especificação das transições é que, com isso, é possível reusar um mesmo conector em elos distintos. (Conectores são discutidos em detalhe na Seção 2.3.5.)

[2]As interfaces associadas à nós de composição possuem um evento adicional, chamado de *evento de composição*, que indica (ou controla) o andamento da exibição da estrutura da composição.

A Figura 2.6 apresenta a estrutura de um elo EDTV. Todo elo define o atributo obrigatório *xconnector* que contém o identificador do conector associado. Um elo é basicamente, um conjunto de elementos <bind>. Cada <bind> associa uma interface, atributo obrigatório *component*, a um papel definido no conector, atributo obrigatório *role*. O papel determina o tipo da transição (condição ou ação) e a transição propriamente dita — e.g. *start*, *pause*, etc. Os elementos <linkParam> e <bindParam> permitem que o autor especifique no elo parâmetros a serem passados para o conector.

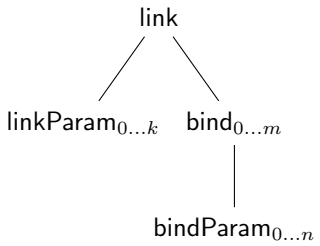

Figura 2.6 Estrutura de um elo EDTV.

2.2.4 Nó settings

O nó *settings* é um objeto de mídia especial que contém as variáveis globais do documento. Só pode haver um *settings* por documento. Apesar das variáveis do *settings* caráter global, o nó em si funciona como uma <media> qualquer. A Tabela 2.2 apresenta algumas variáveis de ambiente pré-definidas que podem ser exportadas através do nó settings.

Tabela 2.2 Atributos opcionais do elemento <transition>.

Nome	Descrição
system.language	linguagem de áudio
system.caption	linguagem de caption
system.screenSize	tamanho da tela
system.CPU	desempenho da CPU em MIPS
system.memory	memória em *megabytes*
system.operatingSystem	tipo de sistema operacional
system.luaVersion	versão da máquina Lua

2.3 Elementos do Cabeçalho

Os elementos do cabeçalho contém as declarações globais do documento. No EDTV, esses elementos são agrupados em *bases* de acordo com o seu tipo. Ao todo são definidas seis bases de elementos: base de documentos importados, base de regiões, base de descritores, base de transições, base de conectores e base de regras. As seções seguintes apresentam a sintaxe e semântica dos elementos que compõem cada uma dessas bases.

2.3.1 Base de documentos importados

A base de documentos importados, delimitada pelo elemento <importedDocumentBase>, contém as declarações de importação, i.e. referencias à documentos externos, do documento. A Figura 2.7 apresenta a estrutura de uma base de documentos importados do EDTV.

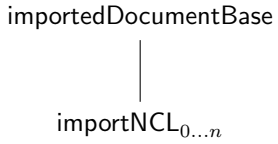

Figura 2.7 Estrutura de uma base de documentos importados EDTV.

Cada elemento <importNCL> contido na base aponta para algum documento externo e define um rótulo associado. Os atributos obrigatórios *documentURI* e *alias* do <importNCL> definem, respectivamente, a URI do documento referenciado e o rótulo associado. Na importação, as bases do documento importado são fundidas com as bases do documento importador. Ou seja, as bases de regiões, descritores, regras, etc. são incorporadas às bases correspondentes no documento importador (pai). Além disso, as mídias do documento importado tornam-se disponíveis para reúso através do atributo *refer*. No documento pai, o *id* dos elementos importados é o resultado da concatenação entre o rótulo definido no atributo *alias* do <importNCL> e o *id* original do elemento. O elemento <importBase>, filho das demais bases, pode ser usado para importar bases específicas ao invés de importar todas de uma vez.

2.3.2 Regiões

O elemento <region> define os parâmetros que determinam a dimensão e o posicionamento do objeto de mídia na tela. Além do *id*, a região possui os seguintes atributos: *left*, *right*, *top*, *bottom*, *height*, *width* e *zIndex*. Os quatro primeiros determinam o posicionamento do objeto, *height* e *width* determinam a sua dimensão, e o último, *zIndex*, especifica a precedência de sobreposição de uma região em relação as outras regiões. Regiões podem conter outras regiões. Nesse caso, os atributos do elemento filho (com exceção do *zIndex*) podem ser especificados de forma relativa aos atributos correspondentes na região pai. As regiões de um documento estão contidas nas bases de regiões do documento. Observe que pode haver mais de uma base de regiões por documento. A Figura 2.8 apresenta a estrutura de uma base de regiões EDTV.

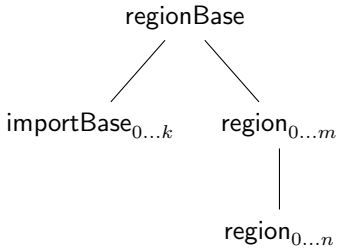

Figura 2.8 Estrutura de uma base de regiões EDTV.

2.3.3 Descritores

O elemento <descriptor> (cf. Seção 2.3.3) define os valores iniciais para as propriedades audiovisuais dos objetos de mídia associados. Esses valores podem ser especificadas através de atributos do elemento <descriptor> ou como parâmetros desse elemento, i.e. elemento <descriptorParam>. Os descritores do documento estão contidos na base de descritores de documento. Todo documento possui no máximo uma base de descritores. A Figura 2.9 apresenta a estrutura de uma base de descritores do EDTV.

O elemento <descriptorSwitch> permite associar a um objeto de mídia um conjunto de descritores alternativos. Cada descritor do conjunto possui uma regra associada. Quando o objeto de mídia é apresentado, as regras do <descriptorSwitch> associado ao objeto são avaliadas em sequência, uma a uma. Assim que uma delas é avaliada como verdadeira, o processo de avaliação é interrompido e o descritor correspondente é utilizado para inicializar as propriedades do objeto de mídia.

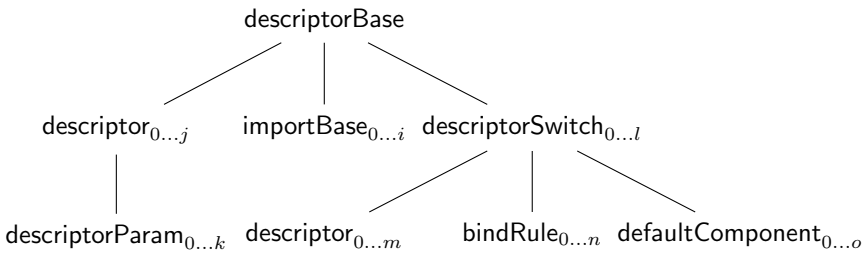

Figura 2.9 Estrutura de uma base de descritores EDTV.

2.3.4 Transições

As transições, elemento <transition>, especificam feitos visuais que podem ser associados ao início ou ao fim da apresentação de uma interface. Uma transição possui os atributos obrigatórios *id* e *type* que definem, respectivamente, o identificador e o tipo da transição. Além desses atributos, cada transição pode definir uma série de atributos opcionais listados na Tabela 2.3.

Tabela 2.3 Atributos opcionais do elemento <transition>.

Nome	Descrição
subtype	subtipo da transição
dur	duração da transição
startProgress	valor inicial do progresso da transição
endProgress	valor final do progresso da transição
direction	direção da transição ("forward" ou "backward")
fadeColor	cor final ou inicial do efeito *fade*
horRepeat	número de repetições no eixo horizontal
vertRepeat	número de repetições no eixo vertical
borderWidth	largura da borda em *pixels*
borderColor	cor da borda

As transições de um documento EDTV estão contidas na base de transições do documento, declarada no cabeçalho. Todo documento possui no máximo uma base de transições. A Figura 2.10 apresenta a estrutura da base de transições de um documento EDTV.

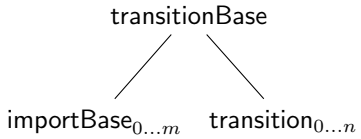

Figura 2.10 Estrutura de uma base de transições EDTV.

2.3.5 Conectores

O elemento <connectorBase> delimita a base de conectores do documento. Todo conector, elemento <connector>, define uma relação do tipo causa-efeito que pode ser usada por um ou mais elos. Nesse tipo de relação, um conjunto de condições deve ser satisfeito para que ações associadas sejam executadas. Uma condição simples, elemento <simpleCondition>, representa a "espera" de uma transição na máquina de estado de evento de alguma interface. Essas condições são usadas para construir condições compostas, elemento <compoundCondition>, que também pode conter testes de propriedade, elemento <assessmentStatement>. Ações simples, elemento <simpleAction>, representam o disparo de uma transição na máquina de estado de evento de alguma interface. Como na condição, as ações simples podem ser compostas através do elemento <compoundAction>. Quando o conjunto de eventos (e testes) esperados pela condição ocorre, a ação associada é executada, ou seja, o conjunto de transições especificado pela ação é disparado.

Os conectores do documento estão contidos na base de conectores. Todo documento EDTV possui no máximo uma base de conectores. A Figura 2.11 apresenta a estrutura de uma base de conectores EDTV.

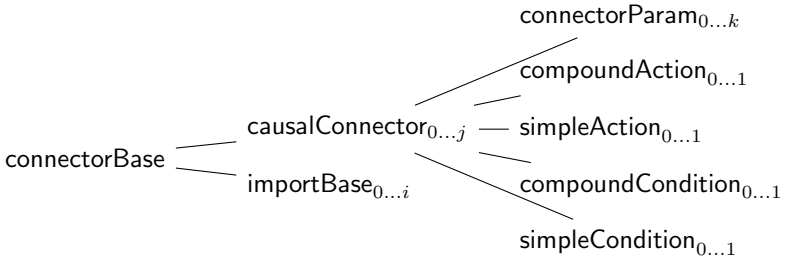

Figura 2.11 Estrutura de uma base de conectores EDTV.

2.3.6 Regras

A base de regras do documento é definida pelo elemento <ruleBase>. As regras simples, elemento <rule> definem testes Booleanos que referenciam as propriedades declaradas no objeto de mídia *settings* (cf. Seção 2.2.4). Regras simples podem ser reunidas em regras compostas através do elemento <compositeRule>. As regras definidas na base de regras podem ser referenciadas por elementos <switch> ou <descriptorSwitch>. A Figura 2.12 apresenta a estrutura da base de regras de um documento EDTV. Observe que pode haver no máximo uma base de regras por documento.

