templates de composição que usam conectores para especificar essas semânticas. Por esse motivo, em X-SMIL, é desencorajado o uso das composições *par*, *seq* e *excl*, sendo sugerido o uso do elemento *composite* (adicionado à X-SMIL utilizando o módulo *BasicComposite* de NCL), que deve ser usado no lugar dessas composições. Para obter a semântica temporal oferecida anteriormente pelas composições SMIL, deve-se utilizar os *templates* pré-definidos por X-SMIL: *templatePar*, *templateSeq*, *templateExcl*[34]. Um exemplo de uma composição X-SMIL, similar à composição da Figura 4:7, é ilustrado na Figura 4:9.

Em X-SMIL, assim com em NCL 2.1, pode-se definir *templates* de composição utilizando todas as facilidades de XTemplate 2.1. Ou seja, é possível definir tanto relações de inclusão (como exemplificado na seção anterior) quanto relações por meio de conectores (como exemplificado no Capítulo 3). Compiladores para documentos nessas linguagens, assim como o processador de *templates* de XTemplate 2.1, serão descritos no próximo capítulo.

```
01. <composition xtemplate="templatePar" >
02.   <video id="video1" />
03.   <audio id="audio1" />
04.   <img id="img1 />
05.   <linkBase>
06.     <link id="link2" xconnector="finishes.xml">
07.       <bind component="audio1" role="on_x_presentation_end"/>
08.       <bind component="img1" role="stop_y"/>
09.     </link>
10.   </linkBase>
11. </composition>
```

Figura 4:9 - Exemplo de uma composição X-SMIL.

[34] Para refletir o uso do atributo *endsync* (W3C, 2001b), definido em SMIL para composições *par* e *excl*, também foram especificados os *templates templateParFirst*, *templateParLast*, *templateParAll*, *templateExclFirst*, *templateExclLast* e *templateExclAll*, que correspondem, respectivamente, aos valores *first*, *last* e *all* que esse atributo pode possuir. Assim, pela semântica definida por SMIL, os *templates templatePar* e *templateExcl* equivalem, respectivamente, aos t*emplates templateParLast* e *templateExclLast*.

5
Framework para Compiladores

Um dos princípios adotados pelo W3C para a criação de linguagens de marcação baseadas em XML sugere uma abordagem modular. Módulos agrupam, de forma coerente, elementos e atributos XML que possuam alguma relação semântica entre si. SMIL (W3C, 2001b) e XHTML (W3C, 2000c) são exemplos de linguagens W3C que seguem esse princípio. As vantagens obtidas pelo uso de uma abordagem modular são possíveis devido às características de XML e de tecnologias relacionadas (W3C, 2001d; W3C, 1999b).

Documentos especificados de acordo com a definição de uma linguagem modular baseada em XML devem ser processados para que suas descrições possam ser utilizadas em um determinado aplicativo. Muitas vezes, diversos processadores de documentos são implementados, por aplicativos diferentes, para uma mesma linguagem. Por exemplo, há um processador de documentos SMIL para cada *player* SMIL.

No entanto, quando processadores analisam documentos de uma mesma linguagem de origem, várias funcionalidades podem ser reaproveitadas e implementadas uma única vez. A identificação dessas funcionalidades deu início ao desenvolvimento do trabalho apresentado neste capítulo. Além das características comuns aos processadores de uma mesma linguagem, observou-se, em outro nível de abstração, aspectos comuns que podem ser reutilizados em processadores de qualquer linguagem modular baseada em XML. A partir dessas observações, uma estrutura em dois níveis para especificação de *frameworks* para implementação de processadores (ou compiladores) é proposta, como se segue.

Em um primeiro nível, existe o *framework genérico de processamento* para aplicações XML. O intuito desse *framework* é reunir as características comuns para construção de compiladores genéricos independentes das linguagens. Por servir como um *framework* para a construção de outros *frameworks*, o *framework* genérico de processamento é também designado *meta-framework*.

Assim, escolhida a linguagem de origem, tem-se um *framework* para compiladores dessa linguagem, gerado a partir do *meta-framework*. Tendo por base esse *framework* de compiladores, em um segundo nível, compiladores específicos podem ser instanciados (como, por exemplo, compiladores para *players* SMIL). O objetivo é que cada compilador possa receber um documento descrito segundo a linguagem de origem e seja capaz de gerar uma especificação em um determinado *modelo de dados de destino*, que pode ser o modelo de um aplicativo, ou mesmo de uma outra linguagem declarativa.

O *meta-framework* foi aplicado a algumas linguagens baseadas em XML, resultando na construção dos *frameworks* de compiladores NCL, SMIL e X-SMIL, entre outros. A partir desses *frameworks*, foram desenvolvidos compiladores para conversão NCL em descrições SMIL e X-SMIL; para conversão SMIL em descrições NCL e X-SMIL; para conversão X-SMIL em descrições NCL e SMIL; para conversão de NCL, SMIL e X-SMIL para uma implementação de objetos do modelo NCM (Soares et al., 2003); e para uma implementação de objetos do modelo de execução do formatador HyperProp (Rodrigues et al., 2003; Rodrigues, 2003) - tornando-o, assim, capaz de exibir documentos dessas linguagens.

Além da compilação entre linguagens e modelos, o *framework* também foi utilizado na adaptação de documentos NCL, SMIL e X-SMIL. Essas linguagens oferecem suporte para ajustar a especificação dos documentos em função de parâmetros do contexto de exibição (ver Capítulo 1), tais como, preferências do usuário, características da plataforma etc. Quando tais parâmetros podem ser conhecidos antes de iniciar a apresentação do documento, a escolha da melhor alternativa de ajuste pode ser realizada por uma ferramenta de análise, que consulta os valores dos parâmetros e modifica a descrição do documento. A adaptação pode, assim, ser vista como a compilação de um documento com alternativas de conteúdos e exibições, para outro documento na mesma linguagem, agora com algumas (ou mesmo todas) alternativas resolvidas.

Este capítulo está organizado da seguinte forma. A primeira seção descreve o *framework* genérico (*meta-framework*) de processamento para linguagens modulares definidas por meio de XML. Nas seções seguintes, detalha-se o *framework* para o processamento de documentos NCL, SMIL e X-SMIL. Cada

uma dessas seções também discute os aspectos de implementação das instâncias desses *frameworks*.

5.1. *Framework* Genérico para Processamento

Diversas linguagens baseadas em XML apresentam uma grande complexidade. Essa complexidade reflete-se, normalmente, em um grande número de elementos e atributos. Com o objetivo de melhorar a estruturação e obter as vantagens mencionadas no Capítulo 2, várias aplicações XML definem módulos para agrupar elementos e atributos que possuam semântica relacionada. Como o número de módulos pode ser grande, algumas linguagens (como NCL e SMIL) definem um segundo nível de estruturação, agrupando seus módulos em *áreas funcionais*.

A complexidade de linguagens modulares também pode ser refletida na interdependência entre seus módulos: elementos de um módulo podem conter elementos pertencentes a outros módulos. Analogamente, atributos de elementos de um módulo são, muitas vezes, definidos em outros módulos.

Observando essa complexidade, este trabalho propõe um *framework* genérico de processamento (ou *meta-framework*), cujo principal objetivo é facilitar o desenvolvimento de *frameworks* de compiladores para aplicações XML modulares. O *meta-framework* define sintática e semanticamente os métodos de *frameworks* de compiladores, além de estabelecer a estruturação das classes desses *frameworks*.

A principal entrada do *meta-framework* é a definição de uma linguagem em XML Schema (W3C, 2001d). Por simplicidade, neste texto, essa entrada é representada utilizando as seguintes tabelas: a *tabela de elementos* e a *tabela de grupos de elementos*. Para linguagens que definem áreas funcionais, uma terceira tabela, que não pode ser obtida pela análise do XML Schema da linguagem, também é necessária: a *tabela de áreas funcionais*.

A tabela de elementos lista todos os elementos da linguagem, conforme exemplificado na Tabela 1. Cada elemento é identificado pelo seu nome (como *composite*) e pelo seu módulo (como *BasicComposite*), estando listados na primeira coluna. A segunda coluna da tabela descreve o conteúdo dos elementos.

Esse conteúdo é constituído de elementos atômicos (como *linkBase*) ou grupos de elementos (como *mediaContentGroup*). Elementos atômicos e grupos de elementos são listados da mesma forma, possuindo um nome e o módulo ao qual pertencem. A Tabela 1 apresenta uma definição parcial do elemento *composition* de NCL, que será utilizado como exemplo neste capítulo.

Elemento	**Conteúdo**
composite / BasicComposite	*linkBase / Linking* *mediaContentGroup / BasicMedia*

Tabela 7 - Tabela de Elementos.

A segunda tabela de entrada para o *meta-framework* lista todos os grupos de elementos da linguagem, conforme exemplificado na Tabela 8. A primeira coluna identifica o grupo de elementos, por seu nome e módulo, enquanto a segunda coluna lista o conteúdo de cada grupo (outros elementos da linguagem).

Grupo	**Conteúdo**
mediaContentGroup / BasicMedia	*text / BasicMedia* *img / BasicMedia* *audio / BasicMedia* *animation / BasicMedia* *video / BasicMedia* *textstream / BasicMedia*

Tabela 8 - Tabela de Grupos de Elementos.

A terceira e última tabela, chamada tabela de áreas funcionais, é necessária apenas para as linguagens que agrupam seus módulos em áreas funcionais, estando exemplificada na Tabela 9. A primeira coluna identifica o nome da área funcional, enquanto a segunda coluna lista os nomes dos módulos que compõem cada área funcional, devendo esses nomes ser únicos na linguagem.

