4
Implementação

O *Composer* é um ambiente de autoria implementado em Java, que visa auxiliar autores com diferentes níveis de conhecimento na construção de documentos NCL nas suas *profiles* para TV digital (NCL 3.0).

Neste capítulo são apresentados detalhes de implementação das melhorias e inovações do ambiente de autoria *Composer*, mostrando, assim, o que este trabalho tem a acrescentar com relação ao editor HyperProp. Cabe ressaltar que a implementação do editor HyperProp foi utilizada como base para o desenvolvimento do *Composer*.

4.1.
Interface Gráfica

O ambiente de autoria *Composer* (classe *ComposerEditor*) é composto por uma barra de menus (classe *EnhancedMenu*) e por uma barra de ferramentas (classe *ToolBar*), conforme apresentado no diagrama de classes da Figura 18.

A barra de menus pode possuir itens de menu simples (classe *EnhancedMenuItem*) e/ou caixas de seleção (classe *EnhancedCheckBoxMenuItem*). Já a barra de ferramentas é composta por botões (classe *EnhancedButton*). Detalhes do funcionamento das classes com prefixo *Enhacend* serão apresentados na Subseção 4.1.2. No editor HyperProp, cada visão possui uma barra de menus e uma barra de ferramentas. No *Composer*, essas barras foram retiradas das janelas das visões, sendo concentradas em uma única barra. Assim, funcionalidades comuns às visões, como por exemplo, *zoom in* e *zoom out*, são ativadas de acordo com a visão que está em foco.

Na atual implementação do *Composer* é possível que o autor trabalhe com mais de um documento NCL ao mesmo tempo, diferentemente do editor HyperProp (Coelho & Soares, 2004). Para isso, o *Composer* trabalha com a abstração de projeto (classe *IProject*), que pode conter um ou mais documentos NCL (classe *EditorDocument*). Visando permitir que projetos criados em versões

mais antigas possam ser utilizados em versões mais recentes, cada versão do *Composer* deve, se necessário, implementar de forma diferente seus projetos (classe *ProjectImpl*). Nesse caso, a idéia é manter a implementação da versão antiga, por exemplo, *ProjectImpl_1* do *Composer* versão 1.0, e acrescentar a implementação da nova versão, por exemplo, *ProjectImpl_2* para o *Composer* versão 2.0. Desta forma, projetos da versão 1.0 poderiam ser abertos na versão 2.0 do *Composer* utilizando a API da classe *IProject*.

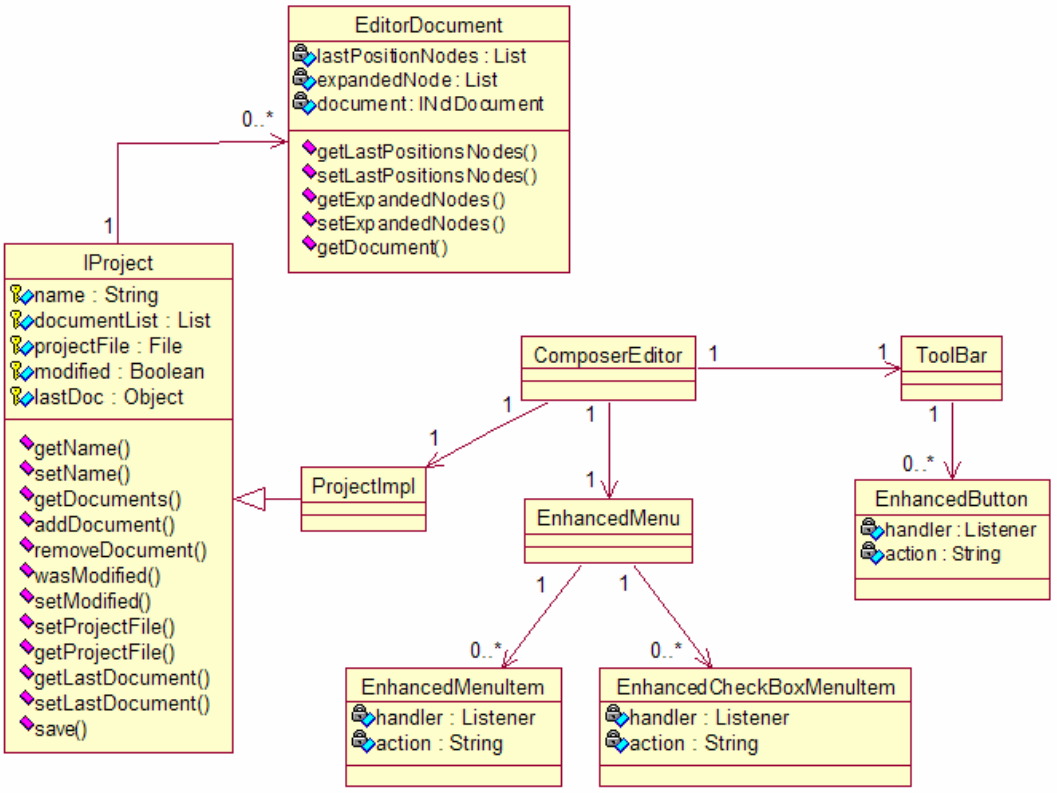

Figura 18 - Diagrama de classes da interface gráfica do *Composer*

A Figura 19 apresenta a janela principal do *Composer*. Nessa figura, a região 1 (um) exibe um projeto e os documentos NCL que o compõem, na forma de uma árvore. A região 2 (dois) é a área de trabalho onde um documento pode ser visualizado e editado através das visões. Já a região 3 (três), não existente no editor HyperProp, é uma área multifuncional que serve para dar suporte ao autor na construção de seus documentos NCL. Essa região é usada, por exemplo, para notificação de erros. O acesso aos recursos da área multifuncional é feito através de abas. Detalhes da utilização dessa área serão apresentados mais adiante.

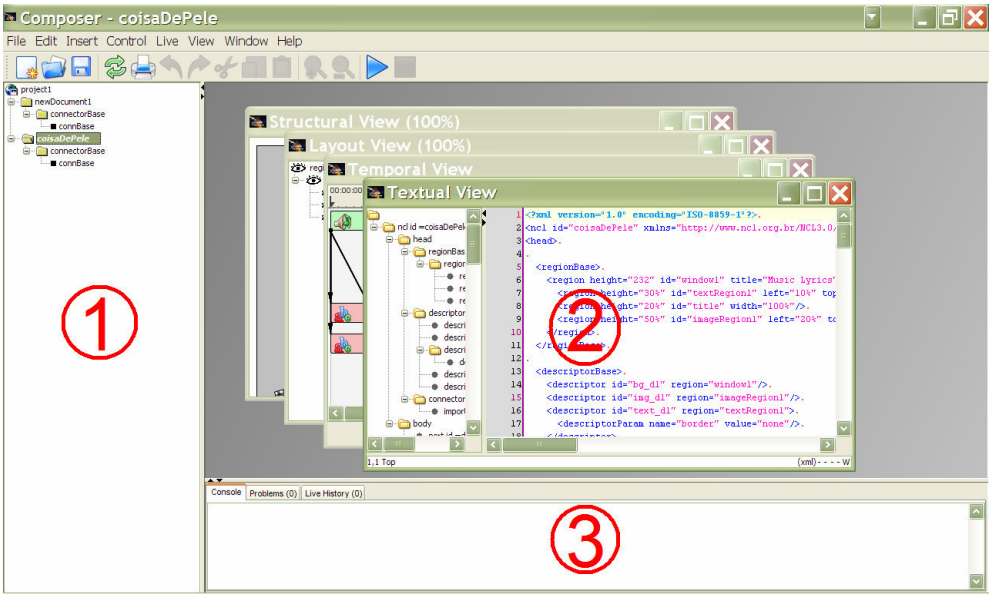

Figura 19 - Interface gráfica do *Composer*

A Figura 20 apresenta a barra de menus e a barra de ferramentas da interface gráfica do *Composer*. A barra de menus contém várias funcionalidades para manipulação de projetos e para criação e edição de documentos NCL. Já a barra de ferramentas é composta por algumas funcionalidades comumente usadas, como, por exemplo, criar projeto, abrir projeto e salvar projeto.

