PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Rogério Miguel Coelho

Integração de Ferramentas Gráficas e Declarativas na Autoria de Arquiteturas Modeladas através de Grafos Compostos

DISSERTAÇÃO DE MESTRADO

DEPARTAMENTO DE INFORMÁTICA

Programa de Pós-Graduação em Informática

Rio de Janeiro

Agosto de 2004

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Rogério Miguel Coelho

Integração de Ferramentas Gráficas e Declarativas na Autoria de Arquiteturas Modeladas através de Grafos Compostos

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do título de Mestre pelo Programa de Pós-Graduação em Informática da PUC-Rio.

Orientador: Luiz Fernando Gomes Soares
Co-orientador: Rogério Ferreira Rodrigues

Rio de Janeiro, agosto de 2004

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Rogério Miguel Coelho

Integração de Ferramentas Gráficas e Declarativas na Autoria de Arquiteturas Modeladas através de Grafos Compostos

Dissertação apresentada como requisito parcial para obtenção do título de Mestre pelo Programa de Pós-Graduação em Informática da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Luiz Fernando Gomes Soares
Orientador
Departamento de Informática - PUC-Rio

Rogério Ferreira Rodrigues
Co-orientador
Departamento de Informática - PUC-Rio

Simone Diniz Junqueira Barbosa
Departamento de Informática - PUC-Rio

Sergio Colcher
Departamento de Informática - PUC-Rio

José Eungênio Leal
Coordenador(a) Setorial do Centro Técnico Científico - PUC-Rio

Rio de Janeiro, 20 de agosto de 2004

Rogério Miguel Coelho

Graduado em Engenharia de Computação pela Universidade Federal do Espírito Santo (UFES) em 2001. Atualmente, integra o grupo de pesquisadores do Laboratório TeleMídia da PUC-Rio, desenvolvendo pesquisa na área de Sistemas Hipermídia.

Ficha Catalográfica

Coelho, Rogério Miguel

Descrição da Ficha Catalográfica

Informação Técnica da Ficha Catalográfica

Natureza da Ficha Catalográfica

Incluí referências bibliográficas.

Autoria de Arquiteturas; Grafos Compostos; Olho-de-Peixe

Este trabalho é dedicado:
A meus pais Louise, Ary, Manuel e a minha irmã Lorena.

A toda amada família Miguel (Queridos avós, tios e primos).

A Deus pela dádiva por poder contar com tantas pessoas maravilhosas em minha vida.

Agradecimentos

Ao meu orientador, Prof. Luiz Fernando, por acreditar em meu potencial e por todo seu apoio nos momentos alegres e difíceis: “Capoeira que é bom não cai, mas seu um dia ele cai, cai bem”.

Ao meu co-orientador, Prof. Rogério Rodrigues, por sua atenção, paciência e boa vontade em me ensinar. Tendo sido um verdadeiro guia para o desenvolvimento deste trabalho, sua ajuda foi fundamental.

Aos meus colegas do TeleMídia, pelo companheirismo e ajuda prestados. Em especial a Heron Vilela (Filhão) pela amizade e ajuda no desenvolvimento do “V0” e, a Sergio Cavendish pela revisão deste trabalho.

Agradecimentos especiais à minha família: Louise (minha mãe), Ary (meu pai, in memorian), Manuel e Lô pelo amor, carinho e incentivo. Ao meu irmão mais velho, Flavio Varejão, pelos conselhos e ombro amigo.

À Reivani, pelo amor, companheirismo e por todos os momentos que passamos juntos aqui no Rio de Janeiro.

Aos Professores da Universidade Federal do Espírito Santo por minha formação na Graduação, sem a qual não teria chegado até aqui.

À CAPES e ao FUNTTEL pelo apoio financeiro.

E a todos aqueles que direta ou indiretamente contribuíram para realização deste trabalho.

Resumo

Coelho, Rogério Miguel. **Integração de Ferramentas Gráficas e Declarativas na Autoria de Arquiteturas Modeladas através de Grafos Compostos.** Rio de Janeiro, 2004. 108p. Dissertação de Mestrado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Este trabalho descreve um conjunto de ferramentas para auxiliar na construção de aplicativos para autoria de arquiteturas de sistemas baseadas em grafos compostos. As ferramentas são divididas em quatro visões de grafos compostos: uma visão gráfica estrutural, uma visão gráfica temporal, uma visão gráfica espacial e uma visão textual. As quatro visões funcionam de maneira sincronizada, a fim de oferecer um ambiente integrado de autoria. As visões são providas de mecanismos de filtragem para auxiliar na especificação de arquiteturas mais complexas. As ferramentas desenvolvidas foram diretamente aplicadas ao domínio de autoria de documentos hipermídia, mas podem ser utilizadas em outros domínios, como na definição de arquiteturas de sistemas de software, ferramentas de especificação formal, projeto de workflows, entre outros.