Área Funcional	**Módulos**
Components	*BasicMedia*; *BasicComposite*
Linking	*Linking*

Tabela 9 - Tabela de Áreas Funcionais.

A abordagem adotada pelo *meta-framework* para construção dos *frameworks* de compiladores foi a seguinte: cada *framework* de compiladores implementa determinados métodos concretos e declara outros métodos abstratos para serem tratados pelas suas instâncias (os compiladores propriamente ditos),

dividindo-se assim as responsabilidades entre o *framework* e suas instâncias. Essa divisão segue o padrão de projeto *Template Method* (Gamma et al., 1995), 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). A definição da nomenclatura dos métodos e das especificações do corpo dos métodos concretos são, assim, de responsabilidade do *meta-framework*. A Figura 5:1 ilustra a abordagem proposta.

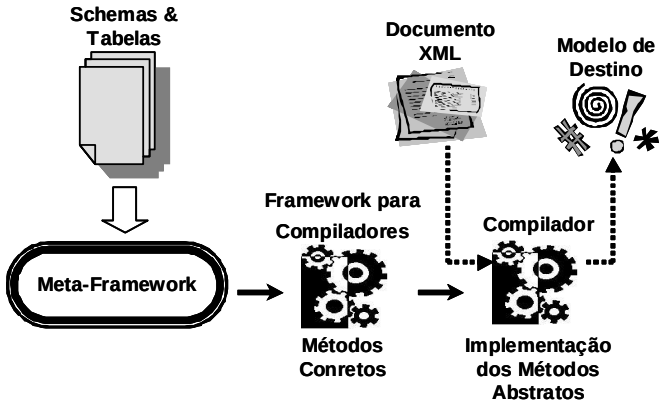

Figura 5:1 - . Visão geral da estruturação em dois níveis dos *frameworks* para compiladores de linguagens modulares.

A nomenclatura definida, pelo *meta-framework*, para os métodos dos *frameworks* de compiladores é a seguinte:

- métodos iniciados pelo nome *parse*, como *parseComposite*, são responsáveis por receber um elemento XML (elemento do tipo *composite*, por exemplo) como parâmetro, analisá-lo e retornar um objeto do modelo de destino representando esse elemento. Para a criação desse objeto, o método realiza chamadas a outros métodos do *framework* (concretos ou abstratos). Deve existir um método concreto do tipo *parse* para cada elemento e para cada grupo de elementos, listados na tabela de elementos e na tabela de grupos de elementos da linguagem de entrada, respectivamente. Os métodos *parse* fornecem a principal funcionalidade a ser herdada pelos compiladores, pois implementam o caminhamento pela estrutura dos documentos;
- métodos iniciados pelo nome *create*, como *createComposite*, são responsáveis por receber um elemento XML (*composite*) e criar o objeto representando o elemento no modelo de destino. Esses métodos são sempre

abstratos e são chamados pelos métodos (concretos) do tipo *parse*, devendo existir um método *create* para cada método *parse* criado;

- 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, *addMediaContentGroupToComposite* recebe duas referências, uma para o objeto representando o elemento *composite* e outra para o objeto representando o elemento do grupo *mediaContentGroup* no modelo de destino; e adiciona o objeto representando o elemento *mediaContentGroup* ao objeto representando o elemento *composite*. Esses métodos são sempre abstratos e chamados pelos métodos *parse*. Para cada elemento da linguagem, deve existir um conjunto de métodos *add*; um para cada elemento (ou grupo) de seu conteúdo (entradas da segunda coluna na tabela de elementos).
- métodos iniciados pelo nome *preCompile* e *posCompile*, como *preCompileComposite* e *posCompileComposite*, são, respectivamente, o primeiro e último método chamado pelos métodos do tipo *parse*. Esses métodos concretos são declarados pelos *frameworks* de compiladores para possibilitar (mas não obrigar) que compiladores os re-implementem a fim de obter uma funcionalidade específica, como será visto. Devem existir métodos desses tipos para cada método do tipo *parse*.
- métodos concretos iniciados pelo nome *handleElementIn*, como *handleElementInComposite*, são chamados pelos métodos *parse* quando um elemento filho não é reconhecido como conteúdo de outro elemento. Compiladores (ou outros *frameworks* de compiladores) podem re-implementar esses métodos para tratar elementos não conhecidos previamente pelo *framework* de compiladores, que, dessa forma, oferece um mecanismo de extensão[35]. Deve ser implementado um método desse tipo para cada método do tipo *parse*.

Pode-se exemplificar a sintaxe e a semântica propostas para os métodos dos *frameworks* de compiladores a partir do elemento *composite* da Tabela 7 (que representa parcialmente o elemento *composite* de NCL). Um *framework* de

[35] Um exemplo do uso desse método será visto no *framework* para compiladores X-SMIL.

compiladores, gerado a partir dessa tabela[36], deve implementar o método concreto *parseComposite*. Esse método, representado em pseudo-código na Figura 5:2, recebe um elemento XML do tipo *composite* (*compositeElement*) como parâmetro e, inicialmente, efetua uma chamada ao método concreto *preCompileComposite*. Esse método pode alterar o *compositeElement* ao retornar um elemento do tipo *composite* pré-processado. Em seguida, *parseComposite* faz uma chamada ao método abstrato *createComposite* para que seja criado o objeto *objComposite*, representando o elemento XML do tipo *composite* no modelo de destino.

```
01  public Object parseComposite(compositeElement) {
02    //pré-compilar elemento composite
03    compositeElement = preCompileComposite(compositeElement);
04    objComposite = createComposite(compositeElement);
05    para cada elemento childCompositeElement filho de compositeElement {
06      se childCompositeElement do tipo LinkBase {
07        //elemento filho do tipo linkBase, criar linkBase
08        childCompositeObject = parseLinkBase(childCompositeElement);
09        //adicionar linkBase à composite
10        addLinkBaseToComposite(objComposite, childCompositeObject);
11      } se childCompositeElement do tipo mediaContentGroup {
12        //elemento filho do tipo mediaContentGroup, criar mediaContentGroup
13        childCompositeObject
                    = parseMediaContentGroup (childCompositeElement);
14        //adicionar mediaContentGroup à composite
15        addMediaContentGroupToComposite
                            (objComposite, childCompositeObject);
16      }se não{
17        //elemento filho de composite não reconhecido.
18        //criar e adicionar elemento à composite
19        handleElementInComposite
                                (objComposite, childCompositeElement);
20      } //fim do bloco "se"
21    } //fim do bloco "para cada"
22    //pós-compilar objComposite
23    objComposite = posCompileComposite(objComposite);
24    //retornar objComposite
25    return objComposite;
26  }
27
28
29  public Element preCompileComposite(compositeElement) {
30    return compositeElement;
```

[36] Cabe observar que, na realidade, o *framework* é gerado a partir do XML Schema da linguagem. Conforme comentado anteriormente, as tabelas de elementos e de grupo de elementos foram utilizadas apenas para facilitar a descrição do processamento.

```
31  }
32
33  public Object posCompileComposite(objComposite) {
34    return objComposite;
35  }
36
37  public void handleElementInComposite
                          (objComposite, childCompositeElement) {
38    //método vazio
39  }
```

Figura 5:2 - Exemplo de um método do tipo *parse* de um *framework* de compiladores.

O próximo passo de *parseComposite* é analisar o conteúdo do elemento XML recebido como parâmetro, ou seja, todos os seus elementos filhos (listados na segunda coluna da tabela de elementos). De acordo com o tipo de cada elemento filho, o método *parseComposite* chama um dos métodos *parse* a seguir: *parseLinkBase* ou *parseMediaContentGroup*. Esses métodos deverão retornar representações dos elementos contidos em *composite* no modelo de destino. Cada um desses objetos é, então, adicionado ao objeto *objComposite* pelos métodos *addLinkBaseToComposite* e *addMediaContentGroupToComposite*, respectivamente. Se o elemento filho do elemento *composite* não estiver listado na tabela de elementos[37], o método *parse* chama o método concreto *handleElementInComposite*, responsável por compilar e adicionar esse elemento a *objComposite*.

Finalmente, o método *parseComposite* efetua uma chamada ao método concreto *posCompileComposite*, que pode alterar o objeto *objComposite* que será, em seguida, retornado pela função *parse*.

Os métodos concretos *preCompileComposite*, *posCompileComposite* e *handleElementInComposite* também estão apresentados na Figura 5:2. O *framework* de compiladores declara esses métodos como concretos para facilitar a implementação de compiladores que não necessitem utilizá-los. O comportamento padrão do primeiro método é retornar o elemento XML recebido como parâmetro, enquanto o segundo retorna o objeto recebido e, finalmente, o terceiro não executa qualquer código.

Além da definição da assinatura dos métodos e das especificações do corpo dos métodos concretos, o *meta-framework* também determina uma estrutura de

classes a ser seguida por cada *framework* de compiladores. Em primeiro lugar, cada um desses *frameworks* deve implementar uma subclasse da classe gerenciadora *CFDocumentParser*[38]. Essa subclasse[39] deve conter métodos para leitura e compilação de arquivos XML especificados na linguagem de entrada, além de referências para todas as outras classes do *framework* de compiladores. Caso a linguagem para a qual se pretende construir um *framework* de compiladores seja organizada em áreas funcionais, propõe-se que seja criada uma classe representando cada área funcional; caso contrário, deve ser criada uma classe para cada módulo da linguagem[40]. Essas classes devem herdar da classe abstrata *CFModuleParser*, que define uma referência para a classe gerenciadora *CFDocumentParser*. Como módulos (ou áreas funcionais) dependem de outros módulos (ou áreas funcionais), a classe gerenciadora também deve realizar as chamadas para atribuir as referências entre as classes, como será visto.

