Geração Automática de *Frameworks* para Processamento de Documentos XML

Luiz Fernando Gomes Soares
Rogério Ferreira Rodrigues
Romualdo Monteiro de Resende Costa
Departamento de Informática – PUC-Rio
Rua Marquês de São Vicente, 225
Rio de Janeiro – 22453-900 – Brasil
{lfgs,rogerio,romualdo}@inf.puc-rio.br

RESUMO

Este artigo apresenta um modelo em duas camadas para o desenvolvimento de processadores de linguagens baseadas em XML. Na primeira camada, um *framework* (processador genérico) é automaticamente gerado a partir do XML *Schema* de uma determinada linguagem (linguagem de entrada). Além de definir a estrutura de processamento, o processador genérico implementa métodos que podem ser reusados por implementações de processadores/compiladores para a linguagem de entrada selecionada. Na segunda camada, um processador para um modelo de dados de saída específico é implementado, reusando a estrutura e os métodos especificados pelo *framework*. Baseado nessa proposta, diversos conversores foram construídos para algumas das principais linguagens de autoria multimídia e hipermídia, incluindo NCL, SMIL e XMT-O (MPEG-4).

ABSTRACT

This paper presents a two-layer model for the development of XML-based language compilers. In the first layer, a framework (generic compiler) is automatically generated from the XML Schema of a selected language (the input language). Besides defining the compiler structure, the generic compiler implements methods that can be reused by compilers of the selected input language. In the second layer, a compiler for a specific output data model is implemented, reusing the structure and methods specified by the framework. Based on this proposal, several compilers were built for some of the main hypermedia and multimedia authoring languages, including NCL, SMIL and XMT-O (MPEG-4).

Categories and Subject Descriptors

.

I.7.2 [Document Preparation]: Markup languages, Languages and systems, Hypertext/hypermedia.
D.3.4 [Processors]: Code generation, Compilers, Parsing, Translator writing systems and compiler generation.

General Terms

Standardization, Languages.

Palavras-chave

NCL, XML, *framework*, TV digital, SBTVD, *middleware* declarativo, Maestro.

1. INTRODUÇÃO

Documentos especificados de acordo com a definição de uma linguagem baseada em XML [16] possuem uma descrição estruturada do seu conteúdo, que normalmente deve ser pré-processada para que os documentos possam ser utilizados por um determinado aplicativo. Muitas vezes, diversos processadores de documentos XML são implementados, por aplicativos diferentes, para uma mesma linguagem. Como exemplo, podem ser citados os processadores de documentos XHTML [15], implementados em cada navegador WWW (*World Wide Web*).

Quando processadores analisam documentos de uma mesma linguagem de origem, algumas tarefas podem ser implementadas uma única vez e aproveitadas para vários modelos de dados de destino (aplicações) distintos. No caso específico das linguagens baseadas em XML, também é possível perceber características comuns existentes nos processadores para diferentes linguagens de origem, como a validação, a representação da estrutura do documento, a tradução estruturada das entidades (elementos e atributos), entre outras. A partir dessas constatações, uma estruturação em dois níveis para implementação de processadores (ou compiladores) de linguagens baseadas em XML é proposta, como se segue.

Em um primeiro nível um construtor recebe como entrada a definição de uma linguagem de origem em XML *Schema* [19] e automaticamente gera, como saída, um *framework* (ou

processador genérico) para construção de processadores da linguagem. No processador genérico gerado, as características comuns ao tratamento de documentos, descritos segundo a linguagem de origem, são implementadas, independente do modelo de dados de destino desejado.

Em termos gerais, o processador genérico construído implementa um conjunto de classes e métodos concretos que permitem identificar a estrutura relativa ao conteúdo de um documento, especificado segundo a linguagem XML de origem. Além dos métodos concretos, o processador genérico declara um conjunto de métodos abstratos (pontos de flexibilização do *framework*), que devem ser implementados quando da instanciação dos processadores propriamente ditos.

No segundo nível ocorre a instanciação do processador. Nessa etapa o programador pode herdar as funcionalidades do processador genérico e apenas acrescentar a implementação dos métodos abstratos. A forma de implementação desses métodos deve variar de acordo com o modelo de dados de destino escolhido. Esses modelos podem ser estruturas de dados específicas de um aplicativo, como, por exemplo, estruturas de exibidores ou editores de documentos, esquemas de bancos de dados, ou até mesmo descrições em outras linguagens, que podem ser ou não baseadas em XML.

Com base em uma implementação Java[1] da arquitetura proposta, foram gerados automaticamente três processadores genéricos para três linguagens de autoria de documentos multimídia/hipermídia: NCL (*Nested Context Language*) [13], SMIL (*Synchronized Multimedia Integration Language*) [20] e XMT-O (*eXtensible MPEG-4 Textual format*) [6]. NCL serviu de base para a proposta de um *middleware* declarativo para o Sistema Brasileiro de Televisão Digital (SBTVD). SMIL é um padrão W3C (*World Wide Web Consortium*) para autoria de documentos multimídia na Web e também tem sido adotado para concepção de aplicações multimídia para telefones celulares e outros dispositivos móveis. A linguagem também é tida como uma opção de extensão ao *middleware* declarativo do atual sistema europeu de TV digital (DVB) [3]. XMT-O é uma alternativa declarativa para especificação de apresentações MPEG-4 [6], outro importante padrão de codificação de vídeo em sistemas de TV digital.

A partir desses processadores genéricos, foram desenvolvidos processadores para a conversão entre as três linguagens (NCL, SMIL e XMT-O) e também conversores para outras linguagens, como, por exemplo, a conversão de documentos XMT-O para descrições em XMT-A [6]. Além da conversão entre linguagens XML, também foram desenvolvidos processadores com o objetivo de realizar a conversão das linguagens citadas para as estruturas de dados do sistema Maestro. Esse sistema reúne um conjunto de ferramentas de autoria e apresentação desenvolvidas como uma proposta para o SBTVD. É importante destacar que os processadores e conversores dessas e de outras linguagens assumem um papel importante no cenário de TV digital, onde sistemas distintos (por exemplo, europeu, japonês e americano) convivem, e um ponto chave, porém pouco explorado, é a questão da importação e exportação dos programas interativos.

Este artigo está organizado da seguinte forma. A Seção 2 realiza um breve resumo dos trabalhos relacionados ao processamento de documentos XML. A Seção 3 detalha o gerador automático de *frameworks* proposto. A Seção 4 apresenta a implementação de um exemplo de processador. Finalmente, as conclusões e os trabalhos futuros são apresentados na Seção 5.