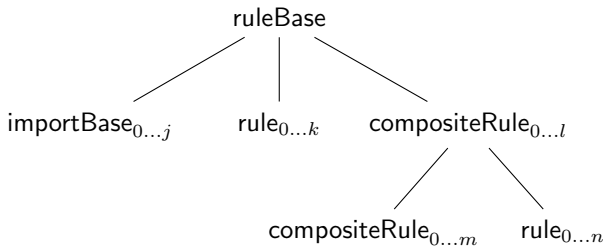

Figura 2.12 Estrutura de uma base de regras EDTV.

3
Eliminação das Redundâncias

O perfil EDTV define ao todo 45 elementos, alguns deles redundantes. Por exemplo, as propriedades *top*, *bottom*, *left*, *right*, que determinam o posicionamento do objeto de mídia na tela, podem ser definidas de três formas distintas: como propriedade do objeto de mídia, como parâmetro (ou atributo) do descritor associado ao objeto de mídia, ou como atributo da região associada ao descritor do objeto. Apesar dessa variedade de opções, em todos os casos o resultado final é o mesmo: a propriedade do objeto de mídia é inicializada com o valor especificado e o conteúdo do objeto é exibido na posição determinada. Este capítulo, trata, primeiramente, da definição dos procedimentos para eliminação desse tipo de redundância. Em seguida, a composição desses procedimentos é utilizada para obter um perfil mínimo, chamado NCL *Raw*, que é, ao mesmo tempo, livre de redundâncias e compatível com o EDTV. Dessa forma, documentos escritos nesse perfil podem ser exibidos pelos formatadores tradicionais. O capítulo está organizado da seguinte forma. A Seção 3.1 identifica as construções redundantes do EDTV e define os procedimentos de eliminação correspondentes. E a Seção 3.2 apresenta o perfil NCL Raw, definido a partir da composição desses procedimentos.

3.1
Procedimentos de Eliminação

Esta seção identifica cada construção redundante do perfil NCL EDTV e define um procedimento de eliminação correspondente. Para cada elemento ou atributo redundante r vamos definir um procedimento δ_r que o elimina. Seja Π um documento EDTV que contém r. Então, o resultado da operação $\delta_r(\Pi)$ é um documento Π' tal que Π' é equivalente a Π, i.e. ambos definem a mesma apresentação, e Π' não contém nenhuma ocorrência do elemento r.

3.1.1
Elementos <importNCL> e <importedDocumentBase>

O elemento <importNCL> é usado para importar documentos NCL (cf. Seção 2.3.1). Esse elemento define dois atributos: *documentURI* e *alias*. O primeiro contém a URI do documento importado, e o último define o seu

rótulo de importação — i.e. o prefixo usado pelos elementos do documento importador para referenciar elementos do documento importado. O procedimento de eliminação do elemento <importNCL> consiste em expandir a sua definição [1]:

> "As bases [do documento importado] são tratadas como se cada uma tivesse sido importada por um elemento <importBase>. [...] O <body> importado, bem como quaisquer de seus nós, pode ser reusado dentro do <body> do documento NCL que realizou a importação."

Seja Π um documento EDTV e seja I um elemento <importNCL> de Π que aponta para o documento externo Σ. O procedimento $\delta'_{\text{importNCL}}$ para eliminação de I consiste dos seguintes passos:

1. Substituir o cada elemento <importNCL> por elementos <importBase> correspondentes. Ou seja, para cada base de elementos B de Π é inserido um elemento <importBase> em B tal que o seu atributo *documentURI* é igual ao *documentURI* de I, e seu atributo *alias* é uma nova *string* única no documento. Além disso, os atributos dos elementos elementos Π que fazem referência aos elementos do cabeçalho de Σ devem ser atualizados para usar o novo *alias*.

2. Expandir a definição dos elementos <media>, <context> ou <switch> de Σ que são referenciados por elementos de Π através do atributo *refer*. Ou seja, cada elemento <media>, <context> ou <switch> de Π que referencia algum elemento correspondente em Σ deve ser substituído pela definição do elemento referenciado, incluindo seus filhos. Além disso, para evitar conflitos de nomes e manter válidas as referências ao elemento importado, o *id* do elemento, o *id* dos seus filhos, e os atributos *refer* correspondentes devem ser atualizados.

O procedimento $\delta'_{\text{importNCL}}$ elimina um elemento <importNCL> de Π. O procedimento $\delta_{\text{importNCL}}$ consiste em aplicar $\delta'_{\text{importNCL}}$ ao documento até que todos os elementos <importNCL> tenham sido removidos. Finalmente, o procedimento $\delta_{\text{importedDB}}$, que elimina a base de documentos importados, consiste em aplicar $\delta_{\text{importNCL}}$ ao documento e, em seguida, remover o elemento <importedDocumentBase> do cabeçalho do documento.

3.1.2 Elemento <importBase>

O elemento <importBase> (cf. Seção 2.3.1) permite que bases de entidades de documentos externos possam incorporadas às bases de um documento qualquer. Além dos atributos *documentURI* e *alias*, o <importBase> possui um atributo *region*, usado para definir a região pai destinatária (no caso da importação de bases de regiões). O procedimento de eliminação do elemento <importBase> consiste em copiar os elementos definidos no documento importado para a base correspondente no documento de origem.

Seja Π um documento EDTV qualquer e seja B um elemento <importBase> de Π que aponta para o documento externo Σ. O procedimento $\delta'_{\text{importBase}}$ para eliminação de B considera os seguintes casos:

1. *B é filho do elemento <descriptorBase>.* Nesse caso, os descritores e *switches* de descritores definidos na base de descritores de Σ são copiados para a base de descritores de Π e seus atributos *id* e as respectivas referências são atualizadas. Além disso, as bases de regiões (pode haver mais de uma), a base de transições e a base de regras de Σ também são importadas — casos (2), (3) e (5) abaixo —, já que descritores e *switches* de descritores de Σ podem referenciar elementos definidos nessas bases.

2. *B é filho de um elemento <regionBase>.* Nesse caso, há duas possibilidades:

   (a) Se o atributo opcional *region* estiver definido, então os elementos das bases de regiões de Σ devem ser inseridos como filhos do elemento <region> especificado — que pertence, necessariamente, a uma das bases de regiões de Π.

   (b) Caso contrário, o conteúdo das bases de regiões de Σ deve ser copiado para a base correspondente em Π.

3. *B é filho do elemento <transitionBase>.* Basta copiar as transições definidas na base de transições de Σ para a base de transições de Π. (Observe que todo documento define no máximo uma base de transições.)

4. *B é filho do elemento <connectorBase>.* Similar ao caso (3).

5. *B é filho do elemento <ruleBase>.* Similar ao caso (3).

Em todos os casos anteriores o *id* dos elementos importados é transformado em uma nova *string* única no documento e os atributos dos elementos de Π que fazem referência a esses *id*s são atualizados. O procedimento anterior

elimina apenas um elemento <importBase> de Π. O procedimento $\delta_{\text{importBase}}$ consiste em aplicar $\delta'_{\text{importBase}}$ ao documento até que todos os elementos <importBase> tenham sido removidos.

3.1.3 Elementos <region> e <regionBase>

O elemento <region> (cf. Seção 2.3.2) define os parâmetros que determinam a dimensão e o posicionamento do objeto de mídia na tela. O procedimento de eliminação do elemento <region> consiste em transformar os seus atributos em parâmetros dos descritores associados. Isso funciona porque os atributos e parâmetros especificados na região e no descritor realizam a mesma função, i.e. eles inicializam as propriedades correspondentes dos objetos de mídia associados. Como os parâmetros definidos no descritor têm precedência sobre os atributos da região, apenas aqueles que não aparecem duplicados devem ser copiados.

Seja Π um documento EDTV qualquer e seja R um elemento <region> de Π tal que R define uma lista de atributos (exceto *id*) $a_1, \ldots, a_n$ em que cada atributo é um par $\langle \text{nome}, \text{valor} \rangle$. O procedimento δ'_{region} para eliminação de R consiste em, para cada descritor D que referencia R através do atributo *region*, incluir em D uma lista $p_1, \ldots, p_n$ de elementos <descriptorParam> tal que os atributos *name* e *value* de cada p_i ($i \leq n$) correspondem, respectivamente, ao nome e valor definidos em a_i. Durante a avaliação de cada a_i é preciso converter os valores relativos em valores absolutos — i.e. torná-los independentes da região pai. Observe que a base de regiões de R pode conter o atributo opcional *device*, que especifica a classe dos dispositivos associados. Nesse caso, deve ser inserido em D um elemento <descriptorParam> adicional, com o atributo *name* igual a "device" e o atributo *value* igual ao valor do *device* da base de regiões de R. Finalmente, o último passo do procedimento é remover o atributo *region* do descritor D.

A função δ'_{region} elimina um elemento <region> de Π. O procedimento δ_{region} consistem em aplicar a função anterior ao documento até que todas as regiões tenham sido removidas. Finalmente, o procedimento δ_{regionB} consistem em aplicar δ_{region} ao documento e, em seguida, remover todos os elementos <regionBase> do cabeçalho do documento.

3.1.4 Elementos <transition> e <transitionBase>

O elemento <transition> (cf. Seção 2.3.2) define um efeito de transição que pode ser associado ao início ou ao final da apresentação de uma interface. O procedimento para eliminação do elemento <transition> funciona de forma similar ao procedimento apresentado na Seção 3.1.3.

Seja Π um documento EDTV qualquer e seja T um elemento <transition> de Π tal que T define uma lista de atributos (exceto *id*) $a_1, \ldots, a_n$. O procedimento $\delta'_{\text{transition}}$ para eliminação de T consiste em, para cada descritor D que referencia T através do atributo *transIn* (ou *transOut*), incluir em D uma lista $p_1, ..., p_n$ de elementos <descriptorParam> tal que, para $1 \leq i \leq n$: (i) o atributo *name* de p_i corresponde à *string* "transIn" — ou "transOut", caso D referencie T através desse atributo — concatenada ao nome de a_i; e (ii) o atributo *value* de p_i corresponde ao valor de a_i.