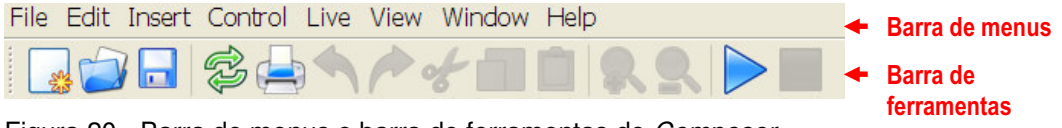

Figura 20 - Barra de menus e barra de ferramentas do *Composer*

O restante desta seção continua destacando outras melhorias e inovações implementadas na interface gráfica do *Composer*.

4.1.1. Módulo de Localização

Por opção de implementação, todos os textos exibidos na interface gráfica do editor HyperProp se encontram dentro do código fonte, vinculados diretamente aos elementos de interface. Dessa forma, qualquer modificação nesses textos implica fatalmente a alteração do código. Isso, além de prejudicar a manutenibilidade e o reuso, dificulta bastante o suporte a vários idiomas.

Para contornar esses problemas, no *Composer* foi criado o módulo de localização. Na nova implementação, os itens de interface gráfica deixaram de referenciar diretamente o texto a ser exibido, e passaram a utilizar uma chave (*string*) para obter esse texto junto ao módulo de localização. A utilização dessas

chaves abstrai o vínculo existente entre os itens de interface e os textos localizados, permitindo que a manutenção desses textos seja feita sem ser preciso alterar o código fonte.

Para cada idioma suportado é definido um arquivo (*properties*) contendo as chaves e seus respectivos valores (textos). Cabe ressaltar que os arquivos de *properties* devem possuir o mesmo conjunto de chaves, sendo que o que muda é o valor para cada chave, ou seja, os textos localizados. Assim, para um botão de confirmação, que referencia a chave *button.yes*, por exemplo, poderíamos ter definido *button.yes=Yes* no arquivo de *properties* para o idioma inglês, e *button.yes=Sim* no arquivo para português. Escolhido um idioma, o módulo de localização simplesmente busca o valor da chave dentro do arquivo de *properties* apropriado.

Além de fornecer o valor de uma chave, o módulo de localização também permite a alteração desse valor durante a execução do *Composer*. Isso é útil, por exemplo, para guardar o estado de um item de menu que fica habilitado ou não, de acordo com mudanças de propriedades dentro de uma visão. Cabe ressaltar que os valores modificados só têm validade durante a execução, não sendo persistidos ao sair da ferramenta.

Na atual implementação é possível a troca entre os idiomas inglês e português em tempo de execução.

4.1.2. Enhanced Items

Durante a implementação do *Composer*, alguns componentes foram desenvolvidos com o objetivo de tornar a criação e manutenção da interface gráfica mais simples.

Esse conjunto de elementos personalizados, chamados *Enhanced Items*, é composto por botões, menus e itens de menu. Todos eles estendem as respectivas classes oferecidas pela API Java, mas possuem um processo de configuração diferenciado.

Na criação de um *Enhanced Item* é passada uma chave (*string*) referente à ação que o elemento irá desempenhar. Para criar o botão de abrir projeto da barra de ferramentas (Figura 20), por exemplo, é passada a chave *openProject*. De posse

dessa chave, o construtor do *Enhanced Item* lança mão do módulo de localização para buscar outras informações importantes, como por exemplo, o ícone associado àquela chave, o atalho do teclado, o mnemônico e se o item deve iniciar habilitado ou não. No caso de itens de menu que são selecionáveis (caixas de seleção), no construtor ainda é conferido se o item deve começar selecionado ou não.

Na Figura 21 é apresentado um exemplo com informações que poderiam ser utilizadas para criar um botão que executa a ação de abrir projeto (chave *openProject*). Neste exemplo, essas informações estariam contidas no arquivo de *properties* do idioma inglês.

```
...
label.openProject=$Open Project...
icon.openProject=editorConfig/images/open.png
shortcut.openProject=Ctrl+O
enabled.openProject=true
...
```

Figura 21 - Configuração de um *Enhanced Item*

Na figura acima, *label.openProject* especifica o texto que é exibido como *tooltip* (dica) do botão criado. Ainda com relação a essa chave, a primeira letra após o símbolo *$* é adotada como mnemônico, no caso de o *Enhanced Item* ser um item de menu. Já a chave *icon.openProject* especifica o ícone a ser utilizado no botão, enquanto que a chave *shortcut.openProject* define a tecla de atalho que pode ser utilizada para executar a ação. Por fim, a chave *enabled.openProject* especifica se o botão deve ser ou não habilitado na barra de ferramentas. Se alguma dessas opções não estiver definida no arquivo de *properties*, um valor padrão é utilizado.

Cabe destacar que no construtor do *Enhanced Item*, além da chave relativa à ação a ser desempenhada, é preciso fornecer um observador (*listener*), que é responsável por monitorar e tratar o evento de acionamento desse elemento de interface. Por fim, é importante ressaltar que os *Enhanced Items* foram vislumbrados após a análise do código fonte da ferramenta LimSee2 (WAM, 2007), a qual usa recursos muito semelhantes para configuração da sua interface gráfica.

4.2. Suporte ao Usuário

Esta seção apresenta detalhes das melhorias e inovações referentes a suporte ao usuário que foram implementadas neste trabalho.

4.2.1. Módulo de Preferências

O módulo de preferências é uma novidade do *Composer* em relação ao editor HyperProp, e tem como objetivo capturar algumas propriedades da interface gráfica modificadas pelo autor. Exemplos das propriedades monitoradas são: as últimas visões ativas, as dimensões e posicionamento de cada visão, o posicionamento da barra de tarefas, e o tamanho da janela principal.

Sempre que o usuário encerra o *Composer*, todas essas propriedades são salvas em um arquivo (*properties*). Se, ao iniciar o ambiente de autoria, o arquivo de *properties* estiver em um local pré-estabelecido, as propriedades monitoradas são restabelecidas com os valores da última sessão. Caso esse arquivo não exista, as propriedades de interface são configuradas com seus valores *default* (padrão).

O mecanismo utilizado para obter os valores das propriedades de interface é similar ao adotado no módulo de localização. Assim, dentro do código fonte os elementos de interface fazem referência a chaves, e buscam os valores delas junto ao módulo de preferências. A grande diferença com relação ao módulo de localização fica por conta da persistência.

Ainda cabe ressaltar que, além de guardar propriedades de interface, o módulo de preferências também armazena a lista dos últimos projetos utilizados. Essa lista, exibida sob um menu da barra de menus, permite o acesso rápido ao projeto selecionado. Sempre que um novo projeto é aberto ou criado, essa lista é devidamente atualizada.

4.2.2. Módulo de Mensagens

Um problema sério existente no ambiente HyperProp diz respeito às mensagens de compilação, que são pouco descritivas. Além disso, a maior parte dessas mensagens só é impressa no *console* de execução. Tal comportamento demanda que o *console* seja verificado constantemente, e pode deixar o autor perdido caso haja algum erro no documento.

