6
Trabalhos Relacionados

Este capítulo apresenta trabalhos relacionados com os principais temas abordados por esta dissertação. São descritos trabalhos relativos a: *templates* de composição, processamento de documentos XML, *framework* para compiladores, compiladores XML, extensões à SMIL e conversão entre modelos de documento hipermídia.

6.1.
Template de Composição

Celentano e Gaggi (Celentano & Gaggi, 2003) descrevem a geração de apresentações multimídia baseada em *templates*. Essas apresentações são construídas automaticamente por uma aplicação que realiza consultas a um banco de dados. *Templates* definem o comportamento da apresentação e a sincronização entre seus objetos. Dessa forma, *templates* são especificados uma única vez e cada nova apresentações é gerada a partir dos dados (instâncias de objetos) obtidos de um repositório.

A Figura 6:1 ilustra um documento que representa um noticiário, composto por três artigos, no ambiente de autoria gráfica do sistema, chamado LAMP (*LAboratory for Multimedia presentations Prototyping*). Uma música de fundo (*soundtrack*) é apresentada continuamente durante a exibição em seqüência dos artigos, sendo cada artigo composto por uma narração ($news_i$), um vídeo ($vídeo_i$) e uma legenda ($caption_i$). Ao término da exibição do último artigo, a música de fundo é substituída por um *jingle* e os créditos (*credits*) são apresentados. Os relacionamentos entre os componentes do documento são definidos por arestas referenciando um dos cinco tipos de relacionamentos oferecidos pelo sistema (Celentano & Gaggi, 2003).

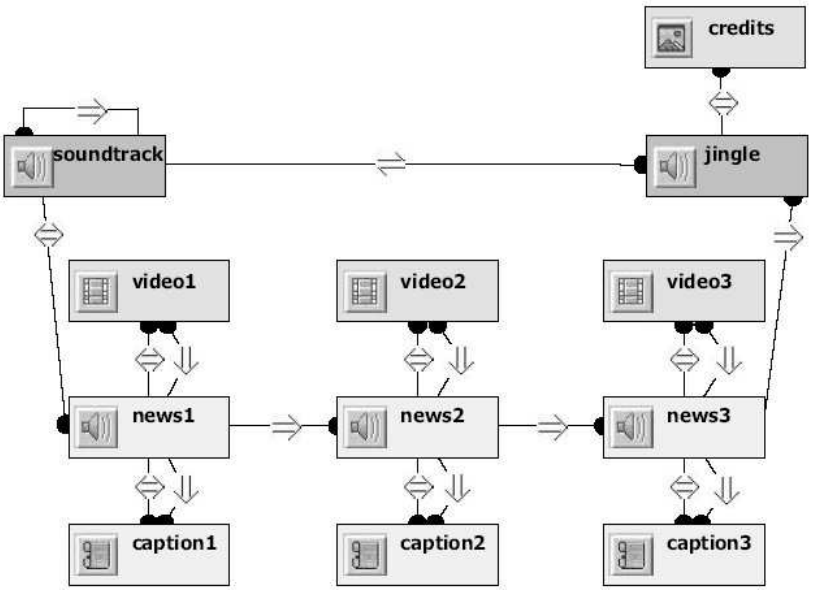

Figura 6:1 - Exemplo de um documento representando um noticiário no sistema LAMP.

Analisando a Figura 6:1, pode-se observar uma estrutura recorrente na definição dos artigos, possibilitando, dessa forma, a especificação de um *template*. O *template* para definição de artigos é ilustrado na Figura 6:2, também visualizado no ambiente gráfico de autoria do LAMP. Na definição de *templates*, uma composição pode ser do tipo *stencil*, que determina um *template* de composição[49] para geração de composições de um mesmo tipo (a partir de uma consulta a um banco de dados). Na Figura 6:2, a composição *article* é do tipo *stencil*, sendo permitido que seus relacionamentos definam, além de um tipo, um atributo referenciando qual tupla retornada pela consulta ao banco de dados deve ser relacionada. Esse atributo pode ter os seguinte valores: *first* (primeira tupla), *last* (última tupla), *next* (próxima tupla) ou *all* (todas tuplas) - valor padrão.

[49] O termo *template de composição* não é o utilizado pelos autores do artigo, sendo utilizado neste texto para facilitar a comparação com as definições desta dissertação.