No procedimento anterior, o valor do atributo *transIn* (ou *transOut*) pode ser uma lista de *id*s de transições separados por vírgula. Nesse caso, o índice de T na lista (que começa em 0) precisa ser sufixado ao nome de cada elemento <descriptorParam> gerado. Por exemplo, considere o seguinte trecho de código:

```
<transition id="t1" type="barWipe" subtype="leftToRight"/>
<transition id="t2" type="fade" dur="5s"/>
<transition id="t3" type="clockWipe"/>
  ...
<descriptor id="d" transIn="t2" transOut="t1,t3"/>
```

Se aplicarmos o procedimento anterior a esse trecho o vamos obter o seguinte resultado:

```
<descriptor id="d">
  <descriptorParam name="transInType" value="fade"/>
  <descriptorParam name="transInDur" value="5s"/>
  <descriptorParam name="transOutType[0]" value="barWipe"/>
  <descriptorParam name="transOutSubType[0]" value="leftToRight"/>
  <descriptorParam name="transOutType[1]" value="clockWipe"/>
</descriptor>
```

A função $\delta'_{\text{transition}}$ elimina um elemento <transition> de Π. O procedimento $\delta_{\text{transition}}$ consiste em aplicar a função anterior ao documento até que todos os elementos <transition> tenham sido removidos. Finalmente, o procedimento $\delta_{\text{transitionB}}$ consiste em aplicar $\delta_{\text{transition}}$ ao documento e, em seguida, remover o elemento <transitionBase> do cabeçalho do documento.

3.1.5 Elemento <descriptor>

O elemento <descriptor> (cf. Seção 2.3.3) define as características audiovisuais dos objetos de mídia associados. Em geral, essas características são especificadas através de parâmetros, elementos <descriptorParam>. Porém, algumas delas, por exemplo *region*, *freeze* e *focusSrc*, podem ser definidas como atributos do elemento <descriptor>.

O procedimento de eliminação do elemento <descriptor> consiste, basicamente, em transformar os seus atributos e parâmetros em propriedades (elementos <property>) dos objetos de mídia associados. Como as propriedades definidas no objeto de mídia têm precedência sobre aquelas definidas no descritor, apenas as que não aparecem duplicadas devem ser copiadas. O procedimento definido a seguir assume que o descritor não possui os atributos *region*, *transIn* e *transOut*. Observe que esse procedimento não funciona para elementos <descriptor> declarados dentro de elementos <descriptorSwitch>. Esse último caso é tratado na Seção 3.1.6.

Seja Π um documento EDTV qualquer e seja D um elemento <descriptor> de Π tal que D possui uma lista de atributos (exceto *id*) $a_1, \ldots, a_m$ e uma lista de parâmetros $p_1, \ldots, p_n$ em que cada a_i $(i \leq m)$ e p_j $(j \leq m)$ denotam um par $\langle nome, valor \rangle$. O procedimento $\delta'_{\text{descriptor}}$ para eliminação de D consiste de dois passos:

1. Transformar os atributos de D em elementos <descriptorParam>. Ou seja, para cada atributo a_i é inserido em D um novo parâmetro p cujos atributos *name* e *value* correspondem, respectivamente, ao nome e valor definidos em a_i.

2. Transformar os parâmetros de D em propriedades das mídias associadas. Para cada mídia M que referencia D é inserida uma lista de propriedades em M cujos atributos *name* e *value* de cada propriedade corresponde, respectivamente, ao nome e valor definidos no parâmetro p_k $(k \leq m+n)$ do descritor D. Essas novas propriedades, assim como os parâmetros do descritor, não podem ser usadas diretamente em *links* e portanto devem declarar o atributo *externable* com valor "false".

A função $\delta'_{\text{descriptor}}$ elimina apenas um elemento <descriptor> de Π. O procedimento $\delta_{\text{descriptor}}$ consiste em aplicar a função anterior ao documento até que todos os descritores tenham sido removidos. Para eliminar a base de descritores do documento é preciso eliminar, além dos descritores, todas as ocorrências do elemento <descriptorSwitch>. O procedimento de eliminação desse elemento é assunto da próxima seção.

3.1.6 Elementos <descriptorSwitch> e <descriptorBase>

O elemento <descriptorSwitch> (cf. Seção 2.3.3) permite associar um conjunto de descritores alternativos a um objeto de mídia. A utilização de um determinado descritor depende da regra associada. As regras, elementos <rule> e <compositeRule>, são testes Booleanos sobre valores de propriedades do objeto de mídia *settings* (cf. Seção 2.3.6).

O procedimento de eliminação do elemento <descriptorSwitch> (e dos seus filhos) consiste em criar um elo para cada elemento <bindRule>, de forma que, se o objeto de mídia for apresentado (ação *start*) e o teste realizado pela regra for verdadeiro, então o valor de cada parâmetro do descritor correspondente deve ser atribuído a uma propriedade homônima do objeto de mídia (ação *set*). Para que esse procedimento funcione, é fundamental observar que:

1. A condição de cada elo gerado faz referência a uma ou mais variáveis do *settings*. Portanto, é preciso que esse nó esteja visível no mesmo contexto em que os elos são gerados.

2. O usuário pode especificar um descritor *default*, através do elemento <defaultDescriptor>, que é utilizado caso nenhuma regra seja avaliada como verdadeira.

3. A ordem de avaliação dos elos gerados é relevante. Ou seja, os elos gerados devem ser avaliados sempre na mesma ordem especificada pelos elementos <bindRule>.

Antes de definir precisamente o procedimento de eliminação do elemento <descriptorSwitch> precisamos definir um procedimento τ que converte uma regra R qualquer (simples ou composta) em um elemento <assessmentStatement> A equivalente. Para facilitar a apresentação do procedimento vamos assumir que cada propriedade do nó *settings* aparece no máximo uma vez em cada regra. Na prática, para evitar conflito entre os atributos *role* gerados, é preciso adicionar a cada *role* um índice que conta o número de ocorrências da propriedade.

O procedimento τ é definido da seguinte forma:

1. Se R é uma regra simples com os atributos *variable* $= x$, *comparator* $= y$ e *value* $= z$, então A é igual a:

```
<assessmentStatement comparator=y>
  <attributeAssessment role=x eventType="nodeProperty"/>
  <valueAssessment value=z/>
</assessmentStatement>
```

2. Caso contrário, R é uma regra composta com uma lista $r_1, \ldots, r_n$ de filhos e com o atributo *operator* $= w$. Nesse caso, A é igual a:

```
<compoundStatement operator=w>
  τ(r1) ··· τ(rn)
</compoundStatement>
```

Seja Π um documento NCL qualquer e seja S um elemento <descriptorSwitch> de Π tal que S possui uma lista $b_1, \ldots, b_n$ de elementos <bindRule> e uma lista $d_1, \ldots, d_n$ de elementos <descriptor>. Seja M, contido em um contexto C, um elemento <media> que referencia S através do atributo *descriptor*. O procedimento $\delta'_{\text{descSwitch}}$ para eliminação de S consiste dos seguintes passos:

1. Incluir em C um novo objeto de mídia que reúsa o nó *settings*, caso esse nó ainda não exista.

2. Incluir em C uma propriedade *counter* que será utilizada para forçar uma ordem de avaliação nos elos gerados no passo 4. O nome dessa propriedade deve ser único em C; seu valor inicial é irrelevante.

3. Adicionar o seguinte elo a C (caso o conector referenciado não exista, ele também deve ser adicionado à base de conectores de Π):

```
<link xconnector="onBeginSet">
  <bind role="onBegin" component=M/>
  <bind role="set" component=C interface=counter>
    <bindParam name="val" value="1"/>
  </bind>
</link>
```

   Esse elo funciona como um "gatilho" que dispara a avaliação das regras do <descriptorSwitch>.

4. Transformar cada <bindRule> em um par elo-conector equivalente. Ou seja, para cada elemento <bindRule> b_i de S ($i \leq n$) com atributo *constituent* $= d_i$ e regra associada r_i, inserir na base de conectores de Π os seguintes conectores (com *id*s únicos c_i e c'_i):

```
<causalConnector id=ci>
  <connectorParam name="val"/>
  <compoundCondition operator="and">
    <simpleCondition role="onEndAttribution"/>
    <assessmentStatement comparator="eq">
      <attributeAssessment role="testCounter"
```

```
        eventType="attribution" attributeType="nodeProperty"/>
      <valueAssessment value=i/>
    </assessmentStatement>
    τ(r_i)
  </compoundCondition>
  <simpleAction role="set" value="$val"
    max="unbounded" qualifier="seq"/>
</causalConnector>
```

Em que c'_i é similar a c_i porém, no lugar de $\tau(r)$, c'_i utiliza $\neg\tau(r)$, i.e. a negação de $\tau(r)$. Para obter essa negação basta adicionar o atributo *isNegated* ao elemento <compoundStatement> mais externo de $\tau(r)$; ou, caso o elemento mais externo de $\tau(r)$ seja um *assessmentStatement*, substituir o valor do atributo *comparator* pela sua negação.

Além disso, os seguintes elos l_i e l'_i devem ser adicionados a C, em que $p_1, \ldots, p_m$ é a lista dos nomes de propriedades do nó *settings* que aparecem em $\tau(r)$, e $\langle k_1, v_1\rangle, \ldots, \langle k_j, v_j\rangle$ é a de nomes e valores dos j parâmetros do descritor d_i referenciado por b_i.

```
<link id=l_i xconnector=c_i>
  <bind role="onEndAttribution" component=C interface=counter/>
  <bind role="testCounter" component=C interface=counter/>
  <bind role=p_1 component=settings interface=p_1/>
    ...
  <bind role=p_m component=settings interface=p_m/>
  <bind role="set" component=M interface=k_1>
    <bindParam name="val" value=v_1/>
  </bind>
    ...
  <bind role="set" component=M interface=k_j>
    <bindParam name="val" value=v_j/>
  </bind>
</link>
<link id=l'_i xconnector=c'_i>
  <bind role="onEndAttribution" component=C interface=counter/>
  <bind role="testCounter" component=C interface=counter/>
  <bind role=p_1 component=settings interface=p_1/>
    ...
  <bind role=p_m component=settings interface=p_m/>
  <bind role="set" component=C interface=counter>
    <bindParam name="val" value=(i + 1)/>
  </bind>
```

```
</link>
```

O propósito de l_i é simular a avaliação da regra r_i associada a b_i. Se r_i for avaliada como verdadeira então cada parâmetro do descritor d_i referenciado por b_i é transformado em uma ação *set* correspondente. Observe que para que c_i funcione é preciso adicionar a M os elementos <property> correspondentes (caso eles não existam). O *link* l'_i é utilizado para dar continuidade a avaliação das regras caso a r_i seja avaliado como falso.

5. Caso S defina um elemento <defaultDescriptor> então um elo l_{i+1} similar ao l_i (definido no passo 4) deve ser adicionado a C. Este novo elo testa se valor de *counter* é $n+1$ e, em seguida, atribui a cada propriedade de M o valor do parâmetro homônimo no descritor d_{i+1}, referenciado pelo elemento <defaultDescriptor> de S.