No *Composer*, todas as mensagens de compilação são exibidas em uma aba (*Problems*) da área multifuncional. As mensagens podem ser de erro, de aviso ou

informativas. Uma mensagem pode especificar ainda o tipo da entidade da NCL à qual faz referência (por exemplo, uma região ou uma âncora), o *id* (identificador) dessa entidade, a descrição da mensagem de compilação, e a qual arquivo a mensagem está ligada.

O duplo clique em uma mensagem que tenha o campo *id* definido coloca a visão textual em foco (se ela já não estiver), e seleciona a linha na qual o elemento com o *id* informado está. Isso facilita a detecção e a solução de um possível problema de compilação.

4.2.3. Módulo de *log*

O módulo de *log* (ou registro) tem por objetivo fornecer suporte adicional no processo de autoria e no funcionamento do *Composer*. Isso inclui capturar as mensagens endereçadas à saída padrão do sistema, como, por exemplo, chamadas explícitas feitas pelo programador no código fonte, ou mesmo exceções geradas pelo mau funcionamento do *Composer*.

Essas mensagens são sempre impressas em uma aba (*Console*) da área multifuncional. Por uma opção de implementação, mensagens de erro são impressas em vermelho, enquanto que mensagens informativas são impressas em verde, e mensagens de aviso são impressas em amarelo. As chamadas dentro do código fonte podem ser feitas utilizando diretamente a API do Java, ou então a classe que implementa o módulo de *log*.

Além de direcionar todas as mensagens da saída padrão para uma região da interface gráfica, o módulo de *log* as guarda em um arquivo (*log*). Isso pode ser bastante útil para a notificação e rastreamento de *bugs*, uma vez que esse arquivo pode ser enviado à equipe de suporte do *Composer* para aperfeiçoamento da ferramenta.

4.2.4. Módulo de Ajuda

No *Composer*, o módulo de ajuda permite ao autor consultar conceitos relativos à linguagem NCL e funcionalidades do ambiente de autoria. O conteúdo

exibido pelo módulo de ajuda foi elaborado de modo que o autor possa fazer um melhor uso e tirar melhor proveito de todas funcionalidades do *Composer*[8].

Outra característica importante é que o módulo de ajuda identifica o contexto do autor. Se, por exemplo, o usuário pressionar a tecla *F1*, a janela de ajuda é exibida em um tópico relativo à parte da interface gráfica em foco naquele instante. Caso não tenha sido especificado um tópico relacionado àquela parte da interface, a janela de ajuda é aberta em uma página padrão.

Para a implementação do módulo de ajuda foi utilizada a tecnologia JavaHelp (Sun, 2003), o que facilitou a integração com o *Composer*.

4.3. Visões

Conforme discutido anteriormente, no *Composer* as abstrações são definidas nos diversos tipos de visões que permitem simular um tipo específico de edição. Nas próximas subseções serão apresentadas as visões do *Composer*, bem como suas principais características.

4.3.1. Visão estrutural

Praticamente toda visão estrutural foi remodelada. No editor HyperProp, sempre que dois vértices são unidos por mais de uma aresta, uma única aresta é exibida com um rótulo identificando a quantidade de arestas entre esses vértices. No *Composer*, essa representação foi alterada, de modo a desenhar arestas paralelas, conforme pode ser visto na Figura 22. Isso foi feito com o objetivo de dar ao autor uma melhor percepção de como o documento está estruturado.

[8] O conteúdo do módulo de ajuda foi elaborado em conjunto com o laboratório SERG/PUC-Rio (http://www.serg.inf.puc-rio.br).

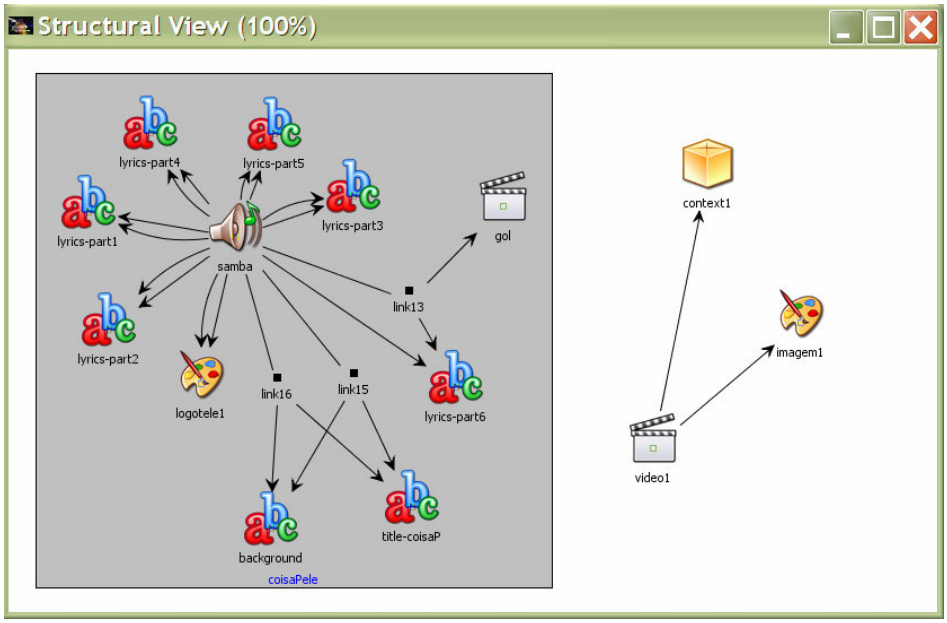

Figura 22 - Visão estrutural do *Composer*

A atual visão estrutural continua não representando as portas existentes nas composições. Contudo, para cada nó de mídia, que é porta do documento, é atribuído um outro ícone, diferenciando-o dos demais nós de mídia do mesmo tipo.

Operações de manipulação do grafo continuam sendo permitidas, tais como, expandir e colapsar composições. As operações de adicionar, editar e remover nós e elos também são permitidas, sendo que nesse caso, as alterações realizadas por essas operações são refletidas nas demais visões do *Composer*. A maior parte das operações pode ser acessada utilizando o mouse, teclado ou menus. Uma novidade fica por conta da possibilidade de criação de elos através do arraste do mouse de um elemento origem até um elemento destino. Essa operação está habilitada somente para relacionamentos um para um e para nós de mídia no mesmo contexto, sendo acionada através do arraste do mouse com a tecla *Ctrl* pressionada do centro de um nó de mídia para outro. Quando isso é feito, é aberta uma caixa de diálogo para criação do elo. Cabe lembrar que para o perfil de TV digital, somente os elos causais estão sendo considerados.

Conforme proposto em (Coelho & Soares, 2004), a visão estrutural do *Composer* foi implementada utilizando como base a biblioteca JGraph (Alder, 2003), para representação do documento NCL como um grafo. O JGraph trabalha com o padrão de projeto MVC (*Model-View-Controller*) (Gamma et al., 1997), onde o "modelo" é a implementação das entidades da linguagem NCL, a "visão"

são os objetos responsáveis pela apresentação do grafo para o usuário, e o "controlador" são os objetos que definem como alterações na visão são refletidas no modelo, e vice-versa.

No HyperProp, sempre que uma das visões, que não a estrutural, é atualizada, o grafo da visão estrutural é reorganizado, de modo que toda a disposição anterior dos nós é perdida. No *Composer*, sempre que uma visão gráfica é atualizada, ou mesmo quando o autor muda de um documento para outro ou fecha um projeto, as posições dos nós e o estado (expandido/colapsado) das composições são guardados. Dessa forma, ao atualizar ou ao voltar a trabalhar com um determinado documento, os nós de mídia e composições do grafo permanecerão no mesmo estado em que estavam antes.