2. TRABALHOS RELACIONADOS

Simple API for XML – SAX [11] e *Document Object Model* – DOM [18] são duas das principais abordagens para o processamento de documentos XML. A primeira abordagem é baseada em eventos que são disparados durante a leitura do documento XML. Na segunda abordagem, uma estrutura de dados em árvore é construída a partir da leitura do documento XML. Quando uma das duas abordagens é utilizada, cabe às aplicações a responsabilidade de interpretar a estrutura gerada (baseada em eventos ou baseada em árvore) e também a tarefa de realizar as conversões necessárias para o modelo de dados de destino. Ambas as abordagens oferecem um baixo nível de abstração para o desenvolvimento de processadores. Quase sempre, os processadores que utilizam essas abordagens são projetados sem possuir características de adaptabilidade e reusabilidade [8]. O gerador proposto neste artigo, assim como os processadores genéricos automaticamente construídos, foram desenvolvidos fazendo uso de DOM, porém oferecem um nível de abstração mais alto para a construção dos processadores propriamente ditos.

Outra abordagem voltada para o processamento de documentos XML é usualmente denominada *data binding* [7][10]. Nessa abordagem, um XML *Schema* é compilado gerando um conjunto de classes. Quando são processados os documentos XML, relativos à linguagem definida pelo XML *Schema* compilado, são geradas instâncias de objetos a partir das classes previamente concebidas (*unmarshaling*). Todavia, se o modelo de dados de destino não espelhar diretamente a estrutura do documento XML de origem, o resultado da geração automática precisa ainda passar por um processador para que a conversão seja realizada. Nesse caso, de forma similar às propostas do SAX e DOM, é total responsabilidade da aplicação o desenvolvimento de processadores.

Além das abordagens citadas, o processamento de documentos XML pode ser definido através de especificações utilizando uma outra linguagem XML. Com esse objetivo, a linguagem XSLT [14] é recomendada pelo W3C. XSLT possui alto poder de expressão, porém mesmo as conversões mais simples exigem a construção de um documento XSLT complexo [1]. Para realizar o processamento de cada documento XML de origem, é necessária a construção de um documento XSLT específico, embora existam trabalhos [1][12]

[1] http://java.sun.com

que propõem a construção automática de documentos XSLT. Por fim, é importante destacar que o uso de XSLT somente permite o processamento de documentos segundo uma linguagem XML de origem para representações textuais de destino (tipicamente um outro documento SGML [4]), ao contrário do proposto neste artigo, que contempla, além de transformações para representações textuais, geração automática de modelos de dados binários.

Uma outra estratégia para o processamento de documentos XML consiste em definir os elementos da linguagem XML de origem como objetos, cujo comportamento é determinado por um componente de software a ele mapeado. *XML Virtual Machine* - XVM [9] é uma arquitetura extensível que segue essa proposta. Para associar os componentes a elementos XML, XVM oferece um serviço de registro, onde os elementos de uma árvore DOM têm acesso a componentes definidos em um repositório. A arquitetura XVM possui características similares às do modelo proposto neste artigo, como a possibilidade do reuso do processamento da linguagem de origem para vários modelos de dados de destino distintos. No entanto, a especificação do serviço de registro na arquitetura XVM não é realizada de forma automática. Essa tarefa torna-se interessante apenas quando é possível reusar componentes de software. Quando o processamento tem por objetivo a conversão de uma linguagem de origem para vários modelos de dados de destino, como propõe este artigo, normalmente são necessários diversos componentes de software distintos, um para cada modelo. Dessa forma, na arquitetura XVM, o desenvolvimento de processadores de documentos XML mostra-se mais trabalhoso do que fazendo uso da arquitetura proposta neste trabalho.

3. GERADOR DE *FRAMEWORKS* PARA PROCESSAMENTO DE DOCUMENTOS XML

Diversas linguagens baseadas em XML definem um número elevado de elementos e atributos, o que influencia a complexidade de manipulação dessas linguagens. Para diminuir essa complexidade, várias linguagens têm sido definidas modularmente [6][13][15][20]. Módulos agrupam elementos e atributos XML que possuam relações semânticas entre si.

Um dos principais benefícios da definição das linguagens através de módulos é a facilidade para a criação de perfis. Perfis reúnem um subconjunto dos módulos oferecidos pela linguagem, definindo assim um subconjunto de funcionalidades apropriadas para a construção de uma determinada classe de documentos. Outra vantagem da estruturação da linguagem em módulos é a facilidade de reutilização; linguagens podem ser construídas reusando módulos oferecidos por outras linguagens.

Considerando a necessidade de processar documentos XML, as possibilidades de reuso do processamento e as dificuldades envolvidas nessa tarefa, esta seção descreve um gerador de *frameworks* para simplificar a construção de processadores de documentos descritos em linguagens baseadas em XML. Os *frameworks* são estruturados com o objetivo de refletir a organização modular da linguagem, quando for o caso. O processo de desenvolvimento de um processador é semi-automático, conforme ilustrado na Figura 1.

A principal entrada do gerador é a definição de uma linguagem em XML *Schema*. Em uma linguagem modular, cada módulo é definido através de um *schema* distinto. Módulos (*schemas* de módulos) podem ser agrupados através de um novo *schema*. Grupos de módulos podem, por sua vez, ser agrupados em super grupos e assim sucessivamente. O perfil de uma linguagem representa o agrupamento final e, como tal, é também descrito através de um XML *Schema*, à parte, que inclui os *schemas* (de módulos ou grupo de módulos) componentes.

Ao definir um agrupamento, elementos anteriormente definidos nos componentes do grupo podem ser redefinidos. Isto permite criar um espaço de nomes (*namespace*) [17] único para aqueles que venham a utilizar o agrupamento, além de possibilitar estender/restringir algumas das especificações, usualmente estabelecendo relações entre elementos de módulos distintos. Como conseqüência, na concepção dos documentos, autores não precisam conhecer e importar os vários módulos e agrupamentos, mas apenas o *schema* referente ao perfil (o agrupamento final).

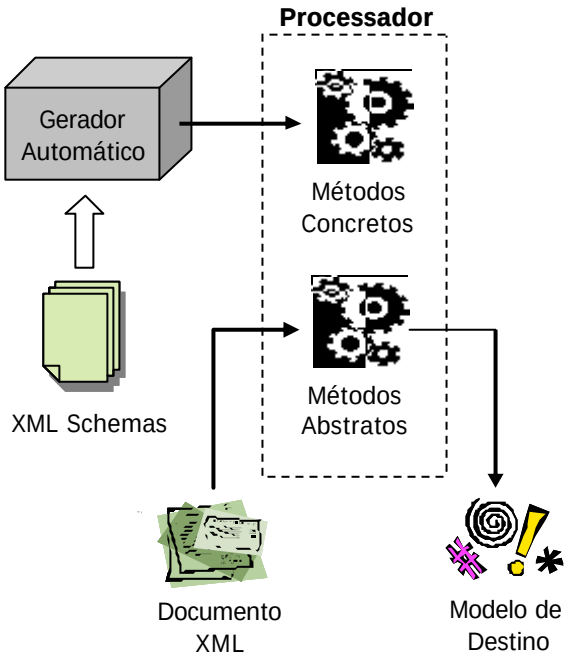

Figura 1. Modelo em dois níveis para a construção de processadores.