O procedimento $\delta'_{\text{descSwitch}}$ substitui cada elemento <descriptorSwitch> por conjunto de elos equivalentes. Esses elos funcionam da seguinte forma. Assim que o objeto de mídia M é apresentado, o elo "gatilho" (criado no passo 3) dispara o processo de avaliação sequencial das regras. Cada regra r_i possui dois elos associados: l_i e l'_i. A condição de l_i é verdadeira se, e somente se, a condição de r_i é verdadeira e a regra corrente é a regra i. A condição de l'_i é verdadeira se, e somente se, a condição de r_i é falsa e a regra corrente é a regra i. Portanto, se l_i é disparado então l'_i não é disparado e vice-versa. Além disso, l_i e l'_i só podem ser verdadeiras se o valor de *counter* for i. Logo, cada par l_i e l'_i é avaliado em sequência até que um dos seguintes casos ocorra:

- Algum l_k é avaliado como verdadeiro. Nesse caso, o processo de avaliação é interrompido (pois l_i não atribui valor à *counter*), e as ações *set* correspondentes são executadas.

- O último elo l'_n é avaliado como verdadeiro. Nesse caso, l'_n atribui o valor $n+1$ a *counter*. O único elo que testa esse valor, caso ele exista, é o elo adicionado no passo 5, que atribui os parâmetros do elemento <defaultDescriptor> de S ao objeto de mídia M. Se esse elo existir, o processo de avaliação é interrompido e as ações *set* correspondentes são executadas. Caso contrário, o processo de avaliação termina pois nenhum l_i ou l'_i $(1 \leq i \leq n+1)$ pode ser verdadeiro.

O procedimento anterior assume que o objeto de mídia M que referencia o elemento <descriptorSwitch> é filho de um contexto. Dessa forma, se M for

filho de um <switch>, antes de aplicar o procedimento é preciso converter o <switch> em um contexto usando o procedimento descrito na Seção 3.1.7.

A função $\delta'_{\text{descSwitch}}$ elimina apenas um elemento <descriptorSwitch> de Π. O procedimento $\delta_{\text{descSwitch}}$ consiste em aplicar a função anterior ao documento até que todos os elementos <descriptorSwitch> tenham sido removidos. Finalmente, o procedimento $\delta_{\text{descriptorB}}$ consistem em aplicar os procedimentos $\delta_{\text{descriptor}}$ e $\delta_{\text{descSwitch}}$ (nessa ordem), e, em seguida, remover o elemento <descriptorBase> do cabeçalho do documento.

3.1.7 Elementos <switch>

O elemento <switch> permite definir um conjunto nós alternativos cuja apresentação é associada a regras declaradas no cabeçalho do documento. Todo <switch> define um conjunto de interfaces, elementos <switchPort>, que mapeiam componentes internos do *switch*. O procedimento de eliminação do <switch> consiste, basicamente, em converter o *switch* em um contexto e usar portas e elos para simular os elementos <switchPort> e <bindRule> respectivamente. Observe que as restrições para simulação de regras através de elos, discutidas na Seção 3.1.6, também se aplicam no caso do elemento <switch>.

Seja Π um documento NCL qualquer e seja S um elemento <switch> de Π tal que S possui uma lista $b_1, \ldots, b_n$ de elementos <bindRule>, uma lista $m_1, \ldots, m_n$ de componentes, e uma lista de $p_1, \ldots, p_k$ de elementos <switchPort>, em que cada p_i $(0 \leq i \leq k)$ mapeia algum subconjunto de mídias de S. O procedimento δ'_{switch} para eliminação de S consiste dos seguintes passos:

1. Converter S em um contexto C contendo $m_1, \ldots, m_n$.

2. Incluir em C uma nova mídia que reúsa o nó *settings*.

3. Transformar cada elemento <switchPort> $p_1, \ldots, p_k$ em uma porta correspondente em C. Cada p_i $(0 \leq i \leq k)$ deve mapear um novo objeto de mídia *trigger* que servirá de âncora para os elos inseridos no passo 4. Ou seja, k objetos de mídia *trigger* devem ser inseridos em C, e cada p_i aponta para algum *trigger* distinto. Além disso, uma porta adicional, que representa o *switch* como um todo, deve ser incluída em C, junto com um novo *trigger*.

4. Transformar cada sequência de elementos <bindRule> associado lista de elementos <mapping> de cada <switchPort> em uma sequencia de

elos/conectores ancorados nos *triggers* correspondentes. Os elos/conectores avaliam as regras (na ordem correta) e, caso uma delas seja verdadeira, apresenta a mídia associada (ação *start*). Esse tipo de conversão é apresentada em detalhes na Seção 3.1.6.

5. Para cada ação diferente de "start", criar um elo ancorado no *dummy* que aplica a ação aos objetos de mídia mapeados por cada porta.

A função δ'_{switch} elimina apenas um elemento <switch> de Π. O procedimento δ_{switch} consiste em aplicar a função anterior ao documento até que todos os elementos <switch> tenham sido transformados em contextos.

3.2 Perfil Raw

A composição dos procedimentos apresentados na seção anterior define uma nova linguagem L, subconjunto do EDTV, com algumas características interessantes. Primeiro, a estrutura de L é mais simples e consistente. Todo documento é formado a partir da combinação de seis conceitos primitivos: mídia, âncora, propriedade, contexto, porta e *link*[1]. Outra característica importante, é que, por construção, L mantém ao mesmo tempo a expressividade e a compatibilidade com o EDTV. Ou seja, para todo documento EDTV existe um equivalente em L que também é um documento EDTV válido. Essas propriedades fazem de L uma base adequada para a definição de um novo perfil, chamado NCL *Raw*. A Tabela 3.1 apresenta a gramática do perfil *Raw*

Elemento	Atributos	Conteúdo
<ncl>	*id, title, xmlns*	(head?, body?)
<head>	-	(connectorBase?, meta*, metadata*)
<body>	*id*	(port | media| context| link| meta| metadata)*
<context>	*id*	(port| media| context| link| meta| metadata)*
<port>	*id, component, interface*	-
<media>	*id, src, refer, instance, type*	(area| property)*
<area>	*id, coords, begin, end, text, position, first, last, label*	-
<property>	*name, value, externable*	-
<link>	*id, xconnector*	(linkParam*, bind)+

Tabela 3.1 Gramática do perfil NCL Raw.

Os elementos estruturais <ncl>, <head>, e <body> foram mantidos para preservar a compatibilidade – eles são obrigatórios no perfil EDTV. Po-

[1]Podemos considerar os conectores como parte da definição dos *links*.

rém, regiões, transições, descritores e *switch* de descritores foram removidos. A única forma de definir propriedades é, portanto, através do elemento <property>, que também é usado para definir transições. Os mecanismos de importação <importNCL> e <importBase> foram removidos da linguagem. Desta foram, o atributo *refer* só pode ser usado para referenciar *id*s definidos no mesmo documento. Além disso, o reúso sintático não é permitido – só é permitido o reúso de objetos de apresentação. O perfil Raw elimina todos os elemento relacionados com o elemento <switch> – inclusive as regras definidas no cabeçalho. A única maneira especificar alternativas é através das condições nos *links*. Com a eliminação das regras, os únicos elementos que restaram do cabeçalho são os conectores, que definem as relações usadas nos *links*.

4
Conversor EDTV⇝Raw

O conversor EDTV⇝*Raw* é o programa que lê um documento escrito no perfil NCL EDTV e gera um documento *Raw* equivalente, i.e. que define a mesma apresentação. Este capítulo, apresenta a arquitetura e a implementação de um conversor extensível, que utiliza os procedimentos de eliminação de redundâncias definidos no Capítulo 3. A arquitetura do conversor é apresentada na Seção 4.1 e a sua implementação é discutida na Seção 4.2.

4.1
Arquitetura

A arquitetura do conversor depende, basicamente, de dois fatores: da sua entrada, i.e. o dado sobre o qual ele opera, e da sua saída, o resultado dessa operação. Existem duas entradas/saídas possíveis para um conversor EDTV⇝*Raw*. A primeira é, obviamente, a *string* do documento; e a segunda são os comandos de edição NCL — operações que permitem modificar o documento durante a sua apresentação. De certa forma, o conteúdo de ambas as representações é equivalente. Ou seja, todo documento pode ser transformado (ou "serializado") em uma lista de comandos de edição, e toda lista de comandos gera algum documento. A diferença está na natureza da representação, que no primeiro caso é estática (imutável) e no segundo caso é dinâmica (está sempre ocorrendo).[1] A Seção 4.1.1, a seguir, apresenta a arquitetura de um conversor estático (ou conversor de documentos). E a seção seguinte, Seção 4.1.2, discute a arquitetura de um conversor dinâmico (ou conversor de comandos de edição). Observe que em ambos os casos o conteúdo, i.e. aquilo o que está sendo convertido, é o mesmo. Portanto, os algoritmos de eliminação de redundâncias apresentados no Capítulo 3 valem tanto para o conversor de documentos quando para o conversor de comandos de edição.

[1] De certa forma, essa distinção entre processo estático e processo dinâmico na conversão de documentos NCL é similar à que ocorre entre as linguagens de programação compiladas *versus* linguagens interpretadas.

4.1.1 Conversão de documentos

Um documento EDTV é uma cadeia finita de caracteres que obedece uma certa sintaxe — a sintaxe definida pelos esquemas do perfil EDTV. O conversor de documentos EDTV⇝*Raw* é o programa que transforma um documento EDTV em um documento no perfil *Raw* que descreve a mesma apresentação. O processo de conversão de documentos consiste, basicamente, de três passos. Primeiro, a *string* do documento EDTV é transformada em uma representação intermediária Π em forma de árvore. Em seguida, os algoritmos $\delta_1, \ldots, \delta_n$ apresentados no Capítulo 3 são aplicados (numa determinada ordem) sobre essa representação Π, transformando-a em uma nova árvore Π' livre de redundâncias. Finalmente, Π' é transformada ("serializada") no documento *Raw* resultante. A Figura 4.1 abaixo ilustra cada um desses passos. Na prática, Π e Π' são as árvore dos elementos que compõem os documentos EDTV e Raw respectivamente. E cada função $\delta_1, \ldots, \delta_n$ descreve uma sequência de operações em árvore, e.g. inserir nó, remover nó, atualizar nó, etc.