Ademais, visando melhorar a representação gráfica, sempre que possível, a visão estrutural tenta evitar que nós (vértices) do grafo sejam sobrepostos. Para isso, quando um nó é movimentado, se houver colisão com outros nós, é executado um método iterativo que tem por objetivo repelir os nós afetados. Esse método tem um número máximo de passos pelo fato de que em determinadas ocasiões, o grafo pode ser tão denso que ao repelir os nós sobrepostos, esses podem sobrepor outros nós, e o processo pode continuar indefinidamente.

Atualmente, a visão estrutural do *Composer* trabalha apenas com o algoritmo *Spring* (Kamada & Kawai, 1989) para disposição de grafos. Como trabalho futuro, outros algoritmos podem ser acrescentados à ferramenta.

4.3.2. Visão de leiaute

A nova visão de leiaute permite ao autor movimentar livremente uma região filha para fora da região definida como seu pai. Nessa situação, somente a interseção da região filha com a região pai é pintada, enquanto que a área disjunta, não. Se a área da região filha estiver totalmente disjunta com relação à área de seu pai, somente os contornos da região filha são desenhados. Isso vale também para a região pai com relação a seu primeiro ascendente, e assim por diante. Seguindo esse raciocínio, conclui-se que somente aparecerá pintada a interseção de uma região com todos os seus ascendentes. Isso não significa que em tempo de apresentação aparecerá somente parte da mídia que usa aquela região. Na verdade,

a exibição da mídia ligada a uma determinada região vai depender da estratégia adotada na implementação do formatador.

A Figura 23 apresenta a visão de leiaute do *Composer*. Nela, a área à esquerda exibe uma árvore com as regiões definidas pelo autor. Muitas vezes, essa hierarquia não fica explicitada através do desenho da região do lado direito (*Canvas*) da visão de leiaute. Ao selecionar um nó da árvore, a região correspondente é imediatamente selecionada do lado direito.

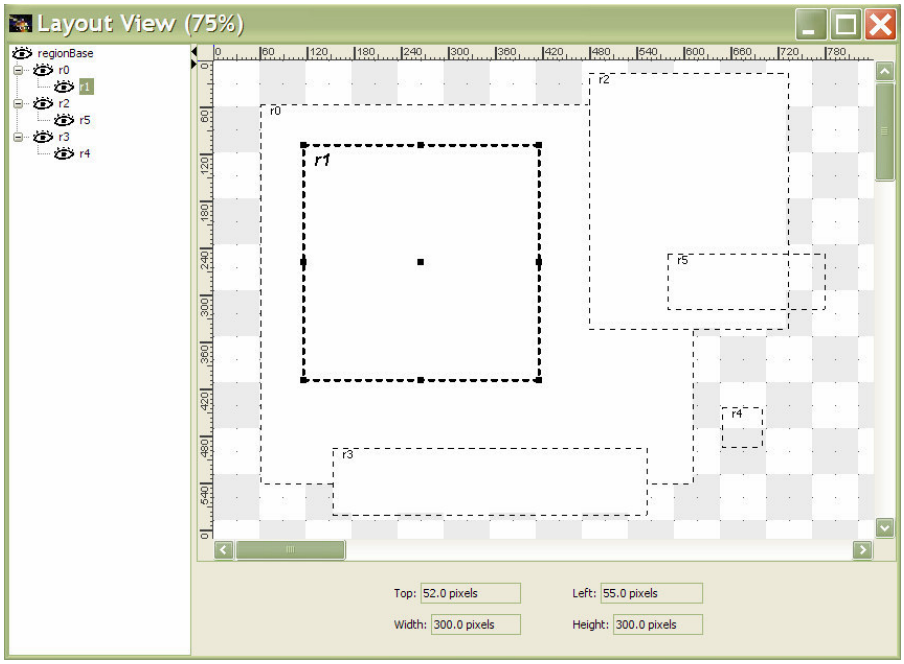

Figura 23 - Visão de leiaute do *Composer*

Outra inovação implementada permite ao autor despoluir a visão de leiaute, exibindo ou não regiões. Para isso, basta clicar sobre a imagem do "olho" correspondente à região de interesse, no lado esquerdo da visão de leiaute. Se o olho referente à região estiver aberto, após o clique com o mouse, ele se fechará e a região deixará de ser exibida do lado direito. Para reverter o processo, basta clicar sobre o olho fechado, e ele se abrirá na árvore à esquerda, sendo a região correspondente novamente exibida do lado direito.

Abaixo do *Canvas* foi inserida uma região que exibe as propriedades *top*, *left*, *width* e *height* de uma região selecionada. Isso é bastante útil, pois informa instantaneamente as propriedades da região quando ela está sendo movimentada. Cabe destacar que as manipulações realizadas pelo autor são devidamente refletidas nas demais visões. Ao *Canvas* da visão de leiaute também foram adicionadas uma régua horizontal e outra vertical (vide Figura 23), que dão uma idéia da ordem de grandeza das regiões (em *pixels*).

Atualmente, as dimensões da visão de leiaute se baseiam na resolução do monitor no qual o *Composer* está sendo executado. Como trabalho futuro, é importante a definição dessas dimensões baseado no dispositivo no qual o documento NCL será exibido, como por exemplo, a resolução da tela de um televisor ou de um celular.

4.3.3. Visão textual

A interface da visão textual do *Composer* é dividida em duas partes, conforme apresentado na Figura 24. A área da esquerda exibe a árvore XML do documento, na qual os elementos compostos podem ser expandidos ou colapsados. Já a área da direita apresenta o código do documento NCL. Se o autor selecionar um elemento na árvore XML, seu correspondente no código NCL é automaticamente selecionado e ganha o foco do lado direito da visão textual.

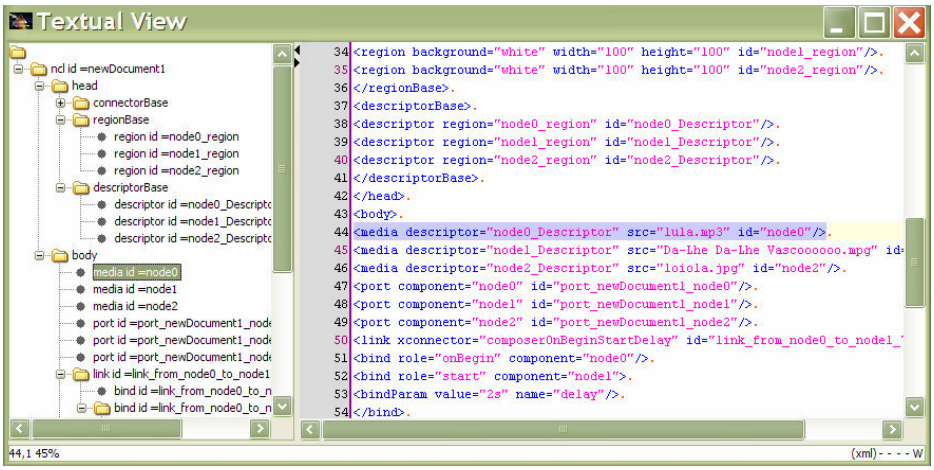

Figura 24 - Visão textual do *Composer*

O texto apresentado no lado direito do editor é exibido com o recurso de destaque (*highlight*), no qual os elementos, seus atributos e valores aparecem em cores diferentes. Alterações feitas no código NCL são refletidas na árvore XML e nas demais visões. Essa sincronização é feita através da tecla *F5* do teclado ou através de uma opção na barra de menus. Para que as modificações sejam devidamente aplicadas, o código do documento precisa ser verificado de acordo com as regras de formação XML e validado com o *schema* da linguagem NCL (Coelho & Soares, 2004). Caso haja algum erro, esse é reportado ao usuário, conforme relatado na subseção referente ao módulo de mensagens.