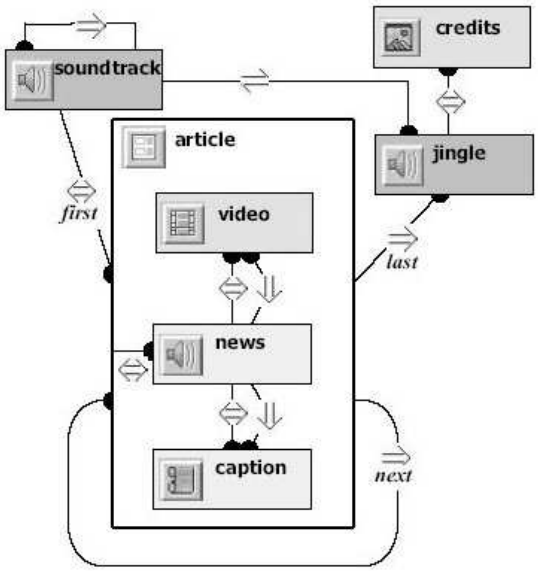

Figura 6:2 - *Template* para artigos de um noticiário.

Baseada no mesmo modelo hipermídia do ambiente de autoria gráfica, foi desenvolvida uma linguagem no padrão XML para especificação de documentos (apresentações) e *templates*. Foi possível a definição de uma única linguagem para a autoria tanto de documentos quanto de *templates*, com a seguinte restrição: elementos do tipo *stencil* somente podem ser declarados na especificação de *templates*. Esse elemento é utilizado para a geração das composições de uma apresentação (ou documento) após uma consulta a um banco de dados. O "processamento de *templates*" consulta um banco de dados e, a partir da resposta, gera composições referentes a elementos *stencil* e, também, os relacionamentos definidos no *template* que envolvem esses elementos.

A proposta apresentada é, em vários aspectos, similar a de *templates* de composição descrita neste trabalho. Suas principais contribuições são: a consulta a bancos de dados para geração de documentos; o oferecimento de um ambiente gráfico para autoria de *templates*; e o uso da mesma linguagem para definição tanto de documentos quanto de *templates*. Entretanto, quando comparado ao NCM, o modelo hipermídia no qual o LAMP se baseia é pouco expressivo (entre outras limitações, apenas cinco tipos de relacionamentos são permitidos), o que facilitou a definição da linguagem para autoria de documentos e *templates*. Além disso, o processador de *templates* do LAMP é baseado em um algoritmo específico que não faz uso de padrões relacionados à XML, enquanto NCL

estende o padrão XSLT para definição de seus *templates* e utiliza processadores de XSLT no processamento de *templates*.

6.2. Processamento de documentos XML

Existem duas principais abordagens para o processamento de documentos XML. A primeira se baseia no caminhamento por uma estrutura de dados em árvore, padronizada em *Document Object Model* - DOM (W3C, 2000a). A segunda abordagem é baseada em eventos (disparados durante a leitura de documentos e tratados por processadores), seguindo o padrão *de facto* SAX (SAX, 2002). Esses dois padrões, que são independentes de linguagem de programação, possuem uma API definida para a linguagem Java: *Java API for XML Processing* - JAXP (Sun, 2002). A Figura 6:3 e a Figura 6:4 ilustram, respectivamente, a definição da API Java para DOM e para SAX[50]. JAXP foi a API utilizada nos compiladores desenvolvidos neste trabalho, por meio das implementações Xerces2 (Apache XML, 2002b) e Xalan (Apache XML, 2002a) para o processamento de arquivos XML e o processamento de folhas de estilo XSLT, respectivamente.

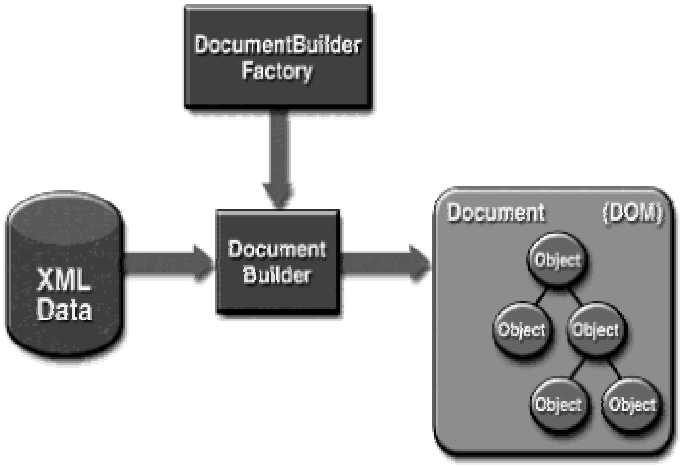

Figura 6:3 - API DOM de JAXP.

A API para DOM define a classe *DocumentBuilderFactory*, a ser utilizada para obter uma instância de *DocumentBuilder*. Essa instância gera, a partir de

[50] As figuras foram extraídas do tutorial "*The J2EE 1.4 Tutorial*", disponível em http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html.

dados XML, um objeto (instância da classe *Document*) em conformidade com a especificação DOM.