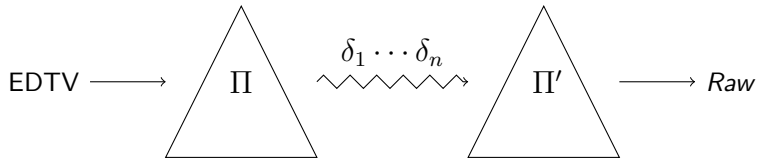

Figura 4.1 Processo de conversão de documentos.

Cada um dos passos anteriores possui um módulo correspondente na arquitetura do conversor. Dessa forma, o conversor de documentos EDTV⇝Raw é composto por três módulos interligados em um *pipeline*:

- *parser*, que recebe o XML do documento EDTV e gera uma árvore de elementos correspondente;
- *processador*, que opera sobre a árvore de elementos, transformando-a em uma nova árvore livre de redundâncias; e
- *gerador de código*, que recebe uma a árvore de elementos e gera o XML do documento *Raw* resultante.

Apesar de estarem interligados, cada um desses módulos é, de certa forma, independente. Ou seja, se a implementação permitir, cada módulo pode ser utilizado separadamente dos demais.

Até agora estamos considerando que o documento de entrada é um EDTV válido — i.e. livre de inconsistências sintáticas e semânticas. Porém, na prática, essa hipótese não é verdadeira. Inconstâncias sintáticas, inclusive as que

não podem ser detectadas a partir dos esquemas da linguagem, são tratadas pelo módulo *parser*. Para garantir que problemas semânticos não sejam transmitidos ao documento *Raw* resultante é preciso incluir no *pipeline* um passo de validação adicional entre o *parser* e o processador. O *validador* é o módulo responsável por verificar a consistência semântica da árvore de entrada e abortar o processo em caso de erro. (De fato, erros podem ocorrer em todas as quatro fases do processo.) A Figura 4.2 apresenta a arquitetura básica de um conversor de documentos EDTV⇝*Raw*.

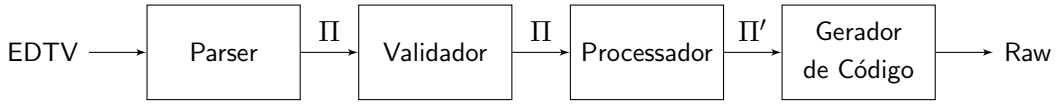

Figura 4.2 Arquitetura do conversor de documentos.

Na Figura 4.2, observe que os módulos *parser* e gerador de código são os únicos que lidam com a representação textual do documento. Isso indica que o núcleo do conversor formado pelo validador e o processador está, de certa forma, imune à mudanças na sintaxe do documento. Dessa forma, se a interface *parser*-validador for mantida, é possível "encaixar" no início do *pipeline* um *parser* de alguma outra sintaxe — e.g. JSON, ASN.1, *S-expressions* — sem afetar o funcionamento do conversor. O mesmo raciocínio se aplica ao módulo gerador de código. Com isso, há uma separação clara entre lógica de *parsing* e geração de código, da lógica da validação e conversão de documentos propriamente dita.

4.1.2 Conversão de comandos de edição

O procedimento de conversão de comandos de edição EDTV em comandos *Raw* é um pouco mais complexo que o procedimento de conversão de documentos discutido na Seção 4.1.1. Antes de apresentá-lo, precisamos definir o que é um comando de edição. Em NCL, um comando de edição é uma notificação enviada pela emissora de TV ou um evento postado por um *script* Lua que adiciona, atualiza ou remove algum nó do documento que está sendo apresentado (ou que encontra-se carregado). Cada comando possui um nome[2] e pode conter um ou mais parâmetros, entre eles o código XML do elemento a ser adicionado, removido ou atualizado.

[2]O nome dos comandos que adicionam e atualizam elementos é igual. A diferença é que se o elemento não existir ele é adicionado. Daí em diante, se o elemento não for removido, os próximos comandos homônimos são considerados comandos de atualização.

Como era de se esperar, alguns comandos de edição do EDTV não possuem comandos *Raw* correspondentes — e.g. *addDescriptor*, *removeRegion*, etc. Cada um desses comandos redundantes precisa ser mapeado em uma série equivalente de comandos Raw. Por exemplo, assim como um elemento <descriptor> do EDTV é traduzido em uma série de elementos <property>, cada *addDescriptor* precisa ser traduzido em uma série equivalente de comandos *addInterface*, que adicionam propriedades. O problema com esse tipo de conversão é que, se considerarmos apenas o conteúdo de cada comando, o conversor possui não possui informação suficiente para realizar a tradução. No exemplo anterior, para traduzir o comando *addDescriptor* o conversor precisa descobrir quais mídias no documento EDTV referenciam esse descritor, e essa informação não consta no *payload* do comando. Dessa forma, para que esse tipo de tradução seja possível, é preciso manter no conversor a árvore de elementos do documento EDTV "virtual", gerado a partir da sequência de comandos de edição EDTV recebidos até o momento.

Com isso, o procedimento de conversão de um comando de edição EDTV consiste, basicamente, de dois passos. Primeiro, aplicar o comando de edição EDTV à árvore do documento virtual. E, em seguida, traduzir o conteúdo do comando EDTV — consultando a árvore virtual, se necessário — em uma série equivalente de comandos *Raw*, que compõem a saída do conversor. Mais precisamente, podemos descrever o processo de conversão de um comando de edição EDTV em um comando *Raw* da seguinte forma. Seja C um comando de edição EDTV e seja Π o documento EDTV virtual mantido pelo conversor. O procedimento de conversão de C consiste em adicionar, atualizar ou remover elementos de Π e gerar uma série $R_1, \ldots, R_n$ $(n \geq 0)$ de comandos Raw equivalentes. Para gerar $R_1, \ldots, R_n$ é preciso considerar apenas o efeito de C sobre Π e traduzir esse efeito em uma série equivalente de operações. A Figura 4.3 ilustra esse procedimento.

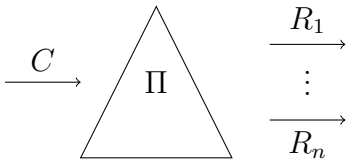

Figura 4.3 Processo de conversão de comandos de edição.

A arquitetura de um conversor de comandos de edição EDTV⇝*Raw* é formada basicamente por três módulos:

- *parser*, que converte o comando EDTV, escrito em uma determinada sintaxe, em uma série equivalente de operações sobre árvores de elementos EDTV;

- *processador*, que aplica as operações geradas pelo *parser* à árvore virtual do documento — mantida no processador — e, em seguida, utiliza essa árvore para gerar uma série equivalente operações sobre árvores de elementos *Raw*;

- *gerador de comandos*, que recebe uma série de operações sobre árvore de elementos *Raw* e gera comandos *Raw* em uma determinada sintaxe.

Nessa arquitetura, o conversor de comandos de edição EDTV⇝*Raw* funciona de forma contínua. Sua entrada é um fluxo (ou *stream*) de comandos de edição no perfil EDTV e a sua saída é um fluxo equivalente de comandos no perfil *Raw*. Assim como na arquitetura anterior, Seção 4.1.1, para garantir que inconsistências semânticas não sejam propagadas durante o processo de conversão, é preciso incluir um passo adicional de validação (módulo *validador*) entre o parser e o processador. Nesse caso, o validador também precisa ter acesso à árvore do documento virtual.

A partir de um documento sempre podemos obter uma sequência de comandos de edição que gera esse documento. Para isso basta percorrer a árvore definida pelo documento e gerar um comando para cada nó encontrado. Da mesma forma, o sentido oposto também é possível. Dada uma sequência de comandos podemos construir a árvore resultante. Isso significa que é possível utilizar o procedimento de conversão de comandos de edição para converter documentos.

4.2 Implementação

Esta seção descreve a implementação do pacote NCC (*NCL Converter Collection*)[3], um conjunto de ferramentas para conversão entre perfis e versões da linguagem NCL. A principal ferramenta do pacote é o programa de linha-de-comando *ncc* que converte um documento escrito em um formato de origem (opção `-from` ou `-f`) para um formato de destino (opção `-to` ou `-t`). O programa *ncc* é, na verdade, uma pequena camada de código que utiliza a biblioteca Libncc para realizar a conversão. A Libncc é o motor de conversão e principal artefato de programação instalado pelo NCC. Ela possui APIs que permitem controlar todas as fases do processo de conversão — do *parsing* à geração de código de destino. Tanto o programa quanto a biblioteca são escritos em ANSI C e rodam nas principais plataformas. Atualmente, as únicas

[3] `http://www.telemidia.puc-rio.br/~gflima/software/ncc`

dependências externas são as bibliotecas Libxml2[4], para *parsing* de XML, e Tmlib[5], para estruturas de dados complexas e funções auxiliares. Esta seção apresenta arquitetura e a implementação da Libncc.

4.2.1 Estrutura da Libncc

O processo de conversão de um documento NCL em outro semanticamente equivalente pode ser visto como uma série de passos (cf. Seção 4.1.1). Nessa arquitetura, cada passo corresponde a um módulo. Os módulos estão conectados em um *pipeline de conversão*, de forma que a entrada de cada módulo é a saída do módulo imediatamente anterior. O *pipeline* da Libncc é composto pelos seguintes módulos (nessa ordem):

- *parser* (`ncc_parser`), que transforma a representação textual do documento em uma árvore de elementos — i.e. a estrutura de dados interna que representa o documento;
- *processador* (`ncc_processor`), que aplica uma série de *operações* na árvore gerada pela fase anterior, de forma que a árvore resultante represente o documento de destino; e
- *gerador de código* (`ncc_codegen`), que transforma a árvore gerada pelo módulo processador na representação textual adequada.

Figura 4.4 *Pipeline* de conversão da Libncc.

A Figura 4.4 apresenta o *pipeline* de conversão da Libncc. Conforme discutido na Seção 4.2.2, cada módulo do *pipeline* possui uma API associada que é usada para controlar a execução da fase correspondente. De fato, essas APIs são independentes entre si, o que significa que cada módulo pode ser usado independentemente dos demais. Dessa forma, a Libncc pode ser utilizada em outros contextos, além da conversão de documentos. Por exemplo, podemos utilizar a saída do *parser* para criar um visualizador estrutural de documentos. Ou, quem sabe, criar uma árvore de elementos a partir de uma representação gráfica e usar o gerador de código para gerar o documento resultante. Ou

[4] `http://xmlsoft.org`
[5] `http://www.telemidia.puc-rio.br/~gflima/software/tmlib`

ainda, podemos trabalhar apenas com a representação gráfica da árvore de elementos e editá-la usando as operações do processador.