Se o autor modificar o código NCL e tentar mudar para outra visão ou documento, sem sincronizar, o ambiente de autoria o informa que o código foi alterado e o pergunta que ação deve ser tomada. Se optar por aplicar as atualizações antes de trocar de visão (ou documento), todas as visões são sincronizadas. Caso contrário, as alterações não têm efeito. Essa notificação é uma novidade do *Composer*, e visa auxiliar o autor durante o processo de edição.

Apesar de na época ter sido uma importante inovação, o editor textual do ambiente HyperProp é muito simples, e só dá suporte a funcionalidades básicas, como por exemplo, copiar, colar e recortar texto. Almejando tornar a edição textual mais atraente, a visão textual do *Composer* foi totalmente remodelada. Nesse processo, praticamente toda implementação existente anteriormente foi descontinuada. Essa decisão foi motivada pelo fato de já existirem vários editores código aberto disponíveis na Internet, fazendo com que a implementação de novas funcionalidades na visão textual do editor HyperProp não trouxessem grandes benefícios inovadores.

Entre alguns editores textuais analisados, o JEdit (Pestov, 2007) foi escolhido como base para a atual visão textual. A incorporação desse editor ao ambiente *Composer* demandou um profundo estudo do seu código fonte, e a adaptação desse para atender as necessidades da nova visão textual. Como resultado, com o JEdit vieram vários recursos, como por exemplo, exibição do número das linhas do código, recursos de seleção de texto, recursos de endentação, recursos de autocompletar, opções de cor e fonte do código NCL, recurso de impressão do código, dentre outros.

Seria interessante ainda a implementação de uma funcionalidade para fazer *wrap* (quebra) das linhas de código na visão textual. Isso permitiria que as linhas fossem ou não quebradas de acordo com a preferência do usuário. Além disso, seria útil uma funcionalidade que permitisse colapsar blocos NCL na região à direita (Figura 24), assim como permite o Internet Explorer (Microsoft, 2007c) na exibição de arquivos XML.

4.3.4.
Visão temporal

A Figura 25 apresenta um diagrama das principais classes utilizadas na implementação da visão temporal. Nesse diagrama, a interface *Drawable* especifica que todas as suas subclasses devem possuir o método *draw()*, definindo como desenhar as entidades na escala temporal.

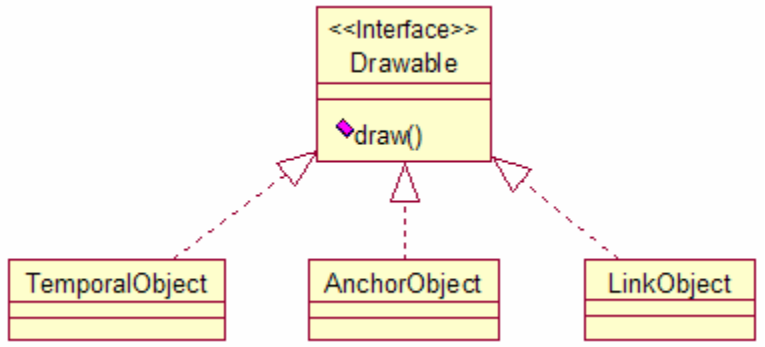

Figura 25 - Diagrama de classes da visão temporal

A conversão de um documento NCL para o modelo de cadeias temporais hipermídia correspondente é feita utilizando um compilador já existente no editor HyperProp. Nesse ponto, o que interessa é que, ao ser gerada uma cadeia, a visão temporal a varre, e instancia de maneira adequada os objetos que representam graficamente as entidades do documento.

Os arcos do modelo de cadeias temporais hipermídia, que correspondem à apresentação dos objetos de mídia, são representados por retângulos preenchidos com uma cor que identifica o tipo da mídia (áudio, texto, vídeo etc.). Além da identificação através de cores, cada retângulo possui um ícone relativo ao tipo da mídia, e um nome (*id*). O comprimento desses retângulos indica suas durações de exibição/execução. Objetos da classe *TemporalObject* (vide Figura 25) são responsáveis por representar graficamente a apresentação de objetos de mídia.

A exibição de âncoras e elos, não contemplada na visão temporal do editor HyperProp, conforme descrito em (Coelho & Soares, 2004), é realizada no *Composer*. As âncoras temporais, representadas por arcos de apresentação no modelo de cadeias temporais, são desenhadas como retângulos tracejados não preenchidos, e ficam dentro do objeto de mídia a que pertencem. As âncoras possuem um ícone de identificação e um *id*. No diagrama da Figura 25, as âncoras são representadas por instâncias da classe *AnchorObject*.

Já as arestas, que representam relações causais no modelo de cadeias temporais hipermídia (elos NCM), são desenhadas como flechas unidirecionais dirigidas, com origem e destino nos retângulos onde os relacionamentos são definidos. A classe *LinkObject* (Figura 25) é responsável pelos elos causais na visão temporal. Com relação aos elos multiponto, na atual implementação, só é dado suporte a elos com um elemento origem (uma condição). Nesse caso, uma flecha unidirecional sai do retângulo origem para cada um dos destinos. A generalização para elos multiponto e o tratamento de auto-elos (entidade de origem igual à de destino) ficam como melhorias futuras.

Além de serem responsáveis pela representação gráfica das entidades do documento, as classes apresentadas na Figura 25 dão suporte à manipulação direta pelo usuário. Com um duplo clique do mouse, por exemplo, é possível editar as propriedades de qualquer uma das entidades da visão temporal.

Para mover uma âncora temporal, o autor deve selecioná-la com o mouse e arrastá-la até a posição desejada. Uma âncora só pode ser movida dentro dos limites do objeto de mídia ao qual pertence. O mesmo vale no redimensionamento. Na implementação atual, âncoras inconsistentes estão sendo tratadas somente no caso de sua duração exceder a duração do objeto de mídia ao qual pertence. Nessa situação, a âncora é desenhada até o limite da duração do objeto de mídia.

O redimensionamento de objetos de mídia é tratado de modo a preservar a consistência das âncoras. Assim, se um objeto for redimensionado, e uma de suas âncoras não couber dentro dele, ela também é redimensionada. Cabe ressaltar que toda manipulação de objetos na visão temporal atualiza os tempos das arestas no modelo de cadeias temporais hipermídia, e conseqüentemente, o documento NCL.

A Figura 26 ilustra a interface gráfica da visão temporal. O número 1 (um) destaca a escala temporal à qual os objetos estão associados. Essa escala é atualizada de acordo com o fator de zoom utilizado. O número 2 (dois) se refere à barra de *status*, que possui campos que informam o nome e os tempos inicial e final de um objeto selecionado. Através dessa barra também é possível escolher a partir de qual porta de entrada o documento será exibido na visão temporal.

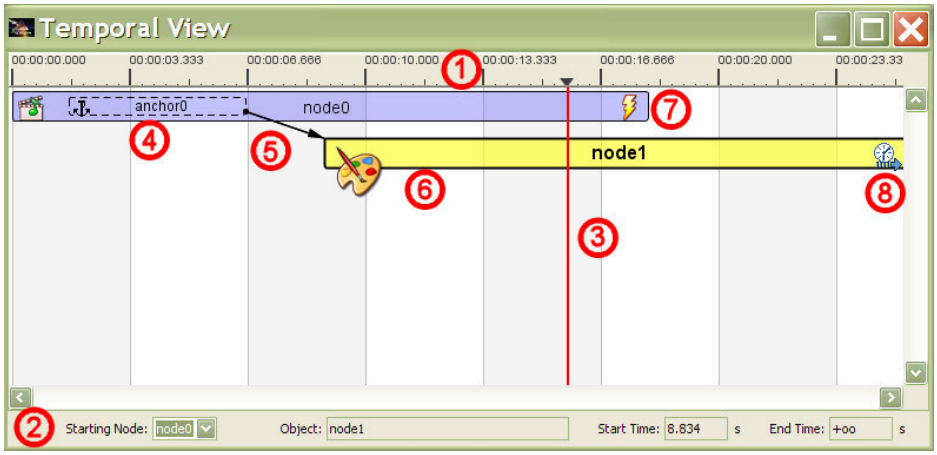

Figura 26 - Visão temporal do *Composer*

Como trabalho futuro, pode ser interessante que um documento seja apresentado a partir de um ponto qualquer, de acordo com o instante definido por uma barra de tempo. Na Figura 26, a barra de tempo da visão temporal é representada pelo número 3 (três).