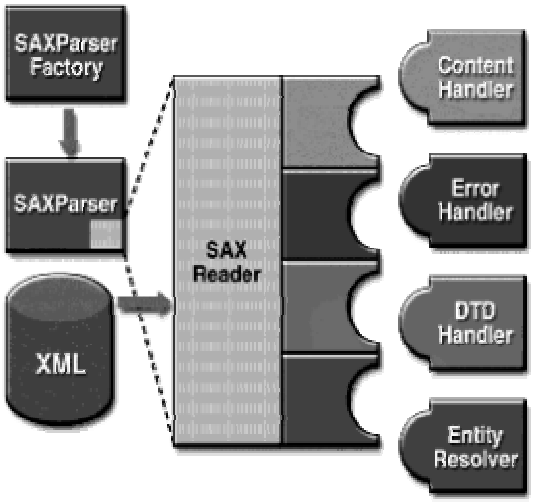

Figura 6:4 - API SAX de JAXP.

O processamento de um documento seguindo a API SAX se inicia pela classe *SAXParserFactory*, que gera uma instância da classe *SAXParser*. Essa classe *parser* possui um leitor SAX (objeto da classe *SAXReader*), que lê um documento XML e invoca métodos (*callback methods*) implementados na aplicação usuária da API (processador XML). Esses métodos são definidos pelas interfaces *ContentHandler*, *ErrorHandler*, *DTDHandler* e *EntityResolver*. Um objeto *DefaultHandler* (não ilustrado na Figura 6:4) implementa todas essas interfaces com métodos "nulos". Aplicações devem herdar dessa classe, implementando somente os métodos de interesse. A interface *ContentHandler* define os principais métodos para processamento de documentos, como: *startDocument* e *endDocument*, chamados no início e término de processamento de um documento, e *startElement* e *endElement*, que são chamados a cada início e término de processamento de um elemento XML. Essa interface também define métodos para tratamento do texto presente como conteúdo de elementos XML. A interface *ErrorHandler* define os métodos *error*, *fatalError* e *warning*, para tratamento dos diversos tipos de erros que podem ocorrer na análise de um documento XML. Os métodos das interfaces *DTDHandler* e *EntityResolver* raramente precisam ser implementados, sendo chamados para tratar especificidades de DTDs e para computar a localização de objetos referenciados pelo documento.

A abordagem DOM possui a vantagem de oferecer acesso a todo documento XML ("*random access*"), já que a árvore que o representa é construída e pode tanto ser consultada quanto alterada. Dessa forma, essa abordagem é recomendada quando o conteúdo do documento como um todo deve ser analisado (de forma não linear), ou quando se deseja alterar o conteúdo de um documento (de forma interativa). Por outro lado, por construir uma representação do documento XML, tipicamente uma aplicação que utilize DOM requer mais recursos de memória.

Quando o acesso serial ("*serial access*") ao conteúdo do documento é suficiente, ou quando somente uma parte desse documento necessita ser analisada, SAX torna-se a opção recomendada. Como nenhuma estrutura intermediária é construída em um processador SAX, quando comparado com processadores DOM, normalmente tem-se um uso mais eficiente de memória e uma execução mais rápida. Entretanto, SAX é um padrão que não apresenta a noção de estado ("*stateless*"), já que os métodos são chamados linearmente durante a leitura do documento, independente do que já foi processado.

Para ser genérica, a estratégia adotada pelo *meta-framework* de compiladores é baseada no padrão DOM. Dessa forma, é permitido um acesso não linear aos elementos quando um documento é compilado. Por exemplo, um compilador pode consultar dados a respeito do pai do elemento sendo computado ou obter a quantidade de filhos de um determinado tipo que esse elemento possui. Isso, a princípio, não é possível utilizando SAX. Outra razão para escolha dessa estratégia é possibilitar o uso de processadores XSLT, já que a representação de um elemento como um nó de uma árvore DOM é necessária para se aplicar transformadas.

6.3. *Framework* e Compiladores

Diversos estudos abordam a geração automática de parte do código para compiladores de linguagens de programação. Ferramentas como Lex/Flex e Yacc/Bison (Levine, 1995) automatizam parte das tarefas de construção de compiladores, utilizando conceitos precisos para expressões regulares e para a gramática de linguagens. *Java Compiler Compiler* - JavaCC (JavaCC, 2003) é uma ferramenta que segue esse princípio, ou seja, lê uma especificação gramatical

de uma linguagem de programação e a converte em um programa Java, que reconhece padrões nessa gramática. Dessa forma, é gerado automaticamente um *parser* em Java para essa linguagem e são oferecidos mecanismos para construção de árvores sintáticas e tratamento de erros, possibilitando o desenvolvimento de compiladores.