O tipo de manipulação descrito nos exemplos anteriores só é possível porque a árvore de elementos possui representação na API. Ou seja, a API da Libncc inclui a descrição do tipo de dado "árvore de elementos" e respectivas funções de manipulação. Os detalhes desta estrutura são apresentados na Seção 4.2.2, porém podemos adiantar algumas características desse tipo de dado abstrato (ou ADT, *Abstract Data Type*). Uma árvore de elementos é uma árvore n-ária composta por nós, em que cada nó possui um nome (ou *tag*), uma lista possivelmente vazia de nós filhos, uma lista possivelmente vazia de atributos (pares do tipo chave-valor), e (com exceção do nó raiz) um nó pai.

Uma vez obtida a árvore de elementos do documento de entrada, a fase de processamento (realizada pelo módulo processador) se resume a editar a árvore até obter o resultado desejado. Na terminologia NCC, cada edição corresponde a uma *operação*, uma função que recebe uma árvore de elementos e retorna a árvore modificada. A Libncc implementa diversas funções básicas (e.g. remoção de elementos inúteis, alteração global atributos, etc.) que podem ser utilizadas para construir operações complexas. Dessa forma, o procedimento de conversão de um documento em um perfil ou versão A para um perfil B consiste em aplicar uma série de operações $o_1, o_2, \ldots, o_n$ sobre a árvore de A. O processador é o módulo responsável por controlar essa aplicação.

4.2.2 API da Libncc

A interface de programação da Libncc é composta, basicamente, por três categorias de tipo abstrato de dados (i.e. dados e funções de manipulação): tipos que representam a estrutura do documento, tipos que representam as três fases do processo de conversão e tipos auxiliares. Os seguintes tipos representam a estrutura do documento (árvore de elementos):

`ncc_node_t`
: Representa um nó do documento. Cada nó possui uma lista encadeada de nós filhos, um ponteiro para o nó pai, um ponteiro para o próximo nó irmão e um ponteiro para o nó irmão anterior na hierarquia. Além disso, todo nó possui uma lista encadeada de atributos.

`ncc_attribute_t`
: Representa um atributo de um nó do documento. Cada atributo possui um nome, um valor, um ponteiro para o próximo atributo, um ponteiro

para o atributo anterior e um ponteiro para o nó pai, do qual o atributo faz parte.

As seguintes funções são usadas para manipular nós e atributos (cf. [9] para uma descrição detalhada de cada função):

```
ncc_node_create                    ncc_node_add_prev_sibling
ncc_node_destroy                   ncc_node_add_next_sibling
ncc_node_set_name                  ncc_node_unlink
ncc_node_get_n_children            ncc_node_replace
ncc_node_get_n_prev_siblings       ncc_node_get_attribute
ncc_node_get_n_next_siblings       ncc_node_set_attribute
ncc_node_add_child                 ncc_node_unset_attribute
```

Conforme discutido na seção anterior (Seção 4.2.1), o processo de conversão do NCC possui três fases: *parsing*, processamento e geração de código. As fases são implementadas, respectivamente, pelos módulos *parser*, processador e gerador de código. Cada módulo define um tipo *handle* utilizado para configurar e controlar a execução da fase. As fases são independentes umas das outros. Isso significa que cabe ao usuário da biblioteca – por exemplo, o programa *ncc* – configurar e executar cada uma das fases na ordem correta.

O módulo *parser* define o tipo *handle* `ncc_parser_t` e as seguintes funções associadas:

`ncc_parser_create ()`
: Cria e retorna um novo parser ou `NULL` em caso de falha.

`ncc_parser_destroy (parser)`
: Libera os recursos associados ao *parser*.

`ncc_parser_status (parser)`
: Retorna o *status* do *parser*. Essa função é utilizada para detecção de falhas durante o *parsing*.

`ncc_parser_parse (parser, buffer, size)`
: Realiza o *parsing* do conteúdo do *buffer* com tamanho *size*. Retorna a árvore de sintaxe correspondente ou `NULL` em caso de falha.

Atualmente, o *parser* reconhece apenas a sintaxe da NCL 3.0. A inclusão de suporte à outras versões da linguagem é discutida no Capítulo 5. Uma forma simples de fazer isso é adicionar um parâmetro extra na chamada `ncc_parser_create` que permita selecionar o *parser* desejado.

O módulo gerador define o tipo *handle* `ncc_codegen_t` e as seguintes funções associadas:

`ncc_codegen_create ()`
: Cria e retorna um novo gerador de código ou `NULL` em caso de falha.

`ncc_codegen_destroy (codegen)`
: Libera os recursos associados ao gerador *codegen*.

`ncc_codegen_status (codegen)`
: Retorna o *status* do gerador *codegen*.

`ncc_codegen_generate (codegen, tree, buffer, size)`
: Gera o código correspondente à árvore *tree* e escreve o resultado no *buffer* cujo tamanho é *size*. Retorna o número de *bytes* escritos ou -1 em caso de falha.

Assim como no caso anterior, a versão atual da biblioteca só gera código XML. A inclusão de suporte à outras sintaxes (e.g. Lua, JSON, *s-expressions*, etc.) é discutida no Capítulo 5. Novamente, a forma mais simples de se fazer isso é adicionar um parâmetro extra à chamada `ncc_codegen_create`. Como era de se esperar, há uma simetria entre a API dos módulos *parser* e gerador. Mais especificamente entre as funções `ncc_parser_parse` e `ncc_codegen_generate`. Essa simetria decorre da distinção entre conversão de forma e conteúdo abordada na seção anterior (Seção 4.2.1).

A API do processador difere um pouco das anteriores. O processador define um tipo *handle* `ncc_proc_t` e um tipo de função `ncc_op_fn` apresentado a seguir. O tipo `ncc_proc_t` possui as seguintes funções associadas:

`ncc_proc_create (oplist)`
: Cria e retorna um novo processador ou `NULL` em caso de falha. Além disso, essa chamada configura o processador para operar sobre a lista de operações *oplist*.

`ncc_proc_destroy (proc)`
: Libera os recursos associados ao processador *proc*.

`ncc_proc_status (proc)`
: Retorna o *status* do processador *proc*.

`ncc_proc_get_current_op`
: Retorna o índice da ultima operação executada.

`ncc_proc_proccess (proc, tree)`
: Executa as operações associadas ao processador *proc* sobre a árvore *tree*. Retorna a árvore resultante ou `NULL` em caso de falha.

A lista de operações, parâmetro da *oplist* chamada `ncc_proc_create`, é um vetor terminado com `NULL` de ponteiros para funções do tipo `ncc_op_t`. Funções desse tipo recebem dois argumentos: o processador que originou a chamada e a raiz da árvore a ser operada; e retornam a raiz da árvore modificada ou `NULL` em caso de erro. Para facilitar a implementação de conversores a biblioteca oferece uma série de operações pré-definidas, listadas abaixo (cf. [9] para uma descrição detalhada de cada uma dessa funções). Obviamente, o usuário da biblioteca pode definir suas próprias operações.

```
ncc_op_import_ncl
ncc_op_import_base
ncc_op_remove_unused_region
ncc_op_remove_unused_descriptor
ncc_op_remove_unused_media
ncc_op_remove_unused_context
ncc_op_remove_unused_switch
ncc_op_conv_region_to_descriptor
ncc_op_conv_transition_to_descriptor
ncc_op_conv_descriptor_to_property
ncc_op_conv_switch_to_context
```

Os três módulos da Libncc (*parser*, processador e gerador de código) são utilizados pelo programa de linha-de-comando *ncc* para construir um *pipeline* de conversão de documentos EDTV⇝*Raw*. A separação entre o código motor (Libncc) do código da interface (e.g. o programa de linha-de-comando *ncc*) permite que diversas interfaces compartilhem o mesmo código motor. De fato, o motor pode ser utilizado inclusive por outros programas ou bibliotecas. Esse tipo de separação é bastante comum em ferramentas UNIX (cf. padrão de projeto *separated engine interface* em [10]).

5
Conclusão

O formatador NCL é um programa complexo. Em parte, essa complexidade está associada a forma como as implementações atuais tratam as redundâncias presentes na linguagem de autoria. O objetivo deste trabalho foi delimitar com precisão o menor subconjunto da linguagem definida pelo perfil EDTV que é capaz de representar, de forma equivalente, qualquer documento EDTV válido. Dessa forma, as principais redundâncias no perfil EDTV foram identificadas e procedimentos de eliminação correspondentes foram definidos. A partir desses procedimentos foi definido um perfil mínimo, praticamente livre de redundâncias, chamado NCL *Raw*. Por ser mais simples e consistente, esse perfil simplifica o modelo de apresentação do formatador NCL. De fato, através do uso de conversores, é provável que este perfil permita uma melhor distribuição da complexidade entre os diversos subsistemas do formatador. Nesta dissertação, também foram propostas duas arquiteturas para um conversor de documentos EDTV em documentos *Raw*. A primeira considera a conversão de documentos propriamente ditos, i.e. a conversão entre arquivos XML. A segunda arquitetura trata da conversão de comandos de edição entre os dois perfis. A dissertação apresentou ainda uma implementação de um pacote para conversão entre perfis NCL, chamado NCC (*NCL Converter Collection*).

Podemos destacar as seguintes possibilidades de trabalhos futuros. No que diz respeito a linguagem, é possível pensar em perfis ainda menores compatíveis ou não o EDTV. Por exemplo, pode ser possível remover o reúso de instância ("instSame" e "gradSame") através da criação de portas e elos. Uma outra linha possível integrar outras linguagens de propósito similar (e.g. SMIL, HTML, etc.) no sistema, por exemplo definindo novos procedimentos de tradução. Dentre as principais características a serem incorporadas ao conversor NCC podemos destacar: a inclusão de chamadas que permitam monitorar o progresso da conversão, possibilidade de seleção da codificação de do documento gerado, implementação de *bindings* para outras linguagens de programação (Libncc), e inclusão de mecanismos que permitam adicionar dinamicamente novos *parsers*, operações, e geradores de código. Outra possibilidade é a incorporação de novas APIs para conversão de comandos de edição.