Evidentemente, linguagens não modulares possuem um único perfil e, conseqüentemente, um único *schema*. Outras linguagens, como SMIL, utilizam dois níveis de agrupamento; o primeiro denominado *área funcional*, e o segundo o próprio perfil da

linguagem[2]. XMT-O e NCL, em seu perfil principal, utilizam apenas o agrupamento de módulos em perfis.

Como exemplo de uma especificação modular utilizando XML *Schema*, as Figuras 2 e 3 ilustram fragmentos simplificados de dois módulos da linguagem NCL. A Figura 2 traz um fragmento do XML *Schema* que define o módulo *Media*. Na Figura 3 encontra-se a parte principal do módulo *Context*.

```
...
<!-- define the media element prototype - ->
<complexType name="mediaPrototype">
    <attribute name="id" type="ID" use="required"/>
    <attribute name="type" type="string" use="optional"/>
    <attribute name="src" type="anyURI" use="optional"/>
</complexType>

<!-- define the global media element in this module -->
<element name="media" type="media:mediaPrototype"/>
…
```

Figura 2. Fragmento de XML *Schema* do módulo *Media*

```
...
<!-- define the context element prototype - ->
<complexType name="contextPrototype">
    <attribute name="id" type="ID" use="required"/>
</complexType>

<!-- define the global context element in this module -->
<element name="context" type="contextcontextPrototype"/>
...
```

Figura 3. Fragmento de XML *Schema* do módulo *Context*.

O exemplo ilustra a declaração de um objeto de mídia (elemento *media* do tipo *mediaPrototype*), que possui como atributos um identificador único, um tipo (vídeo, áudio, texto etc.) e uma URI que informa a localização do conteúdo. Esse elemento e a descrição do seu tipo são feitas no módulo *Media*. Já no módulo *Context* é feita a declaração do elemento *context* (tipo *contextPrototype*), que contém apenas um identificador único.

A Figura 4 apresenta o *schema* de um perfil simples da linguagem NCL.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:media="http://www.telemidia.puc-rio.br/NCL/Media"
  xmlns:context="http://www.telemidia.puc-rio.br/NCL/Context"
  xmlns:profile="http://www.telemidia.puc-rio.br/NCL/Profile"
  targetNamespace="http://www.telemidia.puc-
rio.br/NCL/Profile"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified" >

  <!-- import the definitions in the modules namespaces - ->
  <import namespace=
    "http://www.telemidia.puc-rio.br/NCL/Media"
     schemaLocation=
    "http://www.telemidia.puc-rio.br/NCL/modules/Media.xsd"/>
   <import namespace=
    "http://www.telemidia.puc-rio.br/NCL/Context"
     schemaLocation=
    "http://www.telemidia.puc-rio.br/NCL/modules/Context.xsd"
/>

  <element name="media" substitutionGroup="media:media" />

  <complexType name="contextType">
    <complexContent>
      <extension base="context:contextPrototype">
        <choice minOccurs="0" maxOccurs="unbounded">
          <element ref="media:media" />
          <element ref="context:context" />
        </choice>
      </extension>
    </complexContent>
  </complexType>