Uma abordagem semelhante é proposta em *ANother Tool for Language Recognition* - ANTLR (ANTLR, 2005): uma ferramenta baseada em um *framework* para construção de compiladores e tradutores. Diferente de JavaCC, é oferecido suporte para geração de código em outras linguagens além de Java: C#, Python e C++. ANTLR oferece mecanismos para construção e análise de árvores sintáticas abstratas (*Abstract Syntax Tree* - AST); permite tratamento de erros; e, segundo os desenvolvedores da ferramenta, gera um código de fácil compreensão por programadores.

Essas propostas para geração automática de código para compiladores não estão relacionadas com XML. Assim, estendendo ANTLR, são apresentadas, em (Widemann et al., 2003), duas ferramentas baseadas em XML para geração automática de compiladores: XANTLR e TDOM. A Figura 6:5 ilustra, de forma simplificada, essas ferramentas - implementadas na linguagem Java. A ferramenta XANTLR, definida como um "ANTLR com um pré-processador", recebe como entrada a especificação da gramática de uma linguagem e gera, automaticamente, duas saídas: (1) um código para *parse* de documentos nessa linguagem, que estende o código gerado por ANTLR, ao definir uma representação em XML para a árvore sintática e um conjunto de eventos SAX para o processamento dessa árvore; e (2) uma DTD que reflete exatamente a sintaxe abstrata da linguagem. TDOM, por sua vez, recebe como entrada a DTD gerada por XANTLR e constrói, automaticamente, um conjunto de classes Java representando a sintaxe da linguagem. Uma classe é gerada para cada *elemento* definido pela DTD, contendo implementações de funções para *parse*. Essas funções são chamadas pelos eventos SAX gerados pela árvore XANTLR para construção da árvore TDOM, e podem ser re-implementados por compiladores de documentos na linguagem especificada.

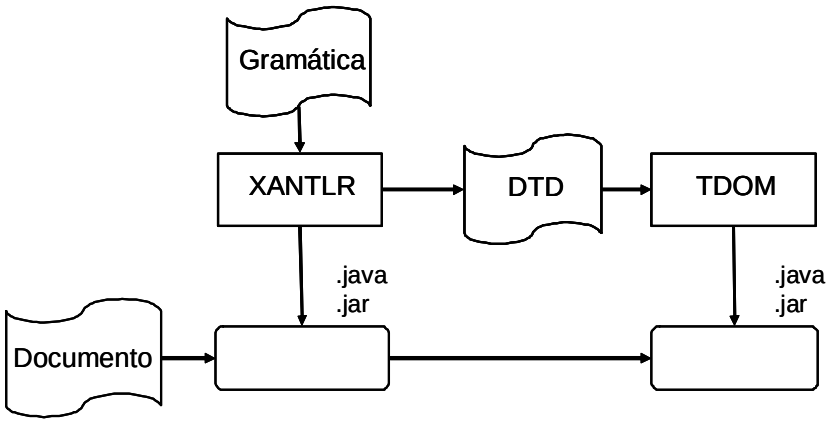

Figura 6:5 - Ferramentas XANTLR e TDOM.

A estrutura de dados em árvore de TDOM é formada por objetos das classes geradas pelos *elementos* definidos na DTD. Dessa forma, ao contrário de árvores DOM e AST, essa árvore possui objetos com tipos definidos. Essa abordagem, segundo os autores da proposta, é mais expressiva, pois conceitos como herança podem ser aplicados aos nós da árvore. A estrutura TDOM não implementa a interface definida pelo padrão DOM, porém, métodos para geração de árvores DOM a partir de TDOM, e vice-versa, são oferecidos.

Os conhecimentos adquiridos na área de compiladores de linguagens de programação devem, sempre que possível, ser aproveitados. Entretanto, as estruturas geradas por compiladores ou *parsers* tradicionais podem não ser as mais adequadas para outras aplicações. Por exemplo, a representação do código de um programa em árvores AST é adequada para compiladores. Porém, como essa estrutura geralmente não reflete diretamente os construtores presentes no nível de programação, possuindo aspectos da gramática da linguagem (que são muitas vezes complexos), a sua utilização em aplicações que apenas analisam o código de uma linguagem nem sempre é trivial (Badros, 2000). Algumas dessas aplicações optam, então, por fazer uma simples análise léxica do código ao invés de reusar um *parser* e analisar uma árvore AST. Assim, cada uma dessas aplicações apresenta uma implementação proprietária para o processamento do código fonte, impedindo, por exemplo, o reuso e o intercâmbio.