Ainda com relação à Figura 26, o elemento 4 (quatro) indica a representação gráfica da âncora temporal *anchor0*. O elemento 5 (cinco), por sua vez, corresponde à representação de um elo causal, que relaciona a âncora *anchor0* e o objeto de mídia *node1* (número 6). Os elementos 4, 5 e 6 estão relacionados a instâncias das classes *AnchorObject*, *LinkObject* e *TemporalObject*, respectivamente. Já o número 7 (sete), referente ao ícone do "raio", indica a existência de um relacionamento de interatividade. No caso da Figura 26, a seleção de um determinado botão do controle remoto, durante a apresentação do objeto de mídia *node0*, dispara uma ação, como por exemplo, o término da apresentação do objeto de mídia *node1*. Por fim, o número 8 (oito) destaca o ícone do "relógio" sobre o objeto de mídia *node1*. Esse ícone indica que o objeto de mídia, sobre o qual está, possui duração indeterminada (no exemplo da Figura 26, não foi especificado um relacionamento de término para a imagem *node1*, e uma vez iniciada sua apresentação, essa imagem será exibida até que uma ação externa a termine).

Nas próximas subseções é dado destaque à especificação e edição de relacionamentos através da visão temporal, bem como ao suporte à simulação de eventos interativos.

4.3.4.1. Especificação e edição de relacionamentos temporais

Na visão temporal, a especificação do sincronismo temporal é definida através de relações de alto nível, como por exemplo, “exiba ao início de” e “termine ao recomeço de”. Para isso, a criação de relacionamentos temporais foi dividida em 2 (dois) grupos: o dos relacionamentos de começo/fim e o dos relacionamentos de pausa/recomeço. Para cada um desses grupos foi criada uma caixa de diálogo que permite ao autor especificar relacionamentos temporais entre objetos de mídia.

A Figura 27 ilustra a caixa de dialogo para criação de relacionamentos de começo/fim. Os parâmetros especificados nessa figura informam que o objeto de mídia *video1* será iniciado quando começar (*start*) a âncora *ancora2* do objeto de mídia *audio1*. Analogamente, o objeto *video1* terminará ao fim (*end*) da âncora *ancora2*. Um *delay* (atraso), desde o acontecimento do evento até a execução da ação, poderia ser definido através do campo *Time*. O mesmo raciocínio serve para criação dos relacionamentos de pausa/recomeço.

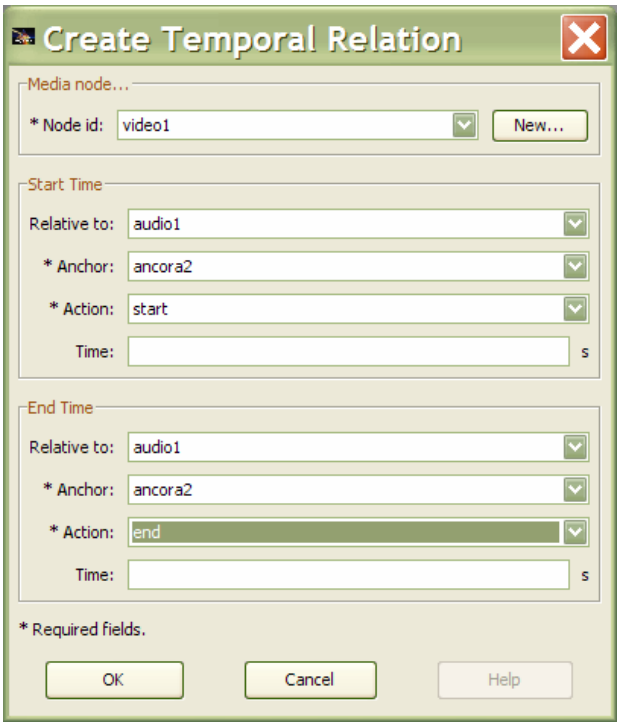

Figura 27 - Especificação de relacionamentos temporais na visão temporal

Ainda com relação à Figura 27, se não fosse especificada nenhuma transição de início, o objeto de mídia seria definido como uma porta do documento. Ademais, caso não seja definida a transição de fim, mas mesmo assim seja

especificado um valor no campo *Time* (caixa *End Time*), esse valor é usado para definir a duração explícita do descritor do objeto de mídia em questão. Pelo fato do campo *Time* poder representar dois conceitos distintos, a interface gráfica dessa caixa de diálogo precisa ser revisada para não confundir o autor.

Na visão temporal, o suporte à especificação de relacionamentos interativos é dado através da caixa de diálogo apresentada na Figura 28. Os parâmetros definidos nessa figura informam que se o telespectador pressionar o botão azul do controle remoto, durante a exibição do objeto de mídia *video1*, o objeto de mídia *ads1* será iniciado. Outras ações, como por exemplo, a ação de pausa, de recomeço ou de fim, também podem ser utilizadas. Uma vez especificado o relacionamento interativo, o objeto de mídia ou âncora, que foi definido como gatilho, nesse exemplo o objeto *video1*, é desenhado com o ícone de interatividade (ícone do "raio" apresentado na Figura 26). A atual implementação só permite a criação de relacionamentos interativos um para um.

Figura 28 - Especificação de relacionamentos interativos na visão temporal

É importante destacar que graficamente o autor pode alterar os relacionamentos temporais entre objetos da visão temporal, desde que seja mantida a consistência do sincronismo. A interface da visão temporal não permite, por exemplo, que um objeto de mídia iniciado por um único elo seja posicionado antes da condição que o dispara.

A análise da manipulação das entidades na visão temporal pode ser dividida em duas partes principais. Na primeira, quando a entidade é movimentada ou redimensionada, todas as outras entidades a ela relacionadas também devem ser devidamente movidas ou redimensionadas, de modo a manter o relativismo dos

relacionamentos. Assim, se o início de uma âncora dispara o início de um vídeo, quando essa ancora é movimentada, o vídeo também deve ser movido junto. Na verdade, o que muda são os tempos de início e/ou fim da apresentação dos objetos dependentes, sendo preservados os relacionamentos.

A segunda parte da análise diz respeito aos relacionamentos que incidem sobre a entidade que está sendo movimentada ou redimensionada. Nesse caso, ao invés das entidades de onde se originam os relacionamentos também serem movidas ou redimensionadas, a estratégia adotada foi a de modificar os relacionamentos, o que parece ser mais intuitivo para o autor.