  <element name="context" type="profile:contextType"
           substitutionGroup="context:context" />
</schema>
```

Figura 4. Especificação de um perfil simples NCL que combina os módulos *Media* e *Context*.

O perfil de linguagem especificado inclui os módulos *Media* e *Context*, e introduz a definição de um elemento *media*, que apenas substitui (atributo *subtitutionGroup* do *schema*) o elemento homônimo definido no módulo *Media*. O perfil também define um elemento *context*, que substitui e estende o contexto, definido através de um elemento homônimo no módulo *Context*, permitindo que um contexto contenha objetos de mídia e outros contextos. Como mencionado anteriormente, a razão de redefinir os elementos dos módulos no perfil é, além de estender/restringir algumas das especificações, criar um espaço de nomes único para aqueles que venham a utilizar o perfil da linguagem, evitando que os autores de documentos precisem conhecer e importar os vários módulos. Seguindo essa abordagem, é necessário conhecer apenas o *schema* referente ao perfil da linguagem.

[2] Nesse caso, são as áreas funcionais que oferecem uma visão das funcionalidades da linguagem, enquanto os módulos são definidos com uma granularidade mais fina, para beneficiar o reuso de partes bem específicas da linguagem.

As classes que modelam o gerador automático de um processador genérico para uma dada linguagem XML de origem estão ilustradas no diagrama UML da Figura 5.

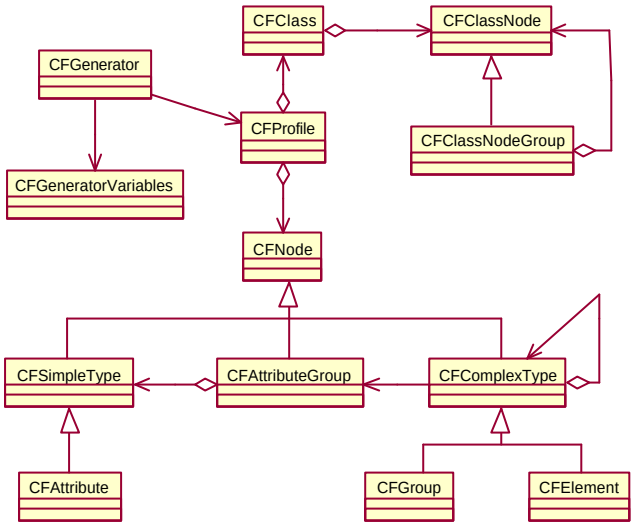

Figura 5. Diagrama de classes do gerador de processadores genéricos. Todas as classes fazem parte de um mesmo pacote. O prefixo "CF" deriva de "*Compiler Framework*".

O ponto de entrada para a geração de um *framework* (processador genérico) é a classe *CFGenerator*, que recebe uma instância da classe *CFGeneratorVariables*. A instância dessa última classe reúne os parâmetros necessários à interpretação do XML *Schema* da linguagem de origem. O principal atributo definido em *CFGeneratorVariables* é a URI que identifica a localização do *schema* do perfil da linguagem (os *schemas* dos módulos e demais agrupamentos, se existentes, são encontrados a partir do XML *Schema* do perfil). A classe *CFGeneratorVariables* contém também informações que podem não ser obtidas diretamente do *schema*, como o nome (do perfil) da linguagem.

A classe *CFGenerator* instancia um objeto da classe *CFProfile* refletindo a especificação de um perfil da linguagem de entrada. A partir do XML *Schema* desse perfil, são criados objetos das classes *CFAttribute*, *CFAttributeGroup*, *CFComplexType*, *CFElement*, *CFGroup* e *CFSimpleType*. Esses objetos contêm as informações descritas no *schema*. No exemplo das Figuras 2, 3 e 4, instâncias de *CFElement* seriam criadas para *profile:context*, *profile:media*, *context:context* e *media:media*[3]. Instâncias de *CFAttribute* seriam criadas para os atributos desses elementos, enquanto instâncias de *CFComplexType* modelariam os tipos *mediaPrototype*, *contextPrototype* e *contextType*.

Após a compilação do perfil e de todos os *schemas* referenciados pelo perfil, a classe *CFProfile* instancia um objeto da classe *CFClass* para cada espaço de nomes encontrado na definição da linguagem. Cada um desses objetos irá possuir objetos das classes *CFClassNode* e *CFClassNodeGroup*, referentes, respectivamente, aos elementos e aos grupos de elementos[4] definidos no espaço de nomes correspondente. Seguindo com o exemplo das Figuras 2, 3 e 4, a partir dos *schemas* da linguagem seriam criadas duas instâncias de *CFClass*, uma representando o módulo *Media* e outra representando o módulo *Context*. O objeto de *CFClass* associado ao módulo *Media* teria uma instância de *CFClassNode*, associada ao elemento *media:media*. Já o objeto de *CFClass*, representante de *Context*, teria uma instância de *CFClassNode* para o elemento *context:context*.

Se o perfil de uma linguagem definir um espaço de nomes separado, como ocorre na definição do perfil simples da Figura 4, *CFProfile* também criará uma instância de *CFClass* para agrupar os elementos e grupos de elementos do próprio perfil (*profile:media* e *profile:context*), visto que o perfil define o agrupamento final da linguagem.

O passo seguinte na geração é a criação automática de classes que irão tratar o processamento de cada um dos espaços de nomes definidos pela linguagem, incluindo o próprio perfil. Essas classes são criadas a partir das instâncias de *CFClass*. A Figura 6 ilustra o diagrama das subclasses geradas a partir de instâncias de *CFClass*. São justamente essas classes geradas que irão compor o processador genérico (*framework* de processamento) da linguagem de origem.

Cada classe do processador genérico tem o seu nome definido automaticamente pela composição do prefixo "CF", seguido pelo nome da linguagem de origem, o *alias* do espaço de nomes e a palavra "Compiler". As classes são então declaradas em um pacote que recebe o nome "compiler.<nome_da_linguagem>. Seguindo com o exemplo das Figuras 2, 3 e 4, e supondo que o nome da linguagem seja definido como "Ncl", o gerador criaria (a partir de objetos *CFClass*) arquivos com a especificação das classes *CFNclMediaCompiler*, *CFNclContextCompiler* e *CFNclProfileCompiler*, pertencentes ao pacote *compiler.ncl*. Todas as classes são geradas como subclasses de *CFSchemaCompiler*, que é uma classe declarada pelo próprio *framework*. Para facilitar a compreensão, o pacote *compiler* foi omitido da declaração das classes na Figura 6 e também será omitido no restante deste documento.

[3] Para evitar que os nomes fiquem muito extensos, as *strings* dos espaços de nomes foram substituídas pelos respectivos *aliases*.

[4] Os grupos de elementos e atributos são facilidades oferecidas por XML *Schema* que agrupam em um único componente um conjunto de elementos (ou atributos). Assim, outros componentes do *schema* que utilizem tais elementos (ou atributos), o fazem com uma única referência ao grupo.

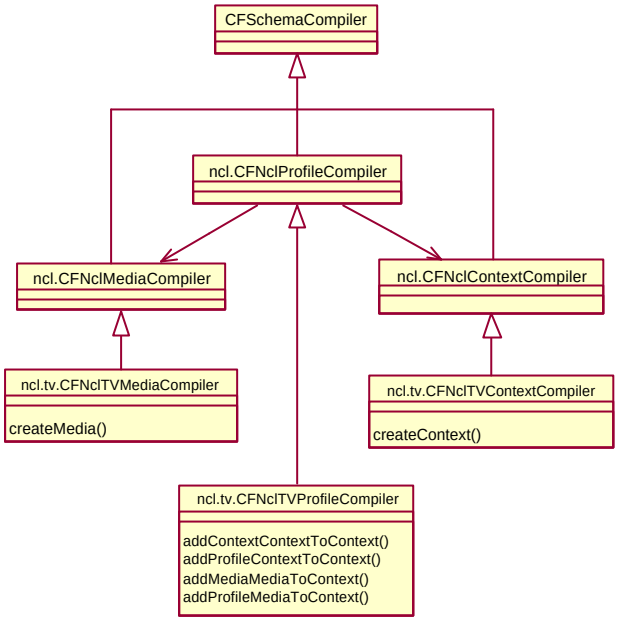

Figura 6. Diagrama parcial das classes geradas automaticamente para o *schema* da Figura 4.

Além das classes apresentadas na Figura 6, uma classe especial é automaticamente criada pelo gerador para oferecer o ponto de entrada para a compilação dos documentos que seguirão aquele perfil da linguagem. Essa classe é inserida no pacote gerado para o processador genérico e o seu nome é formado pela concatenação do prefixo "CF" com o nome da linguagem de origem e a expressão "*DocumentCompiler*", como, no exemplo apresentado na Figura 7, *CFNclDocumentCompiler*. Essa classe é declarada como uma subclasse de *CFDocumentCompiler*, que assim como *CFSchemaCompiler* é definida pelo *framework*.

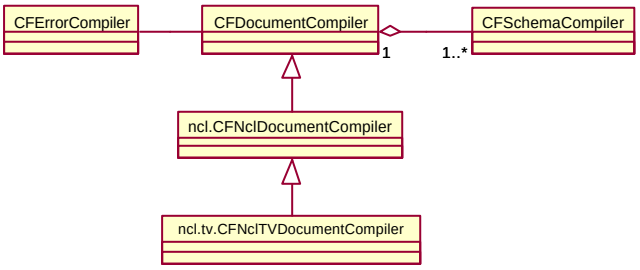

Figura 7. Diagrama parcial das classes geradas automaticamente para o processador.

A subclasse de *CFDocumentCompiler* também é conhecida como classe gerenciadora. Na classe gerenciadora são inseridos os métodos que permitem obter as referências para cada um dos compiladores dos espaços de nomes definidos pela linguagem (por exemplo, *getMediaCompiler*). Além disso, analisada a relação de dependência entre os *schemas*, é gerado automaticamente o corpo do método *setDependencies*. Para a linguagem apresentada na Figura 4, o método *setDependencies* teria o código descrito na Figura 8, uma vez que o perfil da linguagem importa os espaços de nomes definidos nos módulos *Context* e *Media*.