Referências Bibliográficas

[1] ABNT. *ABNT NBR 15606-2:2007 Televisão digital terrestre – Codificação de dados especificações de transmissão para radiodifusão digital. Parte 2: Ginga-NCL para receptores fixos e móveis – Linguagem de aplicação XML para codificação de aplicações.* Rio de Janeiro: ABNT - Associação Brasileira de Normas Técnicas, 2007.

[2] ITU-T. *ITU–T Recommendation H.761, 2009: Nested Context Language (NCL) and Ginga-NCL for IPTV services.* Geneva: International Telecommunication Union, 2009.

[3] MORENO, M. F. *Conciliando flexibilidade e eficiência no desenvolvimento do ambiente declarativo Ginga-NCL.* Tese de doutorado - PUC–Rio, Rio de Janeiro, 2010.

[4] MORENO, M. F. *Um middleware declarativo para sistemas de TV digital interativa.* Dissertação de mestrado - PUC–Rio, Rio de Janeiro, 2006.

[5] RODRIGUES, R. F. *Formatação temporal e espacial no sistema HyperProp.* Dissertação de mestrado - PUC–Rio, Rio de Janeiro, 1997.

[6] SOARES, L. F. G.; RODRIGUEZ, N. L. R.; CASANOVA, M. A. NCM: A conceptual model for hyperdocuments. *I Workshop em Sistemas Hipermídia Distribuídos – SBMídia95*, p. 40–46, 1995.

[7] MEIRE, J. A. NCL: Uma linguagem declarativa para especificação de documentos hipermídia na web. *VI Simpósio Brasileiro de Sistemas Multimídia e Hipermídia – SBMídia2000*, p. 79–95, 2000.

[8] VIANU, V. XML: From practice to theory. In: SBBD. 2003. p. 11–25.

[9] LIMA, G. A. F. *The NCL Converter Collection (for NCC version 1.0).* A ser publicado, 2011.

[10] RAYMOND, E. S. *The Art of UNIX Programming.* Pearson Education, 2003.

A
Gramática do Perfil EDTV

A Tabela A.1 apresenta a gramática pelo perfil NCL EDTV. Na tabela, os elementos aparecem listados em ordem alfabética. Parênteses são usados para delimitar listas de elementos. O símbolo '|' denota um conjunto de alternativas e os símbolo '?', '*' e '+' indicam, respectivamente, zero ou uma, zero ou mais, e uma ou mais repetições. Atributos sublinhados são obrigatórios.

Tabela A.1 Gramática do perfil EDTV

Elemento	Atributos	Conteúdo
<area>	*id, coords, begin, end, text, position, first, last, label*	-
<assessmentStatement>	*comparator*	(attributeAssesment, (attributeAssessment | valueAssessment))
<attributeAssessment>	*role, eventType, key, attributeType, offset*	-
<bind>	*id, component, interface, descriptor*	(bindParam)*
<bindParam>	*name, value*	-
<body>	*id*	(port | property | media | context | switch | link | meta | metadata)*
<bindRule>	*constituent, rule*	-

Tabela A.1 (continuação)

Elemento	Atributos	Conteúdo
<causalConnector>	*id*	(connectorParam*, (simpleCondition | compoundCondition), (simpleAction | compoundAction))
<connectorBase>	*id*	(importBase | causalConnector)*
<connectorParam>	*name*, *type*	-
<compositeRule>	*id*, *operator*	(rule |compositeRule)+
<compoundAction>	*operator*, *delay*	(simpleAction | compoundAction)+
<compoundCondition>	*operator*, *delay*	((simpleCondition | compoundCondition)+, (assessmentStatement | compoundStatement)*)
<compoundStatement>	*operator*, *isNegated*	(assessmentStatement | compoundStatement)+
<context>	*id*, *refer*	(port | property | media | context | switch | link | meta | metadata)*

Tabela A.1 (continuação)

Elemento	Atributos	Conteúdo
<defaultComponent>	*component*	-
<defaultDescriptor>	*descriptor*	-
<descriptor>	*id*, *player*, *explicitDur*, *region*, *freeze*, *moveUp*, *moveDown*, *moveLeft*, *moveRight*, *focusIndex*, *focusSrc*, *focusSelSrc*, *focusBorderColor*, *focusBorderWidth*, *focusBorder-Transparency*, *selBorderColor*, *transIn*, *transOut*	descriptorParam*
<descriptorBase>	*id*	(importBase | descriptor | descriptorSwitch)+
<descriptorParam>	*name*, *value*	-
<descriptorSwitch>	*id*	(defaultDescriptor?, (bindRule | descriptor)*)

Tabela A.1 (continuação)

Elemento	Atributos	Conteúdo
<head>	-	(importedDocumentBase?, descriptorBase?, regionBase*, transitionBase?, connectorBase?, ruleBase?, meta*, metadata*)
<importBase>	*alias*,*documentURI*,*region*	-
<imported-DocumentBase>	*id*	(importNCL)+
<importNCL>	*alias*, *documentURI*	-
<link>	*id*, *xconnector*	(linkParam*, bind)+
<linkParam>	*name*, *value*	-
<mapping>	*component*, *interface*	-
<media>	*id*, *src*, *refer*, *instance*, *type*, *descriptor*	(area | property)*
<meta>	*name*, *content*	-
<metadata>	-	RDF tree
<ncl>	*id*, *title*, *xmlns*	(head?, body?)

Tabela A.1 (continuação)

Elemento	Atributos	Conteúdo
<port>	*id*, *component*, *interface*	-
<property>	*name*, *value*, *externable*	-
<region>	*id*, *title*, *left*, *right*, *top*, *bottom*, *height*, *width*, *zIndex*	region*
<regionBase>	*id*, *device*, *region*	(importBase | region)+
<rule>	*id*, *var*, *comparator*, *value*	-
<ruleBase>	*id*	(importBase | rule | compositeRule)+
<simpleAction>	*role*, *delay*, *eventType*, *actionType*, *value*, *min*, *max*, *qualifier*, *repeat*, *repeatDelay*, *duration*, *by*	-
<simpleCondition>	*role*, *delay*, *eventType*, *actionType*, *value*, *min*, *max*, *qualifier*, *repeat*, *repeatDelay*, *duration*, *by*	-

Tabela A.1 (continuação)

Elemento	Atributos	Conteúdo
<switch>	*id*, *refer*	defaultComponent?, (switchPort | bindRule | media | context | switch)*
<switchPort>	*id*	mapping+
<transition>	*id*, *type*, *subtype*, *dur*, *startProgress*, *endProgress*, *direction*, *fadeColor*, *horRepeat*, *vertRepeat*, *borderWidth*, *borderColor*	-
<transitionBase>	*id*	(importBase | transition)+
<valueAssessment>	*value*	-

B
LibPlayer

A LibPlayer é uma biblioteca C para construção de apresentações multimídia interativas. A biblioteca foi projetada para facilitar o mapeamento de operações sobre objetos de mídia NCL em primitivas audiovisuais do sistema. Este capítulo descreve a arquitetura e a implementação dessa biblioteca.[1]

B.1
API de player

A principal abstração da API da LibPlayer é o *player*. Um *player* é um exibidor de mídia associado a algum conteúdo. Todo *player* possui propriedades, recebe ações e notifica eventos. As propriedades controlam como o conteúdo do *player* é apresentado. Por exemplo, o *player* de imagem possui a propriedade "transparency" que controla a transparência da imagem exibida. As ações, por sua vez, são comandos que podem ser enviados aos *players*. Dois exemplos típicos são as ações "start" e "stop" que disparam, respectivamente, o início ou fim da apresentação do *player*. Finalmente, toda ação gera um evento que é notificado através de *callbacks*.

O conceito de *player* é análogo ao objeto de mídia de NCL. A única diferença é que o *player* não define âncoras. De fato, cada âncora do objeto de mídia pode ser vista como uma determinada configuração do conjunto de propriedades do *player*. Por exemplo, uma âncora temporal do objeto de mídia define um intervalo associado à propriedade "time" do *player*. Desta forma, usando as propriedades e eventos de é possível construir o conceito de âncora.

A LibPlayer define dois tipos básicos: `lp_value_t` e `lp_map_t`. O primeiro define um valor genérico (`bool`, `int`, `double`, `char*` ou `void*`), usado para representar o valor das propriedades dos *players*. O último define um mapa do tipo chave-valor em que a chave é uma *string* e o valor é um `lp_value_t`. Esse mapa é usado para representar parâmetros das ações e eventos dos *players*.

A seguir apresentamos as principais funções da API de *player*. Para facilitar a apresentação, o tipo dos parâmetros e do retorno foi omito. Nas funções o parâmetro `p` indica o *player* sobre o qual a função é aplicada. A

[1] Código-fonte disponível em `http://www.telemidia.puc-rio.br/~gflima/software/libplayer`

maioria das chamadas retorna um valor `bool` que indica se a chamada foi bem-sucedida ou não.

`lp_player_new (source, mime)`
: Cria um novo *player* com conteúdo *source* e tipo *mime*, ambos *strings*. Retorna o endereço do novo *player* em caso de sucesso. Senão retorna `NULL`.

`lp_player_free (p)`
: Destrói o *player* `p`.

`lp_player_get (p, property, *value)`
: Armazena o valor da propriedade `property`, tipo *string*, em `*value`, tipo `lp_value_t*`. Retorna `true` se a propriedade está definida, caso contrário retorna `false`.

`lp_player_set (p, property, value)`
: Atribui `value`, tipo `lp_value_t`, à propriedade `property`, tipo *string*.

`lp_player_unset (p, property)`
: Remove a propriedade `property`, tipo *string*, do *player* `p`.

`lp_player_post (p, action, params)`
: Envia a ação `action`, tipo *string*, com parâmetros `params`, tipo `lp_map_t*`, para o *player* `p`. Se a ação for bem-sucedida, a função notifica as *callbacks* registradas em `p` e retorna `true`. Caso contrário, retorna `false`.

`lp_player_register (p, func)`
: Registra a *callback* `func`, cujo protótipo é o mesmo da função anterior, no *player* `p`.

`lp_player_unregister (p, func)`
: Remove a *callback* `func` da lista de funções registradas em `p`.

`lp_player_notify (p, action, params)`
: Notifica as *callbacks* do *player* `p` de que a ação `action`, tipo *string*, com parâmetros `params`, tipo `lp_map_t`, ocorreu.

Players podem conter outros *players*. Desta forma, é possível construir *players* que modificam ou controlam outros *players*. Um caso típico é o do *player screen*, que representa a janela da apresentação. Para que sejam apresentados, outros *players* devem ser adicionados ao *screen*. As funções seguintes manipulam *players* compostos.

