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.

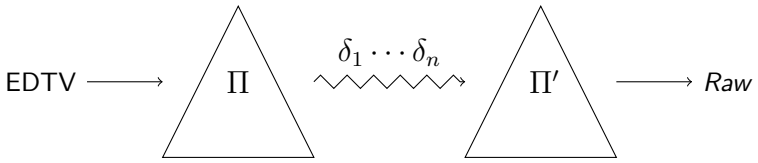

Figura 4.1 Processo de conversão de documentos.

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*.

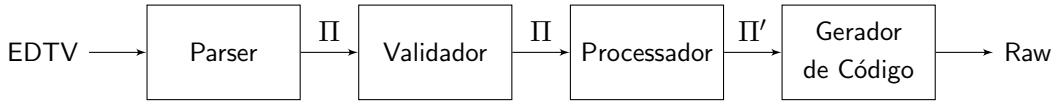

Figura 4.2 Arquitetura do conversor de documentos.

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.

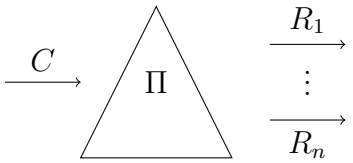

Figura 4.3 Processo de conversão de comandos de edição.

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]).