Outra recomendação do *meta-framework* é a criação de *frameworks* para compiladores refletindo o perfil completo de uma linguagem. Dessa forma, outros perfis da linguagem, sempre mais restritos que o perfil completo, podem utilizar o mesmo *framework*, e os mesmos compiladores, para o processamento de seus documentos. Porém, nada impede a definição de classes gerenciadoras distintas, ou até mesmo *frameworks* de compiladores diferentes (mais simples), para perfis específicos de uma linguagem.

Para geração automática dos métodos e das classes de um *framework* de compiladores, foi desenvolvida uma aplicação baseada no *meta-framework*, chamada *gerador de framework de compiladores*. O diagrama de classes em UML

---

[37] Isso pode ocorrer quando o método *parse* está sendo usado para compilar documentos em outra linguagem (ou perfil) que não a utilizada para gerar o *framework*.

[38] O prefixo *CF* das classes do *framework* significa "*Compiler Framework*" (*framework* de compiladores).

[39] A classe *CFDocumentParser* implementa alguns métodos que são herdados por suas subclasses. Entre esses métodos, encontram-se: um método para geração de uma árvore DOM (W3C, 2000a) a partir de um arquivo XML qualquer; e métodos para criação de tabelas, adição de objetos a essas tabelas, e acesso a objetos nelas contidos. Um exemplo do uso dessas tabelas é apresentado na próxima seção.

[40] Linguagens que não são estruturadas em módulos podem ser consideradas como uma linguagem modular com um único módulo. Assim, *frameworks* de compiladores também podem ser gerados para essas linguagens.

(Rumbaugh et al., 1999) dessa aplicação está ilustrado na Figura 5:3. A classe *CFGenerator* possui métodos tanto para geração de *framework* de compiladores de uma linguagem quanto para geração de esqueletos de código de compiladores para essa linguagem.

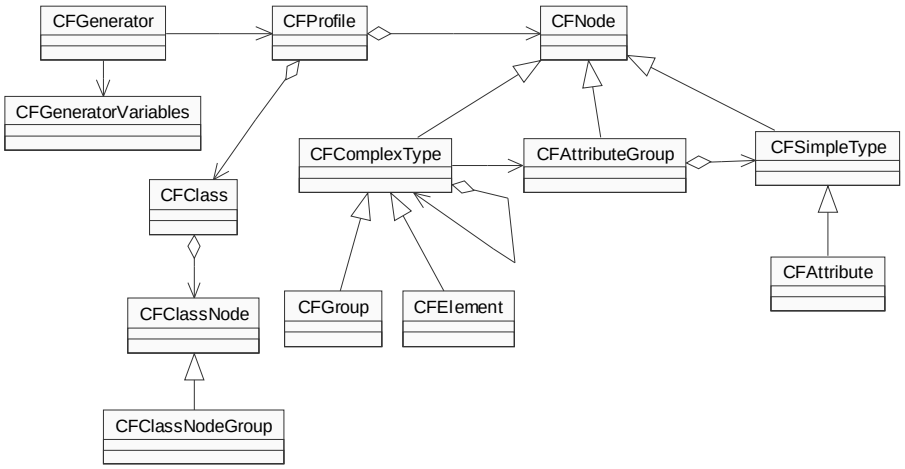

Figura 5:3 - Diagrama de classes do gerador automático de *frameworks* de compiladores.

A geração automática de código necessita de algumas informações, que devem ser parametrizadas por uma instância da classe *CFGeneratorVariables*. Essa classe permite que *CFGenerator* consulte a localização do *schema* que define uma linguagem, e dados que não podem ser obtidos diretamente pela análise desse *schema*, como o nome da linguagem e um mapeamento entre seus módulos e áreas funcionais (informações contidas na Tabela 9).

A partir das informações contidas em *CFGeneratorVariables*, a classe *CFGenerator* constrói um objeto perfil (da classe *CFProfile*) refletindo a especificação de uma determinada linguagem (ou perfil de linguagem). Para isso, a definição de uma linguagem - realizada por meio dos elementos *attribute*, *attributeGroup*, *complexType* , *element*, *group* e *simpleType* de XML Schema (W3C, 2001d) - é usada para geração de objetos das classes *CFAttribute*, *CFAttributeGroup*, *CFComplexType*, *CFElement*, *CFGroup* e *CFSimpleType*, respectivamente[41]. Em um primeiro passo, o perfil (*CFProfile*) gerado reflete

[41] Existem outros elementos de XML Schema, além dos citados, que podem ser usados na especificação de linguagens. Porém, como esses elementos não definem informações relevantes para a geração de *frameworks*, não foram implementadas classes para representá-los. Esses

exatamente o *schema* da linguagem, porém, como XML Schema possui diversas formas de definição de tipos através de referência a outros tipos, um segundo passo é necessário: computar essas referências. A partir desse perfil (*CFProfile*) com as referências calculadas, têm-se disponíveis as informações representadas pela Tabela 7 e pela Tabela 8, sendo possível, assim, a geração das classes do *framework* de compiladores.

Para cada área funcional (ou módulo) da linguagem, a classe *CFProfile* gera, em um terceiro passo, um objeto da classe *CFClass.* Esse objeto possui objetos das classes *CFClassNode* e *CFClassNodeGroup* referentes, respectivamente, aos elementos e aos grupos de elementos da área funcional (ou módulo) que ele representa. Cada objeto *CFClass* irá gerar uma classe do *framework* de compiladores, cujos métodos são criados a partir de cada um de seus nós (*CFClassNode* e *CFClassNodeGroup*).

Dando continuidade ao exemplo do elemento *composite* da Tabela 7, e supondo que o nome da linguagem seja definido como NCL, o gerador de *frameworks* irá criar (a partir de objetos *CFClass*) uma classe *NclComponentsParser*[42] (representando a área funcional *Components*) e uma classe *NclLinkingParser* (representando a área funcional *Linking*). Essas classes herdam de *CFModuleParser* e terão seus métodos do tipo *parse*, *create*, *add*, *preCompile*, *posCompile* e *handleElementIn* definidos para cada um dos seus *nós de classe* (objetos das classes *CFClassNode* e *CFClassNodeGroup*). Por exemplo, na classe *NclComponentsParser*, teremos os métodos *parseComposite*, *createComposite*, *parseMediaContentGroup*, *addLinkBaseToComposite*, *addMediaContentGroupToComposite* etc. Já na classe *NclLinkingParser*, serão gerados os métodos *parseLinkBase* e *createLinkBase*, entre outros. Além das classes representando áreas funcionais, será gerada, também, a classe gerenciadora *NclDocumentCompiler*[43], herdando de *CFDocumentParser*.

---

elementos são utilizados durante a análise do *schema* da linguagem para geração dos objetos das classes definidas pelo *meta-framework.*

[42] O nome das classes geradas é obtido pela concatenação do nome da linguagem com o nome da área funcional, e com a palavra "*parser*".

[43] O nome da classe gerenciadora é obtido pela concatenação do nome da linguagem com a palavra "*DocumentCompiler*".

Como comentado, de forma semelhante à geração de *framework* de compiladores, a classe *CFGenerator* também pode gerar esqueletos para compiladores específicos de uma linguagem, a partir do nome do modelo de destino do compilador. Supondo que o modelo de destino seja o NCM e a linguagem seja NCL, seguindo com o exemplo, podem ser geradas as classes *NclNcmDocumentCompiler*, *NclNcmComponentsParser* e *NclNcmLinkingParser*[44]. Essas classes declaram métodos concretos para cada método abstrato definido por suas superclasses (*NclDocumentCompiler*, *NclComponentsParser* e *NclLinkingParser*). O corpo desses métodos deve ser implementado pelo programador do compilador de documentos NCL para objetos do NCM.

Finalmente, em alguns casos, compiladores específicos para uma determinada linguagem também podem ser gerados automaticamente pelo gerador de *frameworks*. Isso ocorre quando é possível estabelecer um mapeamento genérico entre um documento XML e a estrutura de dados de destino. Um exemplo implementado foi a geração automática de compiladores que produzem simplesmente cópias de documentos de uma determinada linguagem. Apesar de à primeira vista pouco necessário, esse gerador foi extremamente útil na implementação de um adaptador de documentos NCL, como será discutido na Seção 5.2.2. Outro exemplo implementado foi a geração automática de um compilador que constrói uma visualização gráfica em árvore de um documento XML. Java foi a linguagem de programação utilizada na implementação dos *frameworks* e dos compiladores deste trabalho, usando a API JAXP (Sun, 2002).

5.2. *Framework* para Compiladores de Documentos NCL

O *framework* para compiladores NCL é uma instância do *meta-framework* descrito na seção anterior. A Figura 5:4 apresenta o diagrama UML das classes do *framework* gerado automaticamente, onde cada classe representa uma área funcional da linguagem NCL. Existe também uma classe gerenciadora, chamada *NclDocumentCompiler*, que reúne as áreas funcionais e representa o ponto de

[44] A nomenclatura utilizada para as classes dos compiladores é similar àquelas utilizadas para classes do *framework*, adicionando o nome do modelo de destino após o nome da linguagem.

entrada para compilação de documentos que sigam o perfil completo da linguagem. As setas entre classes de áreas funcionais simbolizam relações de dependência entre os seus módulos.