A alteração de relacionamentos, na verdade, corresponde à alteração do tempo que uma ação levará para ser executada após o acontecimento de um evento. Assim sendo, cabe à visão temporal verificar se o conector (relação) utilizado no elo (relacionamento), que precisa ser modificado, dá suporte à configuração de um *delay* (atraso) na execução da ação. Em caso positivo, o conector é mantido, e o novo valor do atraso é definido através de um parâmetro de *bind* (*bindParam*) ou de um parâmetro de elo (*linkParam*). Se o conector não der suporte, existem duas alternativas. A primeira seria modificar o conector, e depois salvá-lo na base de conectores. Essa alternativa não é viável porque outros elos podem estar referenciando o conector em questão, tanto dentro do mesmo documento NCL, quanto em outros documentos. A segunda alternativa, que é a adotada pela visão temporal, consiste em criar uma cópia do conector, inserindo as especificações para que seja possível a configuração do atraso na ação. Nesse caso, o novo conector é inserido na base de conectores do próprio documento que está sendo editado. Uma vez criado o conector, o elo passa a referenciá-lo, e o valor do atraso pode agora ser configurado.

Quando o autor remove um elo ou âncora na visão temporal, o elemento correspondente também é removido do documento. Já na remoção de um objeto de mídia, o autor tem a opção de removê-lo do documento ou simplesmente da visão temporal. O primeiro caso funciona do mesmo jeito que para elos e âncoras. Já no segundo, o objeto de mídia não é removido do documento, mas sim os relacionamentos que iniciam esse objeto.

Por fim, cabe lembrar que o objetivo da manipulação de objetos na visão temporal, e a conseqüente alteração dos relacionamentos temporais, é abstrair do autor a necessidade de codificação em NCL, fazendo com que todo código seja

gerado automaticamente. É importante destacar também que toda a especificação e a edição de relacionamentos na visão temporal são devidamente refletidas nas demais visões do *Composer*. Detalhes de como isso é feito serão apresentados mais adiante.

4.3.4.2. Simulação de eventos interativos

A simulação de eventos interativos permite ao autor visualizar como se dará a apresentação em tempo de criação, sem a necessidade de recorrer ao formatador para apresentar o documento. A simulação faz com que pontos de indeterminação do modelo assumam valores determinísticos mediante interação do autor.

Interagindo com o mouse sobre um objeto de mídia que é gatilho de um relacionamento interativo, o autor pode simular a interação do telespectador. Definido com o mouse o ponto (tempo) da interação, a visão temporal atualiza o modelo, e é novamente desenhada. Em outras palavras, mediante interação do autor, cadeias parciais (trechos de duração previsível cuja ocorrência não pode ser exatamente prevista) deixam de existir, sendo integradas à cadeia principal. A Figura 29 apresenta um exemplo da simulação de evento interativo.

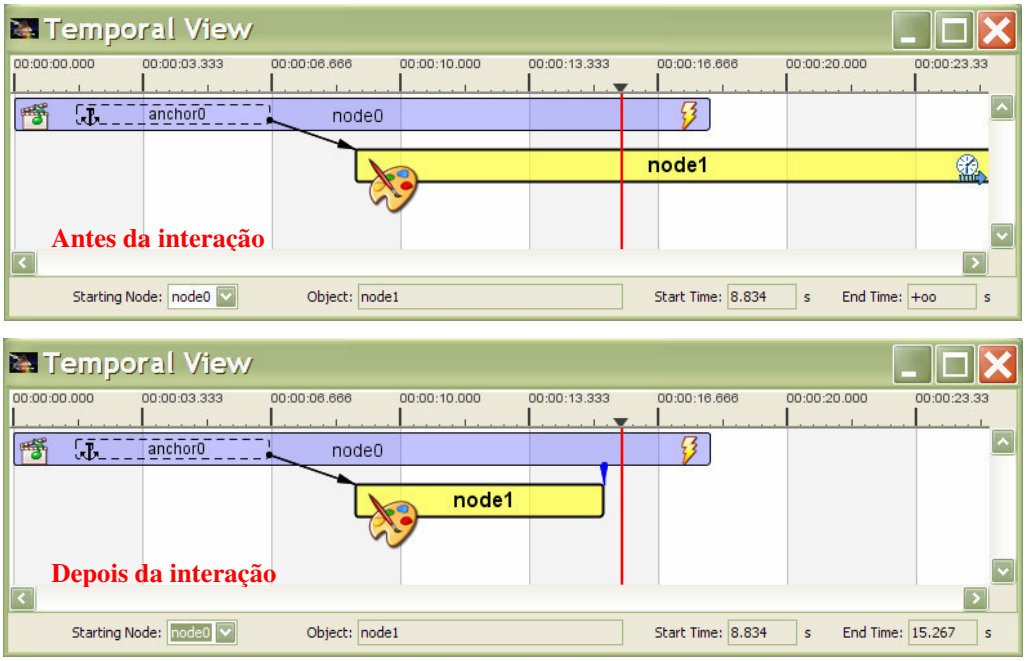

Figura 29 - Simulação de eventos interativos

Nesse exemplo, ao pressionar o botão azul do controle remoto durante a apresentação do objeto de mídia *node0*, é disparada a ação de finalizar a exibição

do objeto de mídia *node1*. Nesse caso, o elo que especifica o relacionamento interativo é representado, coincidentemente, em azul, para diferenciá-lo dos demais relacionamentos. Interagindo com o mouse nos objetos definidos como gatilhos, o autor pode voltar ao estado inicial quando desejar.

À medida que cadeias parciais são adicionadas à cadeia principal, outros eventos interativos, especificados nas cadeias parciais, podem ser simulados sucessivamente. Cabe ressaltar ainda que, se no instante escolhido para acontecimento da interatividade, outros objetos de mídia também forem gatilhos para o mesmo botão do controle remoto, seus relacionamentos interativos têm suas ações disparadas. Esse comportamento dá ao autor uma visão bem próxima da que o telespectador vai ter durante a apresentação do documento.

A implementação da visão temporal dá suporte à exibição de objetos de mídia, âncoras e elos. A representação de outras entidades do NCL, como por exemplo, nós de *switch*, fica como sugestão de trabalho futuro.

4.3.5. Sincronização entre visões

Um problema existente na implementação do editor HyperProp diz respeito à sincronização das visões gráficas com a visão textual, conforme relatado em (Coelho & Soares, 2004), onde um evento de sincronização disparado por uma das visões gráficas faz com que a organização do documento NCL seja perdida. Para resolver esse problema no *Composer*, a biblioteca que implementa a árvore DOM foi estendida, de modo a manter a ordem especificada pelo autor.

Na implementação da arquitetura apresentada na Figura 13, os compiladores utilizados foram os mesmos que já haviam sido desenvolvidos no editor HyperProp. A exceção fica por conta da visão temporal, na qual foi necessária a implementação do compilador responsável por atualizar o modelo de cadeias temporais hipermídia, e conseqüentemente, as entidades no modelo Java NCM.

A Figura 30 apresenta o diagrama de classes da arquitetura de integração do *Composer*, a qual é baseada no padrão de projeto *Mediator* (Gamma et al., 1997). Esse padrão define um mediador (classe *ComposerEditor*) que encapsula como as visões (classe *TextualView*, *StructuralView*, *LayoutView* e *TemporalView*) interagem. Quando uma visão é modificada, ela notifica ao mediador do ambiente

de autoria, e fica a cargo desse mediador acionar os compiladores das demais visões para que as modificações sejam refletidas. O mediador evita que as visões referenciem umas as outras explicitamente, e isso permite que outras visões possam ser acopladas sem necessidade de modificação nas visões já existentes.