Para solucionar esse problema e permitir que ferramentas de software possam facilmente analisar e manipular códigos fonte, é necessário um formato "universal" para representar a estrutura de um programa. O padrão XML, por possuir os requisitos necessários para a definição dessa estrutura, foi utilizado na

proposta de uma linguagem baseada em XML para descrever o código (ou para representar a estrutura) de programas Java: JavaML (Badros, 2000).

Ao utilizar XML, a representação de um código fonte deixa de ser simplesmente baseada em caracteres e pode refletir mais diretamente a estrutura do programa. Uma abordagem para definição de JavaML poderia utilizar um mapeamento direto de AST para XML, mas, como comentado, a estrutura AST apresenta diversos detalhes da gramática da linguagem que iriam gerar uma representação XML muito complexa. Outra abordagem, mais simples, seria apenas adicionar marcações XML ao código original. Nesse caso, tem-se uma agregação de valor ao código, que pode ser útil em diversas aplicações[51]. JavaML foi definida estendendo essa última abordagem. Buscando aproveitar as características principais de XML, a definição da linguagem JavaML procurou modelar os construtores de uma linguagem de programação independente de sua sintaxe, utilizando da melhor forma possível os elementos e atributos XML para representação de *conceitos* do código. A Figura 6:6 ilustra um código para uma classe em Java, cuja representação em JavaML encontra-se na Figura 6:7.

```
01  import java.applet.*;
02  import java.awt.*;
03  public class FirstApplet extends Applet {
04    public void paint(Graphics g) {
05      g.drawString("FirstApplet", 25, 50);
06    }
07  }
```

Figura 6:6 - Especificação da classe *FirstApplet* em Java.

```
01   <java-source-program name="FirstApplet.java">
02    <import module="java.applet.*"/>
03    <import module="java.awt.*"/>
04    <class name="FirstApplet" visibility="public">
05     <superclass class="Applet"/>
06     <method name="paint" visibility="public" id="meth-15">
07      <type name="void" primitive="true"/>
08      <formal-arguments>
09       <formal-argument name="g" id="frmarg-13">
10         <type name="Graphics"/></formal-argument>
11      </formal-arguments>
12      <block>
13       <send message="drawString">
14        <target><var-ref name="g" idref="frmarg-13"/></target>
```

[51] Exemplos de documentos XML nessas duas abordagens, assim como uma análise de suas vantagens e desvantagens, podem ser encontrados em (Badros, 2000).

```
15        <arguments>
16         <literal-string value="FirstApplet"/>
17         <literal-number kind="integer" value="25"/>
18         <literal-number kind="integer" value="50"/>
19        </arguments>
20       </send>
21      </block>
22     </method>
23    </class>
24   </java-source-program>
```

Figura 6:7 - Especificação de *FirstApplet* em JavaML.

Como ilustrado pelo exemplo, arquivos em JavaML, ao mesmo tempo que podem ser utilizados pelas mais diversas ferramentas de análise, mantêm o código legível para programadores. A conversão entre classes Java e arquivos JavaML foi implementada a partir do compilador Jikes (IBM, 1998) para Java; uma folha de estilo XSLT permite a conversão reversa.

A grande vantagem de JavaML é permitir que as diversas aplicações que a utilizem usufruam das ferramentas e padrões XML existentes. Transformadas XSLT podem ser aplicadas a documentos JavaML para, por exemplo, alterar nomes de métodos de forma precisa (editores de texto simples não conseguem distinguir entre um nome que representa um método e uma ocorrência desse nome em outro contexto), e padrões como XSL (W3C, 2001a) podem ser utilizados para a apresentação dos dados XML. A abordagem utilizada em JavaML pode ser seguida para descrever, em XML, códigos fonte de outras linguagens. Assim, transformadas nesses arquivos XML podem ser suficientes para permitir o intercâmbio de códigos fonte entre linguagens de programação.

A definição de JavaML mostra que, para se obter as vantagens de XML, nem sempre existe um mapeamento direto entre linguagens de programação e suas representações em XML. O código fonte ou árvores AST não podem ser utilizados diretamente para gerar essa representação, indicando que as tecnologias utilizadas pelos *frameworks* para compiladores de linguagens de programação ainda precisam ser melhor examinadas para sua aplicação no domínio XML. Dessa forma, ainda é necessário analisar como essas tecnologias, geralmente complexas, podem ser utilizadas para refinar o *meta-framework*, mantendo uma de suas principais características: a simplicidade.