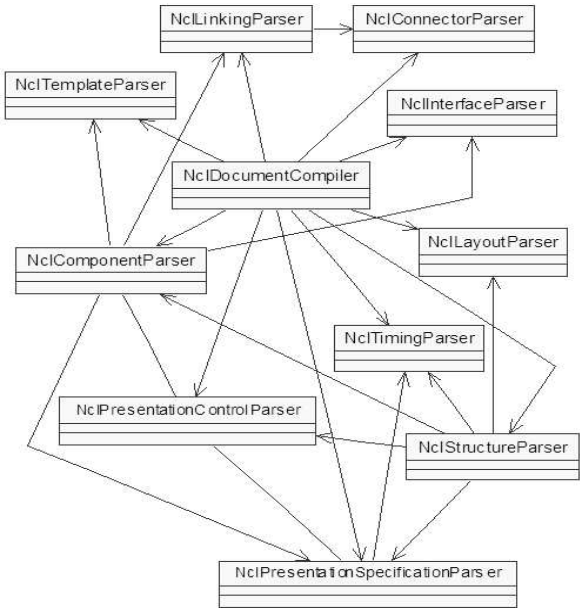

Figura 5:4 - Diagrama de classes do *framework* para compiladores NCL.

5.2.1. Funcionamento do *Framework* de Compiladores NCL

O documento NCL exemplificado na Figura 5:5 servirá de base para ilustrar as principais características do funcionamento do *framework* de compiladores NCL. A compilação de um documento NCL inicia-se quando a classe gerenciadora *NclDocumentCompiler* chama o método *parseNcl* da classe *NclStructureParser*. Esse método analisa o elemento *ncl* e seus elementos filhos: *head* e *body*.

```
<ncl>
 <head>
  <layout>...</layout>
  <descriptorBase>
   <descriptor id="d0" dur="30s" region="audioRegion"/>
   <descriptor id="d1" region="title"/>
   <descriptor id="d2" region="textRegion"/>
   <descriptor id="d3" region="imageRegion" player="quickTime"/>
  </descriptorBase>
 </head>
 <body>
  <composite id="C">
   <composite id="C1">
```

```
82      <port id="portA1" component="audio1" />
83      <audio id="audio1" descriptor="d0" src="">
84       <area id="part1" begin="8.4s" end="18s"/>
85       <area id="part2" begin="18.5s" end="28s"/>
86      </audio>
87      <text id="title1" descriptor="d1" src=""/>
88      <text id="lyr11" descriptor="d2" src=""/>
89      <text id="lyr12" descriptor="d2" src=""/>
90     </composite>
91     <composite id="C2">
92      <port id="portA2" component="audio2" />
93      <audio id="audio2" descriptor="d0" src=""/>
94      <composite id="C21">
95       <text id="title2" descriptor="d1" src=""/>
96       <text id="lyr21" descriptor="d2" src=""/>
97      </composite>
98     </composite>
99     <img id="img-1" descriptor="d3" />
100    <linkBase>
101     <link id="L1" xconnector="finishes.xml">
102      <bind component="C1" port="portA1" role="on_x_presentation_end" />
103      <bind component="C2" port="portA2" role="stop_y" />
104      </link>
105    </linkBase>
106   </composite>
107  </body>
108 </ncl>
```

Figura 5:5 - Exemplo simplificado de documento NCL.

Durante a compilação do elemento *head* (método *parseHead*[45]) o método *parseDescriptorBase* é chamado para compilar a base de descritores (elemento *descriptorBase*). Esse método concreto chama, para cada descritor presente na base (elemento *descriptor*), outro método concreto do tipo *parse*: *parseDescriptor*. Esse método, por sua vez, chama o método abstrato *createDescriptor*, que deve retornar a representação de um descritor no modelo de destino do compilador. Evidentemente, a implementação do método *createDescriptor* é de responsabilidade de cada compilador que venha a ser instanciado a partir do *framework*.

Seguindo com o exemplo da Figura 5:5, durante a compilação do corpo do documento NCL, o método concreto *parseComposite*, declarado na classe *NclComponentsParser*, será chamado para compilar a composição *C*. Esse método

[45] Para facilitar a explicação do exemplo, chamadas a alguns métodos, tais como os métodos de pré e pós processamento, foram omitidas.

retornará um objeto representando essa composição, criado através de uma chamada ao método abstrato *createComposite*. Em seguida, cada elemento interno à composição será compilado. Inicialmente, as composições *C1* e *C2* serão tratadas por chamadas recursivas ao método *parseComposite*. Os objetos representando *C1* e *C2* retornados por essas chamadas serão adicionados à composição *C* através de uma chamada ao método abstrato *addCompositeToComposite*.

Encerrada a compilação das composições *C1* e *C2*, a imagem *img-1* será compilada pelo método *parseMediaContentGroup* (já que o elemento *img*, que define uma imagem, faz parte do grupo de elementos *mediaContentGroup*). O objeto retornado pela chamada a esse método será adicionado a *C* através de uma chamada ao método *addMediaContentGroupToComposite*. Ambos os métodos são também definidos na classe *NclComponentsParser*.

A compilação das composições *C1* e *C2* ocorrerá semelhantemente à compilação de *C*. Os objetos de mídia *audio1*, *title1*, *lyr11* e *lyr12* serão compilados e os resultados das compilações adicionados a *C1*, enquanto *audio2* e *C21* serão compilados e os objetos retornados adicionados a *C2*.

Como pode ser observado, a estrutura genérica do *framework* de compiladores NCL responsabiliza-se pelo caminhamento através da árvore de elementos XML do documento. Às instâncias de compiladores, cabe apenas implementar os métodos abstratos definidos pelo *framework* para criação dos objetos no modelo de destino.

Como descritores são referenciados por objetos de mídia através do atributo *descriptor*, um compilador pode necessitar de uma tabela (indexada pelo *id* do descritor) com todos os descritores retornados pelo método *createDescriptor*. Nesse caso, pode ser usado o vetor de tabelas oferecido pela classe *CFDocumentParser*, superclasse de *NclDocumentCompiler* (como mencionado na Seção 5.1). Assim, o método *createDescriptor* pode adicionar o objeto por ele retornado a uma tabela "descritores" de *NclDocumentCompiler*. Para recuperar uma referência ao descritor apontado por um objeto de mídia, o método *createMediaContentGroup* pode consultar a tabela "descritores" de *NclDocumentCompiler* e obter o descritor a partir de seu *id*.

O vetor de tabelas oferecido por *CFDocumentParser* também pode ser utilizado para criação de uma tabela chamada "retorno". Essa tabela pode conter

alguns dos objetos compilados, de forma a permitir à aplicação que utiliza um compilador obter os objetos retornados pelo compilador. No caso de NCL, essa tabela de retorno pode conter, por exemplo, a composição representada pelo elemento *body*, uma base de descritores, um objeto *layout*, entre outros.

Outro aspecto do *framework* que o exemplo permite destacar é a dependência entre as classes (uma classe chama métodos de outras classes). Como descrito anteriormente, é papel da classe *NclDocumentCompiler* estabelecer as dependências entre as classes do *framework*. Por exemplo, a classe *NclDocumentCompiler* atribui uma referência da classe *NclComponentsParser* para a classe *NclPresentationSpecificationParser*. O código para atribuição das referências entre classes é gerado automaticamente pelo *gerador de frameworks de compiladores.*

5.2.2. Compiladores de documentos NCL

A partir do *framework* de compiladores NCL, foram implementadas quatro instâncias de compiladores, ilustradas na Figura 5:6. Cada instância foi criada pela implementação de uma subclasse para cada uma das classes (abstratas) definidas na Figura 5:6.

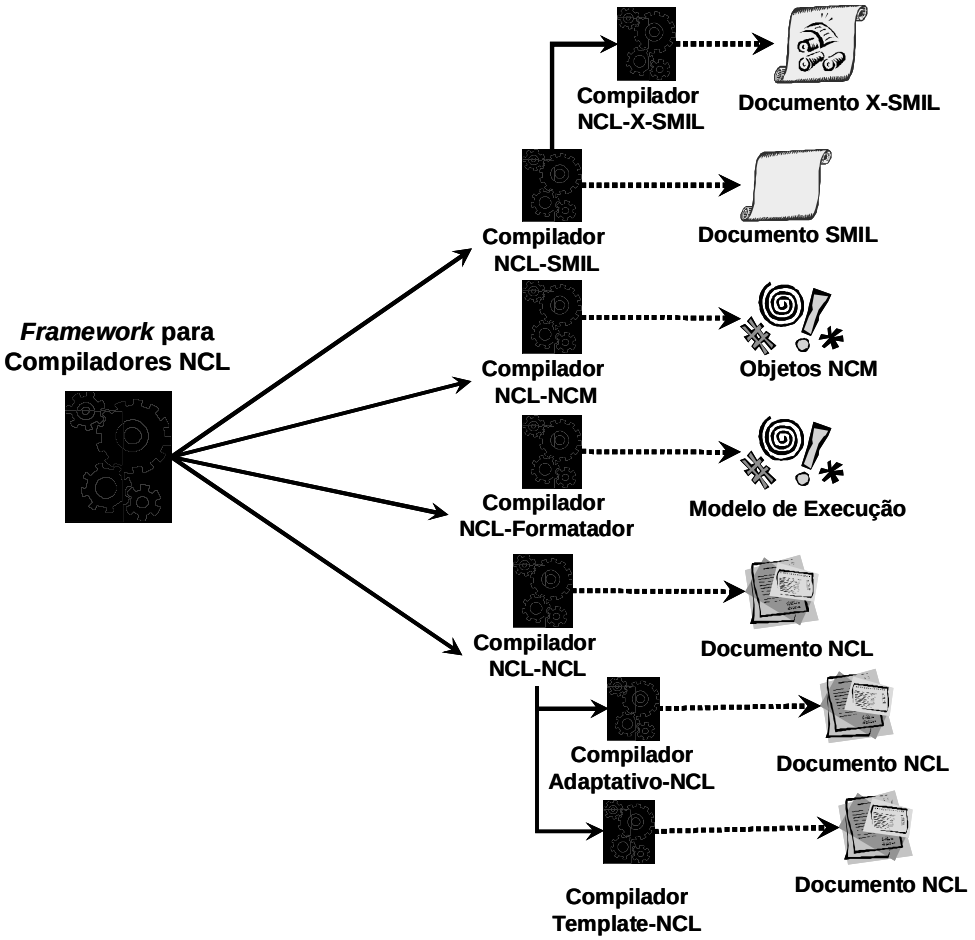

Figura 5:6 - Compiladores NCL.