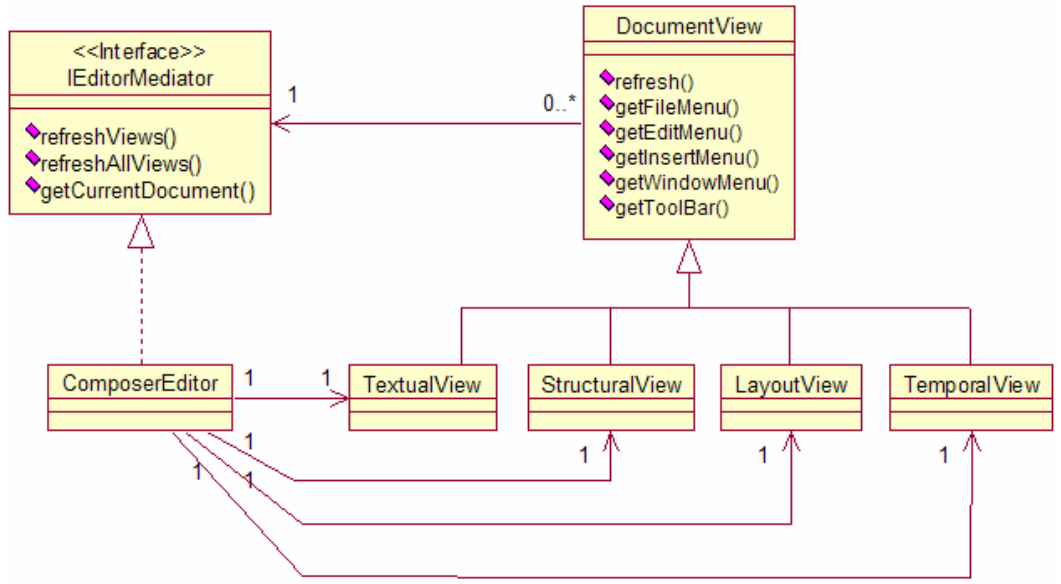

Figura 30 - Diagrama de classes do mediador no *Composer*

A arquitetura acima visa permitir a especificação de uma API para que outros desenvolvedores possam criar novas visões para o *Composer*. Da mesma forma que as visões apresentadas, as novas visões utilizariam como modelo de integração o Java NCM ou a árvore DOM correspondente.

4.4. Módulo de Edição ao Vivo

Para dar início ao processo de edição ao vivo, o autor precisa informar ao *Composer*, e conseqüentemente ao módulo de edição ao vivo, que as modificações pertinentes devem ser monitoradas. Isso pode ser feito através de uma opção na barra de menus.

Uma vez ativado o modo de edição ao vivo, a visão textual é desabilitada, não sendo possível a sua utilização durante esse período. Essa medida foi tomada devido à complexidade de se mapear as modificações feitas nessa visão em comandos de edição. A análise de alternativas para se contornar esse problema fica como trabalho futuro.

A implementação do *Composer* gera comandos de edição na criação e remoção de regiões, descritores, objetos de mídia, composições, conectores e elos.

É possível ainda chamar os comandos de preparar e iniciar apresentação de um documento NCL, no ambiente de apresentação do telespectador. À medida que são gerados, os comandos de edição podem ser visualizados através de uma aba na área multifuncional (Figura 19).

Para manter equivalência na consistência do documento NCL, tanto no ambiente de autoria, quanto no ambiente de apresentação, os comandos de edição não são guardados seguindo somente a ordem de geração. Se, por exemplo, um objeto de mídia é criado, e só depois é criado o seu descritor, o documento estaria consistente no ambiente de autoria, mas não no ambiente de apresentação. O módulo de edição ao vivo organiza os comandos de edição seguindo, primeiramente, a prioridade da entidade que o comando referencia, e posteriormente, a ordem na qual o comando foi gerado. A prioridade das entidades é dada na seguinte ordem: região, descritor, nó de mídia, nó de contexto, conector e elo. Comandos complementares sobre uma mesma entidade, como por exemplo, um comando de adicionar, seguido posteriormente por outro de remover, se anulam e, uma vez identificados, são removidos da lista de comandos de edição.

Concluído o processo de edição, o próximo passo é enviar as alterações para que elas possam ser refletidas no ambiente de apresentação dos telespectadores. Através de uma opção na barra de menus, o autor pode exportar os comandos de edição para um arquivo. Esse arquivo recebe o nome do documento NCL editado, acrescido da extensão *.seq*. Para o documento *documento1.ncl*, por exemplo, o arquivo de edição se chamaria *documento1.seq*. Se for o caso, quando o arquivo de comandos é exportado, também são criados, no mesmo diretório, os arquivos que contêm trechos NCL relativos às entidades que estão sendo adicionadas no documento.

A Figura 31 descreve, através de expressões regulares, os comandos de edição, que podem estar contidos no arquivo exportado. Nesse arquivo, cada linha corresponde a um comando. Os comandos de edição e seus parâmetros foram definidos baseados na API especificada em (Costa et al., 2006).

```
PREPARE_NCL
START_NCL
ADD_REGION [parent region perspective]? [NCL file of the new region]
REMOVE_REGION [region perspective]
ADD_DESCRIPTOR [NCL file of the new descriptor]
REMOVE_DESCRIPTOR [descriptor id]
ADD_NODE [parent context perspective] [NCL file of the new node]
REMOVE_NODE [node perspective]
ADD_CONNECTOR [NCL file of the new connector]
REMOVE_CONNECTOR [connector id]
ADD_LINK [parent context perspective] [NCL file of the new link]
REMOVE_LINK [link perspective]
```

Figura 31 - Comandos de edição ao vivo no *Composer*

Na Figura 31, os comandos de *PREPARE_NCL* e *START_NCL* informam ao ambiente de apresentação que a exibição de um determinado documento NCL deve ser preparada e iniciada, respectivamente. Já o comando de criação de região (*ADD_REGION*) tem como parâmetros a perspectiva da região pai, se houver, e o nome do arquivo que contém a especificação NCL da região a ser criada. Um exemplo de comando seria *ADD_REGION newRegion.ncl*, para inserção da região definida em *newRegion.ncl* diretamente na base de regiões, ou *ADD_REGION r0/r1 newRegion.ncl*, para inserir a nova região dentro da região *r1*, cujo pai é a região *r0*. Para remover uma região, o único parâmetro necessário é a perspectiva da região a ser removida. O comando *REMOVE_REGION r3/r4*, por exemplo, define que a região *r4*, cujo pai é *r3*, deve ser removida.

Os comandos de inserção e remoção de descritor e conector são parecidos. No caso da criação, basta passar o nome do arquivo que contém a especificação NCL do descritor ou conector a ser criado. Já no comando de remoção, basta informar o *id* (identificador) do elemento.

Os comandos de edição para nós e elos possuem a mesma regra de formação. O comando *ADD_NODE doc1/context1 node2.ncl*, por exemplo, define que o nó contido no arquivo *node2.ncl* deve ser inserido no contexto *context1* do documento *doc1*. Já o comando *REMOVE_NODE doc1/node3* especifica que o nó *node3* deve ser excluído do documento *doc1*. A mesma analogia serve para a criação e remoção de elos.

Atualmente, existe um multiplexador, responsável por enviar os comandos de edição, juntamente com o áudio e vídeo principal, implementado em C++. Para fazer a integração com o ambiente de autoria, implementado em Java, foi criado um programa em C++, que fica monitorando um determinado diretório, passado como parâmetro, em busca de arquivos que contenham comandos de edição. Cabe ressaltar que para que os arquivos sejam processados, e, conseqüentemente, os comandos enviados, o multiplexador precisa estar sendo executado.

Como trabalho futuro sugere-se a implementação do multiplexador em Java ou a utilização de JNI (Sun, 1997) para que o *Composer* possa se comunicar diretamente com o multiplexador implementado em C++. De forma geral, precisa ser trabalhada a integração do *Composer* com ambientes de transmissão. Além disso, também pode ser interessante que o instante de envio de cada comando seja especificado e armazenado, permitindo assim, que o arquivo de comandos de edição seja processado em lote (*batch*).