6.4. Compiladores XML

Para facilitar o processamento de dados em aplicações baseadas em XML, em (Li et al., 2003) é proposta uma arquitetura extensível e integrada chamada *XML Virtual Machine* (XVM). Essa arquitetura é modular e baseada em componentes, oferecendo um *framework* para desenvolvimento de aplicações. XVM considera elementos XML como objetos, cujo comportamento (*behavior*) é determinado por um componente de software a ele mapeado. A Figura 6:8 ilustra a abordagem proposta. O nome *XML Virtual Machine* foi utilizado por considerar dados XML como "instruções para processamento", e componentes como "interpretadores dessas instruções".

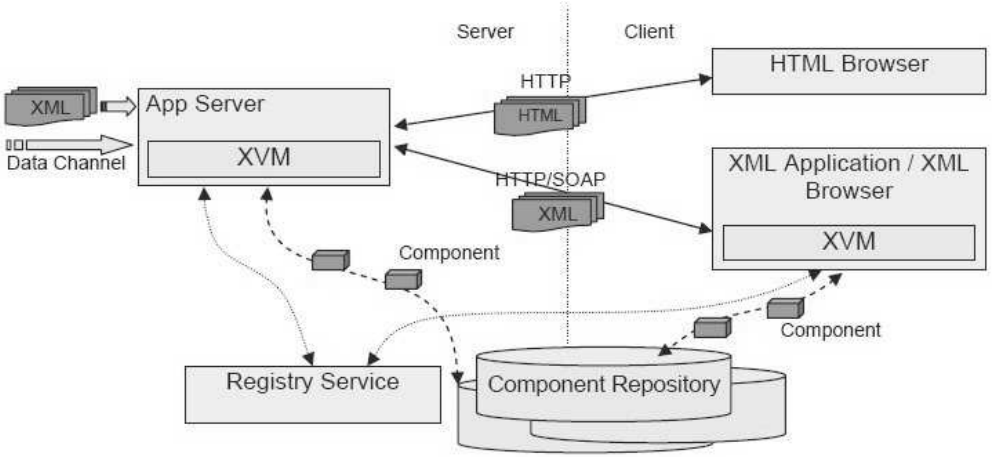

Figura 6:8 - *Framework* XVM para o desenvolvimento de aplicações com XML.

Aplicações XVM obtêm componentes dinamicamente a partir de um repositório (*Component Repository*). Um serviço de registro (*Registry Service*) é oferecido para localização de componentes, mapeando elementos XML em componentes. Utilizando esse registro, XVM associa automaticamente os elementos de uma árvore DOM a componentes, gerando uma estrutura, também em árvore, desses componentes, que determinam a semântica do documento.

Para facilitar o desenvolvimento de aplicações em diversos contextos, XVM permite que o processamento de elementos XML pelos componentes ocorra tanto no cliente (*XML Browser*) quanto no servidor (*App server*), e que um único componente seja associado a uma sub-árvore da árvore DOM (estendendo o mapeamento *um para um* entre elementos e componentes). Outra facilidade de XVM é um mecanismo para que diferentes componentes sejam associados ao mesmo elemento, dependendo do *modo de processamento*. Assim, aplicações

distintas podem obter componentes distintos ao processar o mesmo documento XML. A arquitetura simplifica a adição de novos componentes (oferecendo um mecanismo simples para atualização das aplicações) e permite que o mesmo componente seja reutilizado por diferentes aplicações.

XVM é uma arquitetura para o tratamento de documentos XML. A principal semelhança com o *meta-framework* é delegar o tratamento de elementos XML para outras entidades da "arquitetura". No *meta-framework*, essas entidades são representadas pelas classes dos compiladores, enquanto que em XVM, são representadas por componentes. Embora XVM, diferente do *meta-framework*, não tenha como foco o processamento de documentos XML como um todo, sua abordagem oferece um ponto de partida[52] para que *frameworks* de compiladores sejam utilizados na compilação de documentos XML em um ambiente distribuído, possivelmente explorando recursos de paralelismo.

6.5. Extensões à SMIL

Em (King et al., 2004) é descrita uma extensão à SMIL para permitir a definição de relacionamentos envolvendo entradas que variam constantemente com o tempo, ou seja, que possuem comportamento ("*behavior*") dinâmico em tempo-real. Relacionamentos podem ser especificados utilizando *eventos dinâmicos* ou *dependência contínua de tempo-real*. Um *comportamento* dinâmico pode gerar uma seqüência de *eventos dinâmicos*. Esses eventos são discretos e permitem disparar alguma ação ao se detectar que, por exemplo, um valor ultrapassa um certo limite. *Dependência contínua de tempo-real* considera uma propriedade *p*, de um item *A*, que depende do valor de propriedades dinâmicas de outros itens. Assim, *p* deve ser continuamente re-avaliada refletindo na exibição de *A*. Por exemplo, um objeto deve se movimentar em direção a um alvo, sendo que esse alvo também está em movimento.