A primeira instância é o compilador NCL-NCM Java, que gera, como indica o nome, objetos em uma implementação Java do modelo NCM. A estrutura de dados gerada pode ser usada nas ferramentas de autoria do sistema HyperProp. Como NCL é baseada no modelo conceitual NCM, a implementação dos métodos abstratos dessa instância do *framework* foi trivial.

A segunda instância é o compilador NCL-Formatter. Esse compilador gera objetos segundo a estrutura de dados utilizada pelo formatador HyperProp. Os métodos abstratos foram implementados com base nas funções de pré-compilação do formatador, que constroem uma estrutura de dados orientada ao controle da apresentação de um documento. Antes dessa implementação estar disponível, para exibir um documento NCL no formatador HyperProp, esse documento tinha de ser importado para as ferramentas de autoria do sistema HyperProp, através do compilador NCL-NCM, e somente depois convertido para o modelo do formatador. A eliminação do passo intermediário de compilação para objetos

NCM trouxe, como principais benefícios, a redução das necessidades de memória e armazenamento do formatador, além da redução do tempo de conversão. O intuito é favorecer a implementação do conversor/formatador em dispositivos de exibição com recursos limitados, como telefones móveis e *set-top boxes* de TV.

A terceira instância do *framework* é o compilador NCL-SMIL, que gera documentos SMIL 2.0 a partir de documentos NCL. Os elementos pertencentes a módulos que foram trazidos diretamente de SMIL para NCL (como *BasicMedia*), ou ligeiramente modificados (como *Layout*), foram implementados de forma simples. Composições NCL foram mapeadas em elementos *par* e a sincronização definida pelos elos NCL foi descrita através dos eventos de SMIL, como será visto a seguir. Os principais problemas ocorreram com as entidades que só estão presentes em NCL.

A maior dificuldade no mapeamento diz respeito ao uso de conectores. Para possibilitar a transformação, uma classe foi criada com o intuito de simular o comportamento de um conector em SMIL: *SmilConnector*. Ao compilar um conector, uma instância de *SmilConnector* é criada, cujo objetivo é inferir quais eventos SMIL podem ser utilizados para representar o conector NCL. No algoritmo implementado, nem todas as semânticas que podem ser expressadas por conectores NCL podem ser expressas em documentos SMIL.

A Figura 5:7 ilustra uma composição SMIL gerada pelo compilador NCL-SMIL, ao compilar uma composição NCL que referencia o *template audioComLegendas* - utilizado como exemplo nos Capítulos 2 e 3. Note que foi criada uma composição *par* referente à composição NCL; e que foi definido o sincronismo entre o áudio, o *logo* e as legendas por meio de eventos - nos atributos *begin* e *end* dos objetos de mídia - referentes a cada elo NCL. Como somente dois conectores foram utilizados pelos elos da composição NCL, somente duas instâncias da classe *SmilConnector* foram criadas e utilizadas na compilação de todos esses elos.

```
01  <par>
02   <audio id="samba" region="" src="">
03    <area begin="8.4s" end="18s" id="part1"/>
04    <area begin="18.5s" end="28s" id="part2"/>
05    <area begin="29s" end="39s" id="part3"/>
06    <area begin="39.5s" end="50s" id="part4"/>
07    <area begin="50.5s" end="71.4s" id="part5"/>
08    <area begin="72s" end="94s" id="part6"/>
```

```
09   </audio>
10   <img begin="samba.begin" end="samba.end" id="logo" region="" src=""/>
11   <text begin="part1.begin" end="part1.end" id="lyrics1" region="" src=""/>
12   <text begin="part2.begin" end="part2.end" id="lyrics2" region="" src=""/>
13   <text begin="part3.begin" end="part3.end" id="lyrics3" region="" src=""/>
14   <text begin="part4.begin" end="part4.end" id="lyrics4" region="" src=""/>
15   <text begin="part5.begin" end="part5.end" id="lyrics5" region="" src=""/>
16   <text begin="part6.begin" end="part6.end" id="lyrics6" region="" src=""/>
17  </par>
```

Figura 5:7 - Composição SMIL gerada a partir do compilador NCL-SMIL.

O compilador NCL-X-SMIL é uma especialização do compilador NCL-SMIL. Entretanto, como X-SMIL possui o conceito de bases de elos (e elos referenciando conectores) e o elemento *composite*, a compilação de NCL para X-SMIL é mais simples do que para SMIL. Assim, composições (elemento *composite*) NCL são mapeadas para o elemento homônimo em X-SMIL e não é necessária a compilação de conectores (bases de elos NCL são transformadas em bases de elo X-SMIL). As demais funcionalidades do compilador NCL-X-SMIL são herdadas do compilador NCL-SMIL.

À exceção do compilador NCL-X-SMIL, todos os compiladores NCL descritos anteriormente utilizam o compilador XConnector-NCM, instância do *framework* para compiladores XConnector, para processar os conectores que são referenciados por elos. No caso do compilador NCL-NCM, esse uso é obvio. Nos compiladores NCL-Formatador e NCL-SMIL esse compilador de XConnector foi o utilizado por permitir um acesso mais fácil e eficiente às definições de conectores do que seria possível analisando diretamente a estrutura XML dessas definições. Outro ponto importante, nesses compiladores, é o uso do processador de *templates*, que analisa o elemento *body* do documento NCL, processando todas as composições que referenciam *templates* antes que seja iniciado a compilação do elemento *body* e de seu conteúdo. O compilador de XConnector e o processador de *templates* serão discutidos na Seção 5.5.

O último compilador implementado é o NCL-NCL. Esse compilador (gerado automaticamente, conforme mencionado na Seção 5.1) implementa todos os métodos abstratos do *framework* simplesmente gerando uma cópia do documento NCL de entrada. O compilador NCL-NCL foi projetado para se tornar a base para implementação de compiladores NCL adaptativos. Como exemplo, uma especialização desse compilador foi implementada para computar as alternativas de elementos (conteúdo de *switches*) que podem ser resolvidas e

selecionadas antes de iniciar uma apresentação. No novo documento NCL gerado, o elemento *switch* é substituído pelo elemento escolhido, ou por um elemento *switch* com menos alternativas (são retiradas aquelas impossíveis de serem selecionadas). Outro compilador NCL baseado no compilador NCL-NCL é o que processa todos os *templates* de um documento NCL e gera um outro documento NCL processado (ou seja, sem referências a *templates*). Evidentemente, esse compilador também utiliza o processador de *templates* que será descrito na Seção 5.5.

5.3. *Framework* para Compiladores de Documentos SMIL

O *framework* para compiladores SMIL foi a segunda instância gerada automaticamente a partir do *meta-framework*. O diagrama UML de classes dessa instância é ilustrado na Figura 5:8, sendo constituído da classe gerenciadora *SmilDocumentCompiler* e das demais classes representando as áreas funcionais de SMIL. Como as principais características do processo de instanciação de um *framework* para compiladores já foram abordadas na seção anterior, esta seção limita-se a descrever os compiladores SMIL implementados: SMIL-SMIL, SMIL-NCL, SMIL-NCM e SMIL-Formatador.

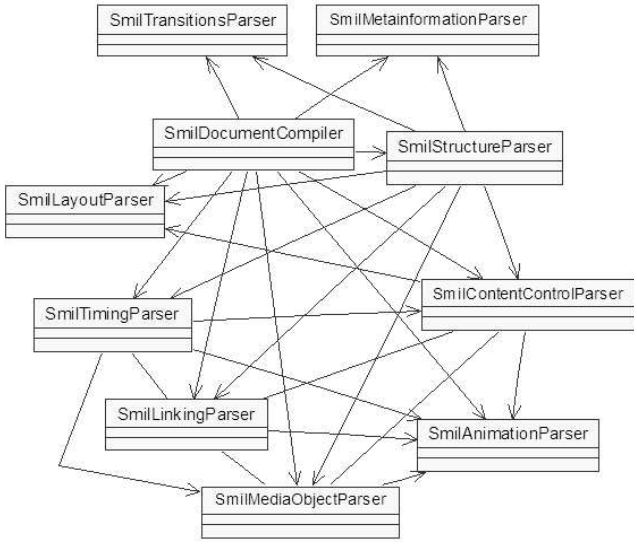

Figura 5:8 - Diagrama de classes do Framework para Compiladores SMIL.