`lp_player_add (p, child)`

Adiciona o *player* `child` à lista de filhos de `p`.

`lp_player_remove (p, child)`

Remove o *player* `child` da lista de filhos de `p`.

A interface de programação de *player*, apresentada acima, possui funções para alterar os valores de propriedades e executar ações. Porém, ela não define quais são essas propriedades e ações. Tais informações dependem da implementação de cada *player*. De fato, todo *player* define uma lista de propriedades e ações reconhecidas — i.e. as quais ele associa alguma semântica.

Para evitar inconsistências entre os *players* foram criadas classes de *players* que compartilham as mesmas propriedades e ações. Um *player* pode pertencer a mais de uma classe. Por exemplo, o *player* de vídeo pertence à classe dos *players* visuais e, ao mesmo tempo, à classe dos *players* de mídia contínua. A Tabela B.1 apresenta a lista de propriedades reconhecidas pelas classes atualmente definidas.

Classe	Propriedade	Descrição
Todos	`__name`	nome do plugin
	`__dlpath`	caminho da biblioteca
	`__mime_list`	lista de *mime-types* suportados
Visuais	`source`	conteúdo
	`x`	posição horizontal na tela
	`y`	posição vertical na tela
	`z`	índice de sobreposição
	`width`	largura em *pixels*
	`height`	altura em *pixels*
	`rotation`	rotação
	`transparency`	transparência
	`state`	"playing", "stopped", ou "paused"
	`original_width`	largura original em *pixels*
	`original_height`	altura original em *pixels*
Mídia contínua	`time`	tempo desde o último `start`

Tabela B.1 Propriedades reconhecidas por cada classe de *player*.

Atualmente, os *players* visuais reconhecem apenas as ações "start", que apresenta o conteúdo, e "stop", que pára a apresentação. Os *players* de mídia contínua reconhecem, além das anteriores, as ações "pause", que pausa a apresentação, e "seek" que avança ou retrocede o tempo da apresentação. Todos os *players* reconhecem a ação "step" que atualiza o estado global do *player*.

No caso de uma propriedade (ou ação) desconhecida, i.e. sem semântica associada, é recomendado que *player* trate-a como outra qualquer. Ou seja, armazene a propriedade ou, no caso da ação, notifique os ouvintes sobre a sua ocorrência. Este comportamento mantém a analogia com objeto de mídia NCL, em que propriedades sem semântica funcionam com variáveis do usuário.

O comportamento padrão e as listas de propriedades e ações reconhecidas são apenas guias definidos pela biblioteca. Os *players* distribuídos pela LibPlayer seguem esses padrões. Porém, cada *player* é livre para definir propriedades e ações da maneira que achar conveniente.

```
1  lp_player_t *screen;
2
3  void quit (lp_player_t * p, const char *action, lp_map_t * params) {
4    if (strcmp (action, "stop") == 0)
5      lp_player_post (screen, "stop", NULL);
6  }
7
8  int main (void) {
9    screen = lp_player_new (NULL, "application/x-libplayer-screen");
10   lp_player_t *image = lp_player_new ("sample.png", NULL);
11   lp_player_t *timer = lp_player_new (NULL,
12                         "application/x-libplayer-timer");
13   lp_player_set_d (timer, "duration", 2.0 /* seconds */ );
14   lp_player_register (timer, quit);
15
16   lp_player_add (screen, image);
17   lp_player_add (screen, timer);
18   lp_player_post (image, "start", NULL);
19   lp_player_post (timer, "start", NULL);
20
21   lp_player_post (screen, "start", NULL); /* event-loop */
22
23   exit (0);
24 }
```

Figura B.1 Exemplo de programa LibPlayer.

A Figura B.1 apresenta o código de um programa que usa a LibPlayer para exibir uma imagem durante dois segundos. Como sempre, a execução inicia pela `main`, linha 8. Na linha 9 é criado o *player* `screen`, que representa a janela da apresentação e, na linha 10, é criado o *player* `image`, que representa a imagem. Nas linhas 11–12, o *player* que conta o tempo (`timer`) é criado e inicializado com duração de dois segundos. A linha 11 registra a função `quit`, definida na linha 3, no *player* `timer`. Essa função aguarda o evento "stop" do `timer`, gerado pelo fim da duração, para terminar a apresentação (linha 5). Para que todos os *players* sejam apresentados eles precisam ser adicionados ao

`screen`, linhas 15–16, e iniciados, linhas 17–18. Finalmente, a linha 20 dispara o inicio da apresentação. Desse ponto em diante o controle é transferido para LibPlayer. Ao final de dois segundos, a chamada retorna e o programa termina (linha 22).

B.2 Plugins

Os *players* são implementados através de *plugins* em que cada *player* corresponde a um *plugin*. Um *plugin* é uma biblioteca dinâmica que implementa a API definida pela estrutura `lp_plugin_t`. Esta estrutura possui as seguintes funções.

`free (plugin)`
: Destrói o *plugin* `plugin`.

`get (plugin, property, *value)`
: Análoga à `lp_player_get` do *player*.

`set (plugin, property, value)`
: Análoga à `lp_player_set` do *player*.

`unset (plugin, property)`
: Análoga à `lp_player_unset` do *player*.

`exec (plugin, action, params)`
: Executa a ação `action`, tipo *string*, com parâmetros `params`, tipo `lp_map_t`, no *plugin* `plugin`. Retorna `true` se a ação for bem sucedida, caso contrário retorna `false`.

Além dessa API, todo *plugin* deve vir companhado de um arquivo de descrição que define o nome do *plugin*, o caminho para a biblioteca dinâmica que implementa o *plugin* e a lista de tipos *mime* suportados pelo *plugin*. O arquivo também pode conter uma lista de pares chave-valor que podem ser usados pelo *plugin* no momento da sua inicialização.

Os *plugins* são carregados pela LibPlayer no momento em que a biblioteca é inicializada. O carregamento funciona da seguinte forma. Primeiro, a biblioteca coleta os caminhos de carregamento definidos pela variável `plugin_path` do arquivo de configuração "libplayerrc". Se a variável estiver vazia é utilizado um valor padrão, definido no momento da compilação. Em seguida, para cada diretório especificado, a biblioteca procura subdiretórios contendo

o arquivo de descrição "plugin.desc". Finalmente, o *plugin* associado a cada "plugin.desc" válido é carregado.

Para facilitar a criação de *plugins* a LibPlayer fornece uma biblioteca auxiliar (Auxlib) que implementa diversas funções comuns, por exemplo, tratamento de propriedades e ações, verificação de erros, etc. Apesar de opcional, todos os *plugins* distribuídos com a LibPlayer utilizam essa biblioteca. Outra vantagem da Auxlib é que ela pode ser usada para verificar alguns dos padrões de codificação definidos anteriormente.

B.3 Loop de eventos

O *loop* de eventos controla como as ações do usuário e do ambiente afetam o resultado da apresentação. Na LibPlayer, o *loop* de eventos, contido no *player screen*, é disparado pela ação "start". Esse loop consiste, basicamente de três passos:

1. aquisição da entrada do usuário,
2. atualização do estado dos *players*, e
3. apresentação dos resultados.

Atualmente, o *loop* roda na *thread* principal da aplicação, o que significa que a chamada que dispara a apresentação só retorna quando o *loop* termina. Isso também significa que o controle da apresentação fica a cargo das *callbacks* registradas nos *players*.

```
1  state = "playing";
2  while (state != "stopped")
3    {
4      handle_input ();
5      clear (screen_surface);
6
7      for (i = 0; i < n; i++)
8        lp_player_post (players[i], "step", screen_surface);
9
10     update (screen_surface);
11     update_counters ();
12     synch (target_fps);
13   }
```

Figura B.2 *Loop* de eventos da LibPlayer.

A Figura B.2 apresenta o pseudo-código do *loop* de eventos da LibPlayer. Assim que o *screen* recebe a ação "start" ele muda o seu estado para "playing" e entra no *loop*, que é repetido enquanto o *screen* estiver tocando (linhas 1–2). O primeiro passo do *loop* é tratar os eventos de entrada, função `handle_input`, linha 4. Esta função verifica se há algum evento pendente na fila de eventos de entrada (eventos de teclado, mouse, etc). Se houver, a função retira esse evento da lista e posta sobre o próprio *screen* uma ação "input" passando como parâmetro o evento. Desta forma, as *callbacks* registradas no *screen* são notificadas e executam o código associado ao evento. Após todos os eventos terem sido tratados, a superfície da tela é preparada para um novo passo de renderização (linha 5).

O próximo passo consiste de atualizar o estado dos *players* – i.e avançá-los para a próxima amostra – contidos no *screen* (linhas 7–8). Ou seja, postar a ação "step" passando como parâmetro a superfície da tela. Desta forma, os *players* visuais podem se desenhar na janela de apresentação. Em seguida, nas linhas 10–11, a superfície da janela é atualizada para refletir as alterações e os contadores locais (tempo de apresentação, taxa de *frames*, etc.) são atualizados. Finalmente, a linha 12 chama a função `synch` que sincroniza o *loop* de acordo com a taxa de *frames* desejada. Ou seja, a função "dorme" o tempo necessário para manter a taxa de frames constante (no máximo `target_fps` *frames* por segundo).

Há diversas formas de otimizar a implementação atual do *loop*. A primeira, é controlar as operações de desenho para evitar operações de redesenho (*blits*) desnecessárias. Isso implica em retirar dos *players* a capacidade de desenhar diretamente na tela. Além disso, é possível criar uma *thread* separada para o *loop* e dessa forma permitir que a ação "start" do *screen* retorne imediatamente – o que torna a programação mais "interativa". Entretanto, para evitar problemas de concorrência, é preciso que a comunicação entre as duas *threads* seja controlada, por exemplo, através de uma fila de eventos.

B.4 Dependências

A LibPlayer (sem os *plugins*) depende apenas da biblioteca de portabilidade do TeleMídia (Tmlib)[2]. Atualmente, estão implementados os *plugins* de imagem, cronômetro (*timer*), texto e *screen*. O *screen* utiliza a LibSDL[3] para acessar os dispositivos de áudio e vídeo, e capturar a entrada do usuário. Os *plugins*

[2]Código-fonte disponível em `http://www.telemidia.puc-rio.br/~gflima/tmlib`
[3]`http://www.libsdl.org`

de imagem e texto utilizam a biblioteca Cairo[4] para manipulação de imagens e renderização de textos.

[4] http://www.cairographics.org