Para permitir que relacionamentos como os descritos acima possam ser especificados, são necessárias duas extensões à SMIL: a habilidade de *avaliação de expressões* e um mecanismo de *definição de eventos*. Como o foco da extensão

[52] O ponto de partida se baseia na analogia entre os *componentes* XVM e as *classes* dos compiladores.

é o módulo de animação de SMIL, essas expressões podem ser aplicadas aos atributos utilizados para definição de animações - como *from*, *to*, *by*, *values* e *path* (W3C, 2001b) - e avaliadas por meio do prefixo *calc*.

Em (Goose et al., 2002) é proposta uma outra extensão para a linguagem SMIL que permite a definição e apresentação de áudios em três dimensões. A definição para o *layout* do documento é estendida, passando a ser especificada em um espaço de três dimensões (3D) ao invés de duas dimensões (2D), o que permite a definição de posições e trajetórias 3D. Baseado no elemento *regPoint* de SMIL (W3C, 2001b), foi definido o elemento *regPoint3d*, cujos atributos identificam a posição no espaço de coordenadas 3D onde objetos de áudio podem estar localizados. A posição de um objeto no espaço 3D também pode ser determinada matematicamente, pelo uso do elemento *trajectory*. Esse elemento, adicionado à SMIL, oferece um mecanismo genérico para especificar uma equação parametrizada para cada dimensão. Os atributos do elemento *trajectory* permitem que variáveis (*variables*) sejam aplicadas a uma expressão (*expression*), calculada utilizando os valores (*values*) correspondentes.

Para animação de áudios 3D, foi estendido o elemento *animate* de SMIL, que deu origem ao elemento *animateLoc*. Esse elemento especifica a animação de um objeto da posição referenciada pelo atributo *from* para a posição referenciada pelo atributo *to*. Para permitir a definição de áudios sintetizados em SMIL, foi definido um novo elemento: *tts*. Esse elemento é similar ao elemento *audio* de SMIL e possui outros atributos, como *voice* (que especifica qual padrão de voz a ser utilizado pelo sintetizador). A proposta foi validada a partir de uma arquitetura cliente/servidor na qual o servidor adapta um documento SMIL 3D gerando um documento SMIL. Os áudios 3D são convertidos em áudios estéreos, que simulam as especificações 3D do documento SMIL 3D.

Ambas as propostas apresentadas para extensão à SMIL abordam casos particulares que requerem alguma funcionalidade não oferecida pela linguagem. A linguagem X-SMIL, descrita nesta dissertação, estende SMIL em sua principal característica: a sincronização de objetos multimídia. Conectores hipermídia podem ser usados na especificação dos diversos relacionamentos descritos em (King et al., 2004), desde que ferramentas de exibição (Rodrigues, 2003) sinalizem as mudanças dos atributos e eventos referenciados pelos conectores.

É importante destacar, também, que a especificação de *layout* 2D em NCL, por exemplo, pode ser facilmente alterada para uma representação 3D. Finalmente, como foi exemplificado no Capítulo 3 (Seção 3.3), o elemento *tts*, introduzido pela proposta (Goose et al., 2002), torna-se desnecessário ao se utilizar o conceito de descritores.

6.6. Conversão entre Modelos

Yang e Yang (Yang & Yang, 2003) propõem uma ferramenta gráfica para autoria de documentos hipermídia baseada em SMIL: SMILAuthor. O processo de autoria é dividido em três partes. Primeiramente um documento SMIL é importado para o ambiente de autoria. Em seguida, são oferecidas funções de edição (como copiar, recortar e colar). Finalmente, pode-se salvar o resultado da edição em um documento SMIL. A principal diferença entre essa ferramenta e outras baseadas em SMIL (Coelho, 2004) é a utilização de um modelo em *timeline* para edição, ao invés de utilizar a própria estrutura de SMIL (composições com semântica temporal etc.). Dessa forma, no primeiro passo é aplicado um algoritmo para conversão de um documento SMIL em um modelo *timeline*, no qual todas as funções de edição podem ser aplicadas.

Segundo os autores da proposta, funções como copiar e colar são mais facilmente realizadas em uma visão baseada em *timeline*. A Figura 6:9, por exemplo, ilustra como um objeto (*object* - item *a* na figura) ou um grupo de objetos (*time zone* - item *b* da figura), que foram previamente recortados, são inseridos (colados) na linha de tempo (*timeline*).