O compilador SMIL-SMIL (também gerado automaticamente) implementa todos os métodos abstratos do *framework* para a geração de uma cópia do documento SMIL de entrada. Similar ao compilador NCL-NCL, esse compilador

é a base para implementação de compiladores adaptativos SMIL, e teve uma especialização implementada para avaliar os elementos *switch* de um documento SMIL.

O compilador SMIL-NCL, como o próprio nome sugere, efetua processamento inverso ao compilador NCL-SMIL. Como nesse último compilador, elementos similares ou comuns nas duas linguagens (como *layout* e o grupo *BasicMedia*) foram trivialmente convertidos. As características de apresentação de cada elemento SMIL (como a região de *layout*) foram agrupadas em descritores NCL (criados pelo compilador). Além disso, cada elemento SMIL, quando compilado para NCL, teve um atributo *descriptor* adicionado para referenciar o descritor criado.

Contêineres SMIL (*par*, *seq*, ou *excl*) foram convertidos em composições NCL. Quando um elemento é adicionado a uma composição NCL, cria-se um elo NCL relacionando esse elemento com os outros elementos já presentes na composição (caso esse não seja o primeiro elemento sendo inserido). Dependendo do contêiner SMIL que deu origem à composição, o elo criado referenciará um determinado conector (já criado pelo compilador): *SmilPar* (contêiner *par*), *SmilSeq* (contêiner *seq*) e *SmilExcl* (contêiner *excl*). Os elos NCL criados oferecem a mesma semântica temporal atribuída pelo contêiner SMIL[46].

Elos em SMIL podem ser definidos a partir de eventos, como os de início e término de apresentação (eventos *begin* e *end*) e de acionamento de mouse (evento *click*), ou a partir do uso de elementos do tipo *a*. Para cada um dos tipos de elos que podem ser definidos em SMIL, foi pré-definido um conector NCL para representar a sua semântica e gerar um ou mais elos NCL correspondentes[47].

Como NCL é baseada no modelo NCM, o compilador SMIL-NCM adota a mesma abordagem utilizada para a compilação de documentos SMIL em documentos NCL e, portanto, não será detalhado neste trabalho. Maiores detalhes do mapeamento entre os modelos podem ser encontrados em [Rodr02].

---

[46] Em alguns casos, mais de um conector é necessário para relacionar elementos de uma composição NCL refletindo uma composição SMIL. Por limitação de espaço, esses casos e as soluções adotadas nos compiladores foram omitidos no texto.

[47] Quando um elemento define o atributo *begin*, o *link* adicionado automaticamente para iniciar a apresentação desse elemento é retirado, para que o início da apresentação do elemento seja dado pelo evento definido no atributo *begin*, como determina a semântica de SMIL.

Por fim, o compilador SMIL-Formatador foi baseado na concatenação dos compiladores SMIL-NCL e NCL-Formatador; e o compilador SMIL-X-SMIL é o próprio compilador SMIL-SMIL, já que todo documento SMIL é um documento válido (e com mesma semântica) em X-SMIL.

5.4. *Framework* para Compiladores de Documentos X-SMIL

Diferente dos *frameworks* para compiladores SMIL e NCL, o *framework* para compiladores X-SMIL não foi gerado automaticamente, mas implementado como uma instância do *framework* para compiladores SMIL. Para tanto, o *framework* para compiladores X-SMIL implementa a classe *XSmilDocumentCompiler*, *XSmilStructureParser* e *XSmilTimingParser* que herdam, respectivamente de *SmilDocumentCompiler*, *SmilStructureParser* e *SmilTimingParser* do *framework* para compiladores SMIL. Utilizando os métodos do tipo *handleElementIn*, esse *framework* adiciona o tratamento dos elementos *linkBase* e *composite* de documentos X-SMIL. Para prover suporte a esse tratamento, o *framework* X-SMIL também aproveita a classe *NclLinkingParser* do *framework* NCL. As demais classes do *framework* X-SMIL são as mesmas do *framework* SMIL.

Por essa estruturação é possível observar que os compiladores X-SMIL são adaptações dos compiladores SMIL combinados com os compiladores NCL já descritos. Assim como os compiladores NCL, compiladores X-SMIL também necessitam processar os *templates* antes da compilação do elemento *body*. Os elementos *linkBase* e *composite* são tratados de forma semelhante ao tratamento dos compiladores NCL, enquanto os demais elementos são compilados da mesma maneira que em SMIL. Herdando partes dos compiladores SMIL-SMIL e NCL-NCL, também foi implementado um compilador X-SMIL-X-SMIL. Uma especialização desse compilador permite gerar documentos X-SMIL com todos os seus *templates* processados.

Os processadores de *templates*, assim como os compiladores de conectores, são assuntos da próxima seção.

5.5. *Framework* para Compiladores de Documentos XConnector e XTemplate

Apesar das linguagens XConnector e XTemplate serem parte integrante da definição da linguagem NCL, cada uma dessas linguagens possui sua própria especificação em XML Schema. Assim, foi gerado o *framework* de compiladores XConnector a partir do gerador de *frameworks*[48], e também uma instância desse *framework*: o compilador XConnector-NCM Java. Uma vez que o modelo NCM serviu como base para a especificação da linguagem XConnector, a implementação desse compilador foi trivial.

Seguindo a mesma abordagem, foi gerado automaticamente um *framework* para compiladores XTemplate 2.1. Uma instância desse *framework* foi implementada como parte integrante do processador de *templates* XTemplate 2.1, que permite a atribuição de semântica às composições NCL, XT-SMIL e X-SMIL.

Um método implementado no processador de *templates* recebe um elemento XML como parâmetro e processa todos os *templates* referenciados pelos elementos filhos desse elemento. Esse método é utilizado por compiladores NCL e X-SMIL para pré-processar o elemento *body* antes de sua compilação (utiliza-se a função *preCompileBody* dos *frameworks* de compiladores).

A Figura 5:9 ilustra graficamente o processamento de um *template*, onde pode-se observar as seguintes entradas: um elemento representando uma composição que referencia um *template* e o documento XTemplate com a especificação do *template* referenciado.

[48] Mesmo a linguagem XConnector não sendo especificada de forma modular, o gerador de *framework* de compiladores pôde ser usado, ao considerar XConnector como uma linguagem com um único módulo.

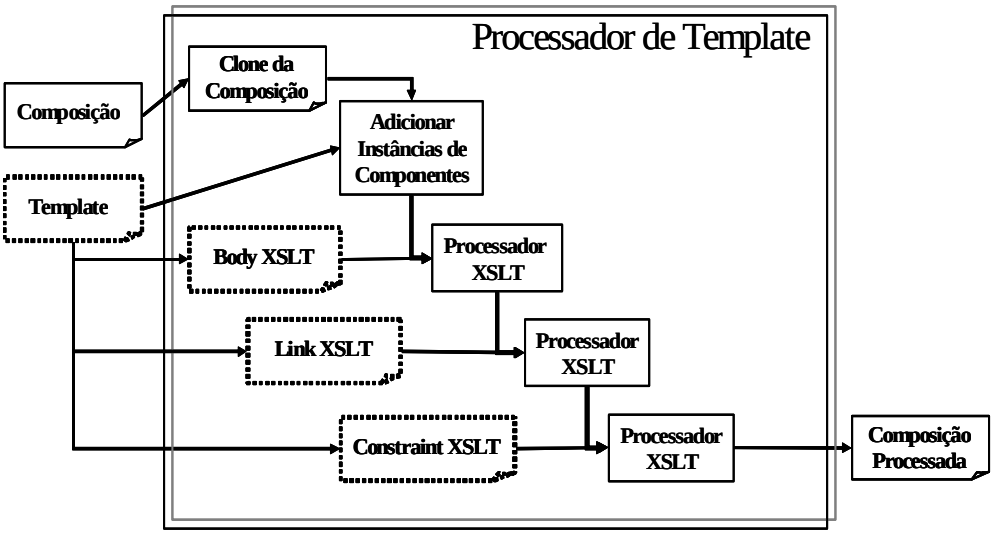

Figura 5:9 - Processador de *Template* XTemplate 2.1.

O processador de *template*, inicialmente, gera uma cópia da composição original. Essa cópia é inserida como único elemento filho do elemento raiz (nomeado *root*) de um documento XML temporário. Considerando, como exemplo, o processamento de uma composição *C1* que referencie o *template audioWithSubtitlesEnPt21*, descrito no Capítulo 3 (Figura 3:14), o documento XML temporário, contendo uma cópia dessa composição, está ilustrado na Figura 5:10.

Em seguida, analisando os recursos declarados no cabeçalho do *template*, instâncias de componentes (recursos) definidas por elementos *resource* são adicionadas a essa cópia. Como *audioWithSubtitlesEnPt21* não define recursos no cabeçalho, nenhum elemento é adicionado nesse passo. No outro exemplo de *template* do Capítulo 3 (Figura 3:12), o recurso *logoJPG* seria adicionado, nesse momento, como elemento filho da cópia da composição.

```
01  <root>
02  <composite id="C1" xtemplate=" audioWithSubtitlesEnPt21.xml">
03    <audio id="samba" src="coisadepeleBeth.wav" type="song">
04      <area begin="8.4s" end="18.5s" id="part1" type="track"/>
05      <area begin="18.5s" end="29s" id="part2" type="track"/>
06      <area begin="29s" end="39.5s" id="part3" type="track"/>
07    </audio>
08    <text id="lyrics-part1a" src="versosEn.html#versos01" type="subtitleEn"/>
09    <text id="lyrics-part1b" src="versosPt.html#versos01" type="subtitlePt"/>
10    <text id="lyrics-part2a" src="versosEn.html#versos02" type="subtitleEn"/>
11    <text id="lyrics-part2b" src="versosPt.html#versos02" type="subtitlePt"/>
12    <text id="lyrics-part3a" src="versosEn.html#versos03" type="subtitleEn"/>
13    <text id="lyrics-part3b" src="versosPt.html#versos03" type="subtitlePt"/>
14  </composite>
```

```
15  </root>
```

Figura 5:10 - Documento XML com a cópia de uma composição sendo processada.