```
public void setDependencies () {
  getProfileCompiler().setContextCompiler(getContextCompiler());
  getProfileCompiler().setMediaCompiler(getMediaCompiler());
}
```

Figura 8. Código do método *setDependencies* gerado para *CFNclDocumentCompiler*.

O método *setDependencies* da classe *CFDocumentCompiler* é chamado na construção do processador, logo após a chamada ao método abstrato *createSchemaCompilers*. Esse último método deverá ser implementado pelo processador propriamente dito (instância do *framework*), como será exemplificado na Seção 4.

Outro método importante da classe *CFDocumentCompiler* é o método *compile*, que realiza a leitura de um arquivo XML e constrói uma árvore DOM do documento especificado nesse arquivo. Esse método, após a leitura, inicia o processamento do documento, solicitando que o elemento raiz do documento XML seja processado, através de uma chamada ao método abstrato *parseRootElement*.

O método *parseRootElement* é então automaticamente gerado na subclasse de *CFDocumentCompiler* (no exemplo, *CFNclDocumentCompiler*). Primeiro esse método descobre em qual espaço de nomes o elemento raiz está especificado e então efetua uma chamada ao método do tipo *parse* para processar esse elemento. O método do tipo *parse* é um dos tipos de métodos que compõem as classes geradas a partir dos *schemas* da linguagem. Os outros tipos são: *create*, *add*, *preCompile*, *posCompile* e *handleUnknownElementFrom*. A sintaxe e a semântica definidas pelo *framework* para os tipos de métodos dos processadores são as seguintes:

? Métodos iniciados pelo nome *parse*, como *parseContext*, são responsáveis por receber como parâmetro um elemento XML (*context*, no exemplo) e processá-lo a fim de gerar o objeto do modelo de destino representando esse elemento. O gerador cria um método concreto do tipo *parse* para cada elemento (e grupo de elementos) presente na linguagem. O método será composto por chamadas aos outros tipos de métodos do *framework*, como será exemplificado mais adiante. Os métodos do tipo *parse* fornecem a principal funcionalidade a ser herdada pelos processadores, pois implementam o caminhamento pela estrutura dos documentos.

? Métodos iniciados pelo nome *create*, como *createContext*, são responsáveis por receber um elemento XML (*context*) e criar o objeto representando o elemento no modelo de destino. Se o elemento substituir um outro elemento, o método é criado como um método concreto e apenas realiza uma chamada ao método do tipo *create* do elemento substituído. Caso contrário, o método é

criado como abstrato. Da mesma forma que os métodos do tipo *parse*, é criado um método *create* para cada elemento (e grupo de elementos) da linguagem.

? Métodos iniciados pelo nome *add* são responsáveis por adicionar um objeto do modelo de destino a outro objeto do modelo de destino que o contém. Por exemplo, *addMediaToContext* recebe duas referências, uma para o objeto representando o elemento *context* e outra para o objeto representando o elemento *media*, ambos já implementados no modelo de destino. Esses métodos são sempre abstratos e chamados pelos métodos do tipo *parse*. Para cada elemento da linguagem, deve existir um conjunto de métodos do tipo *add*; um para cada elemento (e grupo) de seu conteúdo.

? Métodos iniciados pelo nome *preCompile* e *posCompile*, como *preCompileContext* e *posCompileContext*, são, respectivamente, o primeiro e último métodos chamados pelos métodos do tipo *parse*. Esses métodos são declarados pelo *framework* como concretos para possibilitar (mas não obrigar) que processadores os sobrescrevam. Se o elemento em questão substituir um outro elemento, são inseridas chamadas aos métodos *preCompile* e *posCompile* do elemento substituído. Devem existir métodos desses tipos para cada elemento (e grupo de elementos) declarado na linguagem.

? Métodos concretos iniciados pelo nome *handleUnknownElementFrom*, como, por exemplo, *handleUnknownElementFromContext*, são chamados pelos métodos *parse* quando um elemento filho não é reconhecido como conteúdo do elemento pai. Por serem concretos, processadores podem, mas não são obrigados a, re-implementar esses métodos para tratar elementos não conhecidos previamente pelo *framework*, uma vez que não estão declarados no *schema*. Esses métodos oferecem tanto um mecanismo de extensão como de tratamento de exceções no processamento de documentos da linguagem. Deve ser implementado um método desse tipo para cada elemento (e grupo de elementos) da linguagem. Da mesma forma que os métodos do tipo *create*, *preCompile* e *posCompile*, para elementos que são declarados substituindo outros elementos, são adicionadas chamadas aos métodos *handleUnknownElementFrom* dos elementos substituídos.

Quando um elemento em um documento XML não é reconhecido pelo processador, o documento pode não estar bem formado em relação ao seu XML *Schema*. Ocorrências dessa natureza, normalmente, representam erros que devem ser claramente reportados a fim de serem resolvidos. Para modelar esses e outros erros, que possam ocorrer durante o processamento de documentos, a classe *CFErrorCompiler*, apresentada na Figura 7, é automaticamente construída pelo *framework*. Essa classe define os atributos: *status*, *description*, *element* e *file*, cujos valores são especificados pelo construtor da classe. O atributo *status* define o tipo de problema, por exemplo, erro ou advertência. O atributo *description* armazena uma descrição pré-definida do tipo do erro ocorrido. O atributo *element* armazena uma referência para o elemento onde o erro ocorreu. O atributo *file* é uma referência para o arquivo onde o erro foi identificado.

Para registrar os erros ocorridos durante o processamento, são definidos métodos iniciados pelo nome *register*, como os métodos definidos na classe *CFDocumentCompiler*. Essa classe define os métodos *registerError*, *registerCreationError*, *registerAddElementError* e *registerUnknowElement*. O método *registerError* cria uma instância de *CFErrorCompiler* com os parâmetros recebidos (status, local e descrição do erro) e a insere na lista de erros ocorridos. Os demais métodos citados também criam instâncias de *CFErrorCompiler*, porém com descrições padronizadas, que indicam a situação em que o erro foi detectado. O método *registerCreationError* é chamado quando um erro é identificado na criação do elemento (método *create*). O método *registerAddElementError* é chamado quando um erro é identificado na adição de um objeto do modelo de destino a outro objeto do modelo de destino que o contém (método *add*). Finalmente, o método *registerUnknowElement* é chamado quando um elemento que não se encontra no XML *Schema* da linguagem é encontrado (método *handleUnknownElementFrom*).

Pode-se exemplificar a sintaxe e a semântica propostas para os métodos do *framework* a partir do elemento *context* da Figura 4. A classe *CFNclProfileCompiler* do processador genérico irá implementar o método concreto *parseContext*. Esse método, representado em pseudo-código na Figura 9, recebe um elemento XML do tipo *context* (*contextElement*) como parâmetro e retorna a implementação desse elemento no modelo de destino.

Inicialmente, no corpo do método *parseContext* é realizada uma chamada ao método concreto *preCompileContext*. Esse método pode alterar o *contextElement* e retornar um elemento do tipo *context* pré-processado. Em seguida, *parseContext* faz uma chamada ao método *createContext* para que seja criado o objeto *contextObj*, representando o contexto no modelo de destino. Como o elemento *context* declarado no perfil substitui o elemento *context* declarado no módulo *Context*, as chamadas aos métodos *preCompileContext* e *createContext* serão repassadas aos métodos homônimos da classe *CFNclContextCompiler*, conforme ilustrado na Figura 10.

Para validar a chamada ao método *create*, o objeto retornado é testado. Caso o conteúdo do objeto seja nulo, um erro de criação é registrado, através do método *registerCreationError* da classe *CFDocumentCompiler*, cuja instância é denominada *document*. Se um erro de criação for detectado, o valor nulo é retornado pelo método *parseContext*, que é imediatamente interrompido.

O próximo passo de *parseContext*, caso não ocorram erros na criação de *contextObj*, é analisar o conteúdo do elemento XML recebido como parâmetro, ou seja, todos os seus elementos filhos. No exemplo, de acordo com o tipo de cada elemento filho, o método *parseContext* chama um dos métodos *parse*. Esses métodos deverão retornar representações, no modelo de dados de destino, dos elementos contidos em *context*. Cada um desses

objetos é, então, adicionado ao objeto *contextObj* pelas chamadas aos métodos do tipo *add*. No entanto, antes de cada chamada a um método do tipo *add* é verificado se a representação do modelo de dados de destino foi realmente retornada. Caso seja detectado que o modelo de dados de destino possui valor nulo, os métodos do tipo *add* não são chamados. Nesse caso, um erro de adição é registrado, através do método *registerAddElementError*.

Retornando à questão da substituição, em XML *Schema*, os elementos da linguagem que contêm um elemento substituído passam a poder conter também o elemento que o substituiu (substituto). Dessa forma, no exemplo, o elemento *context* pode conter, além dos elementos *context* e *media* originais definidos nos módulos, os respectivos substitutos definidos no perfil.