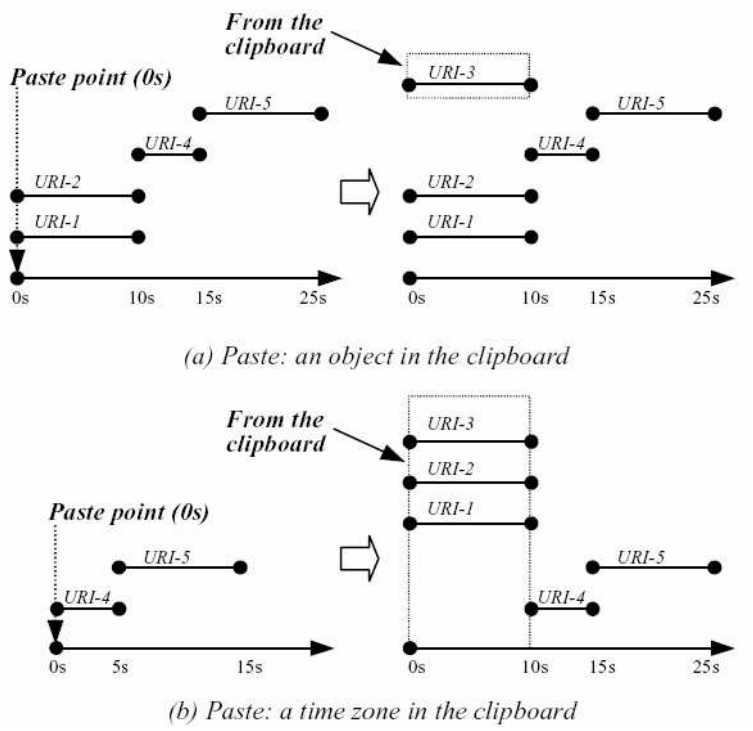

Figura 6:9 - Edição de um documento em *timeline*.

Após a edição, a estrutura em *timeline* deve ser convertida para um documento SMIL. A solução trivial para essa conversão é agrupar todos os objetos em um contêiner *par* e utilizar atributos *begin* e *dur* para refletir o comportamento temporal desses objetos em SMIL. Entretanto, esse algoritmo é apontado como problemático, já que *players* SMIL devem tratar simultaneamente todos os elementos filhos de contêineres *par*. Por isso, buscou-se maximizar o número de contêineres do tipo *seq* no documento SMIL. Assim, é utilizado um algoritmo que percorre a lista de objetos, gerando grupos de objetos com apresentação disjunta. Esses grupos de objetos geram, então, contêineres *seq*, filhos de um contêiner *par*. Esse algoritmo considera que objetos do mesmo tipo de mídia têm maior probabilidade de não serem apresentados em paralelo, sendo o mesmo verdade para objetos que referenciam a mesma região do *layout.*

Os problemas e possíveis soluções descritos pela proposta na geração de documentos SMIL, a partir de outro modelo, são sua principal contribuição. A principal limitação da proposta é a abordagem adotada para a sincronização temporal entre objetos de mídia (sincronização inter-mídia), que é baseada em *timeline*: a sincronização é realizada através do posicionamento dos componentes de uma apresentação em relação a um eixo do tempo. Essa abordagem apresenta

uma série de problemas, como a impossibilidade de modularizar uma apresentação e a dificuldade de se especificar requisitos temporais entre eventos cuja duração é variável, ou desconhecida, até o momento da execução.

Essas limitações não existem em NCL ou SMIL, que adotam um paradigma baseado em restrição e causalidade (*constraint-and-causal-based*) para sincronização. Como visto, é possível passar de um paradigma de sincronização para outro, mas, às vezes, há perda de expressividade. Isso se deve ao fato dos diversos paradigmas terem poderes de expressividade diferentes. Além de não possuir as limitações de *timeline*, o paradigma de restrição e causalidade tem maior poder de expressão (FLEXTV, 2004a).

O compilador NCL-SMIL apresenta alguns dos problemas descritos na conversão entre *timeline* e SMIL, como o número de composições *par* que são geradas. Porém, as limitações do paradigma *timeline* implicam em uma perda de informações na seqüência de conversão SMIL-*timeline* e *timeline*-SMIL. Essas informações (por exemplo: o término do objeto A gera o término do objeto B) são preservadas na conversão NCL-SMIL. Além disso, o poder de expressão dos modelos baseados em restrição e causalidade se reflete na maior complexidade do compilador NCL-SMIL.