Analisando em seguida o corpo do *template*, são geradas três transformadas XSLT para serem aplicadas, em seqüência, ao documento da Figura 5:10, gerando a composição processada da Figura 5:11. Essas transformadas seguem o padrão XSLT (W3C, 1999d) e podem ser aplicadas ao documento utilizando qualquer processador dessa linguagem.

```
01    <composite id="C1">
02      <audio id="samba" region="audioRegion1" src="coisadepeleBeth.wav">
03        <area begin="8.4s" end="18.5s" id="part1"/>
04        <area begin="18.5s" end="29s" id="part2"/>
05        <area begin="29s" end="39.5s" id="part3"/>
06      </audio>
07      <switch id="Switch1">
08        <text id="lyrics-part1a" src="versosEn.html#versos01"/>
09        <bindRule component="lyrics-part1a" rule="RuleEnUS"/>
10        <text id="lyrics-part1b" src="versosPt.html#versos01"/>
11        <bindRule component="lyrics-part1b" rule="RuleEnPT"/>
12      </switch>
13      <switch id="Switch2">
14        <text id="lyrics-part2a" src="versosEn.html#versos02"/>
15        <bindRule component="lyrics-part2a" rule="RuleEnUS"/>
16        <text id="lyrics-part2b" src="versosPt.html#versos02"/>
17        <bindRule component="lyrics-part2b" rule="RuleEnPT"/>
18      </switch>
19      <switch id="Switch3">
20        <text id="lyrics-part3a" src="versosEn.html#versos03"/>
21        <bindRule component="lyrics-part3a" rule="RuleEnUS"/>
22        <text id="lyrics-part3b" src="versosPt.html#versos03"/>
23        <bindRule component="lyrics-part3b" rule="RuleEnPT"/>
24      </switch>
25      <linkBase>
26        <link xconnector="starts.xml">
27          <bind component="samba" port="part1" role="on_x_presentation_begin"/>
28          <bind component="Switch1" role="start_y"/>
29        </link>
30        <link xconnector="finishes.xml">
31          <bind component="samba" port="part1" role="on_x_presentation_end"/>
32          <bind component="Switch1" role="stop_y"/>
33        </link>
34        <link xconnector="starts.xml">
35          <bind component="samba" port="part2" role="on_x_presentation_begin"/>
36          <bind component="Switch2" role="start_y"/>
37        </link>
38        <link xconnector="finishes.xml">
39          <bind component="samba" port="part2" role="on_x_presentation_end"/>
40          <bind component="Switch2" role="stop_y"/>
41        </link>
```

```
42        <link xconnector="starts.xml">
43          <bind component="samba" port="part3" role="on_x_presentation_begin"/>
44          <bind component="Switch3" role="start_y"/>
45        </link>
46        <link xconnector="finishes.xml">
47          <bind component="samba" port="part3" role="on_x_presentation_end"/>
48          <bind component="Switch3" role="stop_y"/>
49        </link>
50      </linkBase>
51    </composite>
```

Figura 5:11 - Composição gerada pelo processador de *templates*.

A primeira transformada, chamada *body XSLT*, é gerada a partir dos elementos *xsl:stylesheet* definidos como filhos diretos do elemento *body* do *template*, ou seja, ela especifica relacionamentos de inclusão. A transformada do exemplo está ilustrada na Figura 5:12, onde algumas partes são constantes para todas as transformadas desse tipo: as linhas 2-6 indicam a cópia do elemento raiz (*root*); as linhas 7-9 definem a cópia de todos os elementos filhos da composição; e as linhas 10-20 definem a cópia do elemento representando a composição (retirando seu atributo *xtemplate*), cujo conteúdo é formado a partir de outros *xsl:template* da transformada (pelo comando da linha 17) e pelo *xsl:template* nomeado *mainTemplate* (chamado na linha 18). O restante da transformada é obtido a partir da especificação do *template*, sendo todos os elementos do tipo *xsl:resource* substituídos por um elemento *xsl:element*, para geração de elementos referentes aos tipos declarados no vocabulário do *template*. Por exemplo, na linha 24 foi definido o *xsl:element* referente ao recurso do tipo (*type*) *switch*, cujo nome (*name*) é o tipo de conteúdo (*ctype*) desse recurso: *switch*. Um atributo *type* para esse novo elemento é gerado automaticamente, a partir de seu tipo (linha 25). Após a substituição de todos os *xsl:resource* pelos *xsl:element* correspondentes, obtém-se o *xsl:template mainTemplate* (linhas 21-43) que contém todos os elementos definidos na especificação do *template* que não são do tipo *xsl:template*. Os elementos do tipo *xsl:template* são adicionados diretamente ao elemento *xsl:stylesheet* da transformada (linhas 44-74). Após o processamento da transformada *body XSLT* a composição gerada é similar à composição da Figura 5:11 excluindo o elemento *linkBase*.

```
01  <xsl:stylesheet ...>
02    <xsl:template match="/">
03      <xsl:copy>
04        <xsl:apply-templates/>
05      </xsl:copy>
```

```
06    </xsl:template>
07    <xsl:template match="/*/*/*">
08      <xsl:copy-of select="."/>
09    </xsl:template>
10    <xsl:template match="/*/*">
11      <xsl:copy>
12        <xsl:for-each select="@*">
13          <xsl:if test="not(name(.) = 'xtemplate')">
14            <xsl:copy/>
15          </xsl:if>
16        </xsl:for-each>
17        <xsl:apply-templates/>
18        <xsl:call-template name="mainTemplate"/>
19      </xsl:copy>
20    </xsl:template>
21    <xsl:template name="mainTemplate">
22      <xsl:for-each select="//*[@type='subtitleEn']">
23        <xsl:variable name="i" select="position()"/>
24        <xsl:element name="switch">
25          <xsl:attribute name="type">switch</xsl:attribute>
26          <xsl:attribute name="id">Switch
27            <xsl:value-of select="$i"/>
28          </xsl:attribute>
29          <xsl:copy>
30            <xsl:for-each select="text()|@*">
31              <xsl:copy/>
32            </xsl:for-each>
33          </xsl:copy>
34          <xsl:call-template name="bindRuleElement">
35            <xsl:with-param name="component" select="./@id"/>
36            <xsl:with-param name="rule">RuleEnUS</xsl:with-param>
37          </xsl:call-template>
38          <xsl:call-template name="ptSwitchElement">
39            <xsl:with-param name="i" select="$i"/>
40          </xsl:call-template>
41        </xsl:element>
42      </xsl:for-each>
43    </xsl:template>
44    <xsl:template match="/*/*/*">
45      <xsl:if test="not(./@type='subtitleEn') and not(./@type='subtitlePt')">
46        <xsl:copy-of select="."/>
47      </xsl:if>
48    </xsl:template>
49    <xsl:template name="ptSwitchElement">
50      <xsl:param name="i"/>
51      <xsl:for-each select="/*/*/child::*[@type='subtitlePt'][$i]">
52        <xsl:copy>
53          <xsl:for-each select="text()|@*">
54            <xsl:copy/>
55          </xsl:for-each>
56        </xsl:copy>
```

```
57        <xsl:call-template name="bindRuleElement">
58          <xsl:with-param name="component" select="./@id"/>
59          <xsl:with-param name="rule">RuleEnPT</xsl:with-param>
60        </xsl:call-template>
61      </xsl:for-each>
62    </xsl:template>
63    <xsl:template name="bindRuleElement">
64      <xsl:param name="component"/>
65      <xsl:param name="rule"/>
66      <xsl:element name="bindRule">
67        <xsl:attribute name="component">
68          <xsl:value-of select="$component"/>
69        </xsl:attribute>
70        <xsl:attribute name="rule">
71          <xsl:value-of select="$rule"/>
72        </xsl:attribute>
73      </xsl:element>
74    </xsl:template>
75  </xsl:stylesheet>
```

Figura 5:12 - Exemplo de transformada *body XSLT*.