```
public Object parseContext (Element contextElement) {
  //pré-compila o elemento context
  contextElement = preCompileContext(contextElement);
  Object contextObj = createContext(contextElement);
  //não conseguiu criar o elemento
  se contextObj é igual a null {
    //registra erro de criação do elemento
    document.registerCreationError(contextElement);
    return null;
  }
  para cada elemento filho de contextElement {
    se childContextElement é do tipo profile:context {
      //elemento filho do tipo context definido no própio perfil
      Object childContextObj =
                                  parseContext(childContextElement);
      //adiciona elemento filho se ele não é nulo
      se childContextObj é diferente de null {
        addProfileContextToContext(childContextObj, contextObj);
      }
    } se childContextElement é do tipo context:context {
      //elemento filho do tipo context definido no módulo Context
      Object childContextObj =
        getContextCompiler().parseContext(childContextElement);
      //adiciona elemento filho se ele não é nulo
      se childContextObj é diferente de null {
      addContextContextToContext(childContextObj, contextObj);
      }
    } se childContextElement é do tipo profile:media {
      //elemento filho do tipo media declarado no próprio perfil
      Object childContextObj =
                                  parseMedia(childContextElement);
      //adiciona elemento filho se ele não é nulo
      se childContextObj é diferente de null {
        addProfileMediaToContext(childContextObj, contextObj);
      }
    } se childContextElement é do tipo media:media {
      //elemento filho do tipo media declarado no módulo Media
      Object childContextObj =
              getMediaCompiler().parseMedia(childContextObj);
      //adiciona elemento filho se ele não é nulo
      se childContextObj é diferente de null {
        addMediaMediaToContext(childContextObj, contexObj);
      }
    } se não
      //elemento filho de context não reconhecido
        handleUnknownElementFromContext
                                  (childContextElement, contextObj);
    }//fim do bloco "se"
    //se childContextObj for nulo registra o erro
    se (childContextObj é igual a null) e (childContextElement é
          um elemento filho de context (media ou context) ) {
      document.registerAddElementError(childContextObj);
    }
  }// fim do bloco "para cada"
  contextObj = posCompileContext(contextObj);
  return contextObj; //retornar contextObj
}
```

Figura 9. Exemplo de código do método *parseContext*, automaticamente construído na classe *CFNclProfileCompiler*.

```
public Object preCompileContext (contextElement) {
  return getContextCompiler().
                              preCompileContext(contextElement);
}

public Object createContext(contextElement) {
   return getContextCompiler().createContext(contextElement);
}
```

Figura 10. Exemplo de código do métodos *preCompileContext e createContext*, automaticamente construídos na classe *CFNclProfileCompiler*.

No caso em que um elemento filho de *context* é de um tipo que não esteja previsto no XML *Schema* da linguagem, é chamado o método *handleUnknownElementFromContext*. Quando esse método é chamado, um erro é registrado, através do método *registerUnknowElement*, conforme apresentado na Figura 11. Apesar desse comportamento padrão, processadores específicos podem sobrescrever esse método para tratar extensões da linguagem que não devam ser consideradas como erros.

```
public void handleUnknownElementFromContext (Element
                              element, Element parentElement) {
  document.registerUnknownElement(element);
}
```

Figura 11. Código do método *handleUnknownElement* gerado automaticamente para *CFNclProfileCompiler*.

Por fim, o método *parseContext* efetua uma chamada ao método concreto *posCompileContext*, que pode alterar o objeto *contextObj* antes de retorná-lo.

4. EXEMPLO DE INSTANCIAÇÃO DE UM FRAMEWORK DE PROCESSADORES

De forma semelhante à geração dos processadores genéricos, apresentada na seção anterior, o *framework* de processadores, a partir da classe *CFGenerator*, também pode gerar esqueletos para processadores específicos de uma linguagem, utilizando o nome do modelo de destino do compilador para criação das classes.

Seguindo com o exemplo da seção anterior, onde foi apresentada a construção automática de um processador genérico para a linguagem NCL em um perfil simples, o processador específico a ser construído terá como objetivo receber um documento NCL desse perfil, apresentado na Figura 4, e construir uma estrutura de dados apropriada para apresentação em um terminal de acesso de um sistema de TV digital.

Na seção anterior, a partir do XML *Schema* do perfil simples da linguagem NCL foram construídos arquivos com a especificação das classes *CFNclMediaCompiler*, *CFNclContextCompiler* e *CFNclProfileCompiler*. Sendo definido o modelo de destino do compilador, para cada uma dessas classes serão geradas subclasses que declaram métodos concretos para cada método abstrato definido em suas superclasses. Considerando que o nome dado para o modelo de destino seja "TV", serão geradas as classes *CFNclTVMediaCompiler* *CFNclTVContextCompiler* e *CFNclTVProfileCompiler*. Complementarmente, a classe *CFNclTVDocumentCompiler* será construída como uma subclasse de *CFNclDocumentCompiler*, implementando o método *createSchemaCompilers*, apresentado na Figura 12.