A segunda transforma a ser aplicada é chamada *link XSLT*. Essa transformada, gerada a partir da especificação do elemento *linkBase* do corpo do *template*, corresponde à inserção dos relacionamentos baseados em conectores definidos pelo *template*. Na Figura 5:13, a primeira parte da transformada (linhas 01-20) é similar à transformada da Figura 5:12, sendo que a chamada da linha 18 passa a apontar para o *xsl:template linkBaseTemplate*. Esse *xsl:template* (linhas 21-25) adiciona um elemento *linkBase* à composição (linhas 22-24), cujo conteúdo (*links*) é definido pelo *xsl:template linkTemplate*, que é gerado a partir do *template*, como se segue. Todo conteúdo da transformada especificada pelo elemento *linkBase* no cabeçalho do *template* é copiado para *linkTemplate*, sendo que:

- todo elemento *xsl:link* é transformado em um elemento *xsl:element* (como nas linha 29-72) para geração de um *link* na base de elos da composição, sendo adicionados automaticamente os atributos *xconnector* (linha 30) e *type* (linha 31), obtidos a partir do vocabulário do *template*;
- para cada elemento *xsl:bind* de *xsl:link* é gerado um elemento *xsl:for-each* (como nas linhas 32-51 e 52-71), cujo atributo *select* (linhas 32 e 52) corresponde ao atributo *select* do *xsl:bind*. Para cada *xsl:for-each* define-se um elemento com nome *bind* (linhas 33-50 e

53-70), com um atributo *role* igual ao atributo *role* de *xsl:bind* (linhas 54 e 34). Além do atributo *role*, o elemento *bind* terá os atributos *component* e *port*, caso o elemento referenciado por esse *bind* seja do tipo *port* ou *area* (linhas 36-43 e 56-63); ou terá somente o atributo *component* caso contrário (linhas 44-48 e 64-68). O atributo *component* (ou a combinação dos atributos *component* e *port*), como visto na Seção 2.4, mapeiam um componente do documento ao papel do conector (atributo *role*) definido pelo *bind*.

Na Figura 5:13 temos, portando, a definição de um *loop* do tipo *xsl:for-each* (linhas 27-117) - especificado pelo arquivo XTemplate - que define dois tipos de elos, referenciando dois conectores do vocabulário do *template*, o primeiro nas linhas 29-72 e o segundo nas linhas 73-116.

```
01  <xsl:stylesheet ...>
02    <xsl:template match="/">
03      <xsl:copy>
04        <xsl:apply-templates/>
05      </xsl:copy>
06    </xsl:template>
07    <xsl:template match="/*/*/*">
08      <xsl:copy-of select="."/>
09    </xsl:template>
10    <xsl:template match="/*/*">
11      <xsl:copy>
12        <xsl:for-each select="@*">
13          <xsl:if test="not(name(.) = 'xtemplate')">
14            <xsl:copy/>
15          </xsl:if>
16        </xsl:for-each>
17        <xsl:apply-templates/>
18        <xsl:call-template name="linkBaseTemplate"/>
19      </xsl:copy>
20    </xsl:template>
21    <xsl:template name="linkBaseTemplate">
22      <xsl:element name="linkBase">
23        <xsl:call-template name="linkTemplate"/>
24      </xsl:element>
25    </xsl:template>
26    <xsl:template name="linkTemplate">
27      <xsl:for-each select="child::*[@type='song']/child::*[@type='track']">
28        <xsl:variable name="i" select="position()"/>
29        <xsl:element name="link">
30          <xsl:attribute name="xconnector">starts.xml</xsl:attribute>
31          <xsl:attribute name="type">L</xsl:attribute>
32          <xsl:for-each select="current()">
33            <xsl:element name="bind">
```

```
34              <xsl:attribute name="role">on_x_presentation_begin</xsl:attribute>
35              <xsl:choose>
36                <xsl:when test="name(.)='port' or name(.)='area'">
37                  <xsl:attribute name="component">
38                    <xsl:value-of select="../@id"/>
39                  </xsl:attribute>
40                  <xsl:attribute name="port">
41                    <xsl:value-of select="@id"/>
42                  </xsl:attribute>
43                </xsl:when>
44                <xsl:otherwise>
45                  <xsl:attribute name="component">
46                    <xsl:value-of select="@id"/>
47                  </xsl:attribute>
48                </xsl:otherwise>
49              </xsl:choose>
50            </xsl:element>
51          </xsl:for-each>
52          <xsl:for-each select="//*[@type='switch'][$i]">
53            <xsl:element name="bind">
54              <xsl:attribute name="role">start_y</xsl:attribute>
55              <xsl:choose>
56                <xsl:when test="name(.)='port' or name(.)='area'">
57                  <xsl:attribute name="component">
58                    <xsl:value-of select="../@id"/>
59                  </xsl:attribute>
60                  <xsl:attribute name="port">
61                    <xsl:value-of select="@id"/>
62                  </xsl:attribute>
63                </xsl:when>
64                <xsl:otherwise>
65                  <xsl:attribute name="component">
66                    <xsl:value-of select="@id"/>
67                  </xsl:attribute>
68                </xsl:otherwise>
69              </xsl:choose>
70            </xsl:element>
71          </xsl:for-each>
72        </xsl:element>
73        <xsl:element name="link">
74          <xsl:attribute name="xconnector">finishes.xml</xsl:attribute>
75          <xsl:attribute name="type">P</xsl:attribute>
76          <xsl:for-each select="current()">
77            <xsl:element name="bind">
78              <xsl:attribute name="role">on_x_presentation_end</xsl:attribute>
79              <xsl:choose>
80                <xsl:when test="name(.)='port' or name(.)='area'">
81                  <xsl:attribute name="component">
82                    <xsl:value-of select="../@id"/>
83                  </xsl:attribute>
84                  <xsl:attribute name="port">
```

```
85                    <xsl:value-of select="@id"/>
86                  </xsl:attribute>
87                </xsl:when>
88                <xsl:otherwise>
89                  <xsl:attribute name="component">
90                    <xsl:value-of select="@id"/>
91                  </xsl:attribute>
92                </xsl:otherwise>
93              </xsl:choose>
94            </xsl:element>
95          </xsl:for-each>
96          <xsl:for-each select="//*[@type='switch'][$i]">
97            <xsl:element name="bind">
98              <xsl:attribute name="role">stop_y</xsl:attribute>
99              <xsl:choose>
100               <xsl:when test="name(.)='port' or name(.)='area'">
101                 <xsl:attribute name="component">
102                   <xsl:value-of select="../@id"/>
103                 </xsl:attribute>
104                 <xsl:attribute name="port">
105                   <xsl:value-of select="@id"/>
106                 </xsl:attribute>
107               </xsl:when>
108               <xsl:otherwise>
109                 <xsl:attribute name="component">
110                   <xsl:value-of select="@id"/>
111                 </xsl:attribute>
112               </xsl:otherwise>
113             </xsl:choose>
114           </xsl:element>
115         </xsl:for-each>
116       </xsl:element>
117     </xsl:for-each>
118   </xsl:template>
119 </xsl:stylesheet>
```

Figura 5:13 - Exemplo de transformada *link XSLT*.

Terminado o processamento da segunda transformada, a composição já possui a semântica definida pelos relacionamentos herdados do *template*. Entretanto, mais um passo é necessário: checar as restrições. Para isso a transformada *constraint XSLT* é gerada a partir das restrições do *template*, especificadas pela cardinalidade e aninhamento dos elementos do vocabulário, e pelas restrições definidas por elementos *constraint*. A Figura 5:14 ilustra uma transformada desse tipo, onde as linhas 2-11 definem a cópia da composição de entrada (eliminando-se os atributos *type* e *label*) e as linhas 12-29 especificam as restrições que devem ser verificadas no *template*. Caso essas restrições não sejam válidas, o processador de *templates* gera uma mensagem de erro. No exemplo,

apenas restrições sobre cardinalidade e aninhamento são validadas (a partir de testes e mensagens de erro geradas automaticamente pelo processador de *templates*), já que o *template* não define restrições adicionais.

```
01  <xsl:stylesheet ...>
02    <xsl:template match="/*//*">
03      <xsl:copy>
04        <xsl:for-each select="@*">
05          <xsl:if test="not(name(.) = 'type' or name(.) = 'label')">
06            <xsl:copy/>
07          </xsl:if>
08        </xsl:for-each>
09        <xsl:apply-templates/>
10      </xsl:copy>
11    </xsl:template>
12    <xsl:template match="/">
13      <xsl:apply-templates/>
14      <xsl:if test="count(//*[@type='song']) &gt; 1">
15        <xsl:message terminate="no">Max number of elements of type song is
1</xsl:message>
16      </xsl:if>
17      <xsl:if test="1 &gt; count(//*[@type='song'])">
18        <xsl:message terminate="no">Min number of elements of type song is
1</xsl:message>
19      </xsl:if>
20      <xsl:if      test="not(count(//*[@type='song']/*[@type='track'][1])      =
count(//*[@type='song']))">
21        <xsl:message terminate="no">Min number of elements of type /song/track
is 1</xsl:message>
22      </xsl:if>
23      <xsl:if  test="not((count(//*[@type='switch']/*[@type='subtitleEn'][1])  =
count(//*[@type='switch']))                                                  and
(not(//*[@type='switch']/*[@type='subtitleEn'][2])))">
24        <xsl:message   terminate="no">Min/Max   number   of   elements   of   type
/switch/subtitleEn is 1/1</xsl:message>
25      </xsl:if>
26      <xsl:if  test="not((count(//*[@type='switch']/*[@type='subtitlePt'][1])  =
count(//*[@type='switch']))                                                  and
(not(//*[@type='switch']/*[@type='subtitlePt'][2])))">
27        <xsl:message   terminate="no">Min/Max   number   of   elements   of   type
/switch/subtitlePt is 1/1</xsl:message>
28      </xsl:if>
29    </xsl:template>
30  </xsl:stylesheet>
```

Figura 5:14 - Exemplo de transformada *constraint XSLT*.

Finalmente, aplicada a transformada *constraint XSLT*, obtém-se a composição processada. Essa composição está ilustrada na Figura 5:11, onde é possível observar os relacionamentos de inclusão e os relacionamentos por meio de conectores obtidos pelas transformadas da Figura 5:12 e Figura 5:13,

respectivamente. Essa composição pode, então, substituir a composição original do documento.