```
public void createSchemaCompilers() {
  profileCompiler = new CFNclTVProfileCompiler();
  mediaCompiler = new CFNclTVMediaCompiler();
  contextCompiler = new CFNclTVContextCompiler();
}
```

Figura 12. Código do método *createSchemaCompilers* gerado automaticamente para *CFNclTVDocumentCompiler*.

O método *createSchemaCompilers* irá conter em seu corpo os comandos para instanciar os três processadores do perfil da linguagem. Sendo assim, ao programador do processador restará a tarefa de preencher o corpo dos métodos para criar objetos de mídia e contextos no modelo de destino, assim como programar de que forma deve ser tratada no modelo de destino a inserção de mídias e contextos em um outro contexto. A Figura 13, apresenta o diagrama de classes completo do processador gerado. Os métodos descritos nessa figura são aqueles cuja especificação do conteúdo é responsabilidade do programador.

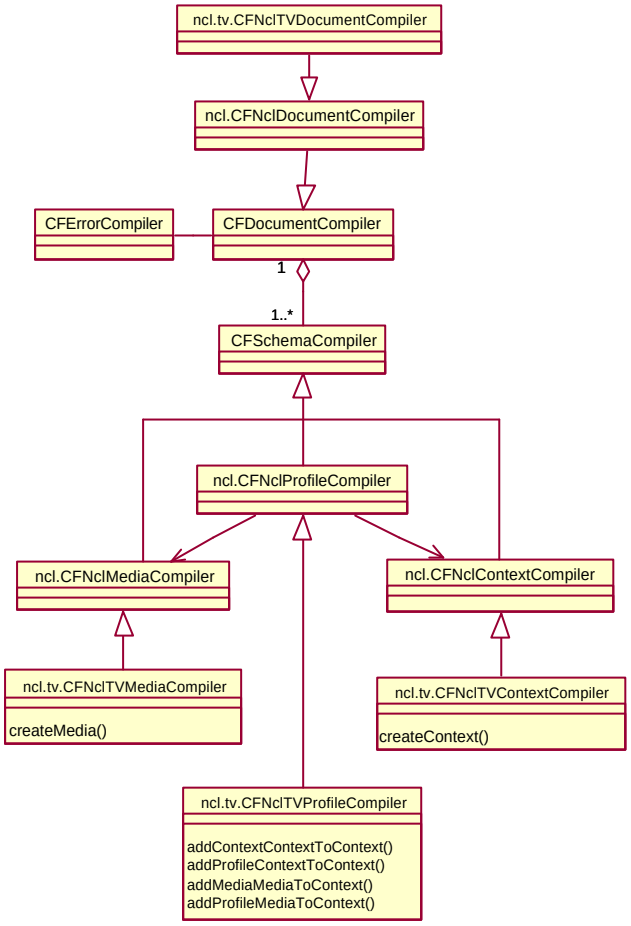

Figura 13. Diagrama das classes geradas automaticamente para o processador do perfil simples de NCL para o modelo de apresentação de programas de TV digital interativa.

Tomando como exemplo o documento NCL apresentado na Figura 14 e considerando que esse documento seja processado por uma instância de *CFNclTVDocumentCompiler*, teríamos a seguinte seqüência de ações.

```
<context id="clipeCoisadePele"
   xmlns = "http://www.telemidia.puc-rio.br/NCL/Profile"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation =
  "http://www.telemidia.puc-rio.br/NCL/Profile
   http://www.telemidia.puc-rio.br/NCL/Profile/SimpleProfile.xsd"
>
   <media type="video" id="clipe" src="coisadepele.mp3"/>
   <media type="text" id="titulo" src="titulo.html"/>
   <context id="letra">
      <media type="text" id="parte01" src="parte01.html"/>
      <media type="text" id="parte02" src="parte02.html"/>
   </context>
</context>
```

Figura 14. Exemplo de documento NCL no perfil simples.

Como comentado na Seção 3, o início do processamento acontece com a chamada ao método *compile* do processador, herdado da classe *CFDocumentCompiler*, passando como parâmetro a URI que localiza o documento da Figura 14. Essa ação resulta em uma chamada ao método *parseRootElement*, implementado na classe *CFNclDocumentCompiler*. Esse método irá identificar o elemento raiz do documento como sendo um contexto (*clipeCoisaDePele*) e irá chamar o método *parseContext* implementado em *CFNclProfileCompiler* (Figura 9). No processamento do contexto, a chamada ao método *createContext* de *CFNclProfileCompiler* irá resultar em uma chamada ao método *createContext* de *CFNclTVContextCompiler*, por conta da substituição de elementos. Nesse momento, o código inserido pelo programador fará com que seja retornada uma implementação do contexto NCL no modelo de destino.

O método *parseContext* passará então a percorrer os elementos do contexto. O primeiro elemento encontrado será a mídia *clipe*, que resultará em uma chamada ao método *parseMedia* implementado em *CFNclProfileCompiler*. Analogamente a *parseContext*, *parseMedia* irá retornar o objeto criado na implementação do método *createMedia* da classe *CFNclTVMediaCompiler*, cuja construção, de forma similar ao método *createContext*, é responsabilidade do programador do processador. Tal objeto será então passado como parâmetro ao método *addProfileMediaToContext*, juntamente com o objeto resultante do processamento do contexto *clipeCoisaDePele*, para que o programador defina como tal relação de inclusão deve ser processada.

O passo seguinte será o processamento da mídia *titulo*, cuja seqüência de operações será idêntica às do processamento do vídeo *clipe*. Ao encontrar para processamento o contexto *letra*, o método *parseContext* fará uma chamada recursiva para que o conjunto de objetos HTML, especificados através dos elementos *media*, sejam tratados.

O *framework* proposto neste artigo, além de automaticamente tratar erros originados por divergências no documento XML de origem em relação ao XML *Schema* da linguagem, como apresentado na seção anterior, também facilita o tratamento de erros do documento XML em relação ao modelo de dados de destino. No exemplo da Figura 14, os objetos de mídia do documento NCL são objetos de vídeo e de texto. A linguagem NCL não impõe qualquer restrição em relação aos tipos dos objetos de mídia especificados, conforme pode ser observado no XML *Schema* do módulo *Media* (Figura 3), onde o atributo *type* do elemento *media* é do tipo *string*. No entanto, a ferramenta utilizada para apresentação dos documentos, usualmente, é capaz de exibir um conjunto limitado de tipos de mídia. Nesse caso, para que os documentos sejam apresentados, é o modelo de dados da ferramenta de apresentação que deve determinar os valores válidos para o atributo *type* do elemento *media*.

Considerando que o modelo de destino reconhece os tipos "*text*", "*img*", "*video*" e "*audio*", na implementação do método *createMedia*, o programador deverá realizar a validação do documento em relação ao modelo de dados dessa ferramenta. Para os casos em que o tipo de mídia não é reconhecido, o tratamento de erros do processador pode ser utilizado, conforme apresentado no pseudo-código da Figura 15.

```
…
    se element é do tipo "text" {  …
    } senão se element é do tipo "img" { …
    } senão se element é do tipo "video" { …
    } senão se element é do tipo "audio" {  …
    } senão {
          document.registerError(element, "Tipo do elemento de
    mídia não reconhecido pela ferramenta de apresentação …")
    }
…
```

Figura 15. Tratamento de erros no método *createMedia*.

É importante observar que, embora o exemplo apresentado seja simples e um mapeamento da linguagem para um conjunto de objetos Java também pudesse ser obtido utilizando *data binding* (Seção 2), a proposta descrita neste artigo objetiva ser mais abrangente e flexível no processamento de documentos baseados na linguagem XML. O processador NCL para TV exemplificado nesta seção poderia, por exemplo, gerar uma estrutura binária para transmissão do documento através de um carrossel de objetos, seguindo o padrão DSM-CC [4]. O código dos métodos *create* e *add* poderia construir a estrutura de transmissão diretamente, sem necessariamente passar por uma implementação Java fiel ao XML *Schema* da linguagem de origem.

5. CONSIDERAÇÕES FINAIS

O principal objetivo do trabalho apresentado neste artigo é simplificar o desenvolvimento e a manutenção de processadores de documentos XML, através de um mecanismo semi-automático para concepção de tais processadores. Um gerador interpreta como entrada um XML *Schema*, descrevendo uma linguagem, e gera como saída a estrutura de classes (um *framework*) de processadores para essa linguagem, definindo a sintaxe e semântica dos métodos das várias classes. A abordagem foi baseada no padrão de projeto *Template Method* [2], que define um esqueleto para os algoritmos de cada classe (os métodos concretos) e delega certos passos às subclasses (através da definição de métodos abstratos) que irão compor uma instância de processador da linguagem.

A proposta foi implementada utilizando a linguagem de programação Java e permitiu a construção de processadores para algumas linguagens de autoria multimídia/hipermídia: NCL, SMIL e XMT-O. Um dos principais objetivos das instâncias geradas é fornecer uma interoperabilidade entre linguagens de autoria multimídia/hipermídia, favorecendo cenários atuais, como o da geração de conteúdo para os sistemas de transmissão de TV digital interativa.

Para a leitura dos documentos de entrada, foi utilizada uma estrutura baseada em DOM. Como trabalho futuro, pretende-se estudar a viabilidade de utilizar *data binding* tanto na geração dos processadores genéricos como na leitura dos arquivos. A idéia é gerar automaticamente classes que herdem das classes geradas por ferramentas como JAXB e construir o processador genérico como um arcabouço de código que navegue por essas classes de acordo com a estrutura do documento. Métodos abstratos nessas subclasses funcionariam como pontos de flexibilização do *framework*.

6. Agradecimentos

Os autores gostariam de agradecer as contribuições de Heron Vilela de Oliveira e Silva, pelas pesquisas relacionadas ao desenvolvimento de processadores XML e Livia Fonseca Fracalanza, pela implementação do tratamento de erros no processador da linguagem NCL para criação de objetos Java do modelo NCM.

7. REFERÊNCIAS

[1] Boukottaya, A., Vanoirbeek, C., Paganelli, F., Abou Khaled, O. Automating XML Documents Transformations: A Conceptual Modelling based Approach. *ACM International Conference Proceeding Series*, v. 59, (2004), 81-90.

[2] Gama, E., Helm, R., Jonhson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1º Edição (Janeiro 1995).

[3] ETSI European Telecommunications Standard Institute. *DVB Specification for Data Broadcasting – ETSI EN 301 192*, (Junho 2004).

[4] ISO/IEC International Organization for Standardization. 8879:1986, *Information processing - Text and office systems - Standard Generalized Markup Language (SGML)*, (1986).

[5] ISO/IEC International Organization for Standardization. *13818-6:1998, Generic Coding of Moving Pictures and Associated Audio Information – Part 6: Extensions for DSM-CC*, (1998).

[6] ISO/IEC International Organization for Standardization. *14496-1:2004, Coding of Audio-Visual Objects – Part 1: Systems*, 3º Edição, (2004).

[7] JAXB, *Java Architecture for XML Binding – JAXB 2.0 Project*, disponível em http://jaxb.dev.java.net, acesso em Junho 2006.

[8] Kurtev, I., Berg, K. Building Adaptable and Reusable XML Applications with Model Transformations. *14º International World Wide Web Conference*, (2005), 160-169.

[9] Li, Q., Kim, M.Y., So, E., Wood, S. XVM: A Bridge between XML Data and Its Behavior. *13º International World Wide Web Conference*, (2004), 155-163.

[10] Reinhold, M. An XML Data-Binding Facility for the Java Platform. *Sun Microsystems White Paper*, (Julho, 1999).

[11] SAX, *Simple API for XML – SAX 2.0 Project*, disponível em http://sax.sourceforge.net, acesso em Junho 2006.

[12] Shin, D.H., Lee, K.H. Generating XSLT Scripts for the Fast Transformation of XML Documents. *14º International World Wide Web Conference*, (2005), 1098-1099.

[13] Silva, H.V.O., Muchaluat-Saade, D.C., Rodrigues, R.F., Soares, L.F.G. NCL 2.0 Integrating New Concepts to XML Modular Languages. *ACM Symposium on Document Engineering 2004*, (Outubro 2004), 188-197.

[14] W3C World Wide Web Consortium, *Extensible Stylesheet Language Transformations – XSLT 1.0*, (Novembro 1999).

[15] W3C World-Wide Web Consortium, *Extensible HyperText Markup Language – XHTML 1.0*, 2º Edição, (Agosto 2002).

[16] W3C World Wide Web Consortium, *Extensible Markup Language – XML 1.0*, 3º Edição, (Fevereiro 2004).

[17] W3C World Wide Web Consortium, *Namespaces in XML 1.1*, (Fevereiro 2004).

[18] W3C World-Wide Web Consortium, *Document Object Model – DOM Level 3 Specification*, (Abril 2004).

[19] W3C World-Wide Web Consortium, *Extensible Markup Language Schema – XML Schema Part 1: Structures*, 2º Edição (Outubro 2004).

[20] W3C World-Wide Web Consortium, *Synchronized Multimedia Integration Language – SMIL 2.1 Specification*, (Dezembro 2005).