Palavras-chave

Autoria de Arquiteturas; Grafos Compostos; Olho-de-Peixe; Sistemas Hipermídia

Sumário

Lista de figuras

1
Introdução

Em julho de 1945, Vannevar Bush publicou um artigo com o título *"As We May Think"* (Bush, 1945) no qual questionava os métodos de organização da informação utilizados na comunidade científica, baseados em uma ordem puramente hierárquica. Segundo o autor, deveria ser desenvolvido um método inspirado na maneira como a mente humana funciona, ou seja, através de associações, interligando uma informação a outra por meio de referências.

Ao longo das décadas, o projeto de Bush influenciaria muitos pesquisadores como Ted Nelson, que nos anos 60 difundiu o termo hipertexto (Nelson, 1965), e Doug Engelbart, criador do *mouse* e do sistema NLS - *On Line System* (Engelbart, 1968). Contudo, foi Tim Berners-Lee, no final dos anos 80 nos laboratórios do CERN (Conselho Europeu para Pesquisa Nuclear), que iniciou o desenvolvimento da *Word Wide Web* (Berners-Lee et al., 1994), tornando-se hoje o maior e mais bem difundido sistema hipermídia do mundo, motivando inúmeros trabalhos e pesquisas.

Este capítulo descreve as principais partes que compõem um sistema hipermídia, dando ênfase ao subsistema de autoria, uma vez que é esse o contexto deste trabalho. O capítulo também detalha os objetivos e apresenta a estrutura desta dissertação.

1.1.
Motivação

Um sistema hipermídia normalmente é composto por três ambientes: o ambiente de autoria, no qual os autores criam os seus hiperdocumentos; o ambiente de armazenamento, onde os hiperdocumentos criados na fase de autoria são armazenados e mantido em servidores; e o ambiente de execução, onde os documentos recém-editados (ou já armazenados) são apresentados ao usuário. A Figura 1-1 ilustra os ambientes que modelam as três fases principais de um

sistema hipermídia, estando, conforme mencionado anteriormente, este trabalho centrado nos aspectos de autoria.

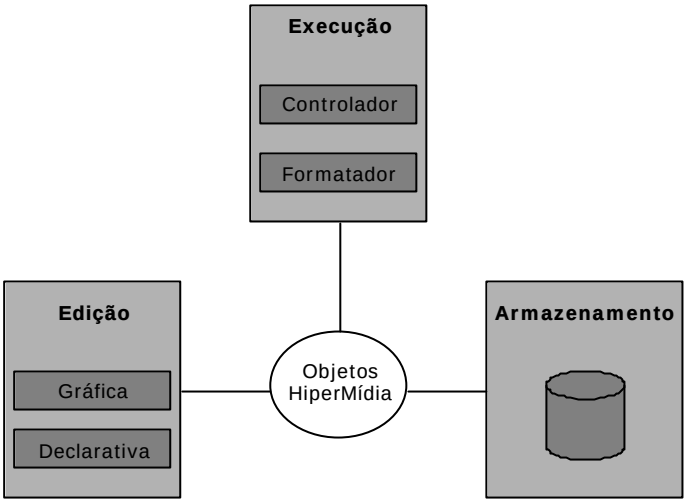


Figura 1-1 Subsistemas de um sistema hipermídia

A autoria em sistemas hipermídia costuma ser feita através do uso de interfaces gráficas, com editores especializados que oferecem facilidades para criação e edição de hiperdocumentos, ou então pela forma textual, na qual o autor utiliza um editor de texto e uma linguagem de descrição para especificar seus documentos.

Segundo Shneiderman (Shneiderman, 1998), alguns pontos positivos podem ser apontados no uso de editores gráficos para autoria de documentos hipermídia:

- visão global e prévia do documento: o autor pode ter um retorno visual mais imediato se o documento que está sendo elaborado satisfaz suas expectativas (WYSIWYG – *What You See Is What You Get*); e
- uso de ícones como rótulos para ações: o autor precisa somente estabelecer a associação entre as imagens de componentes da interface gráfica com elementos de edição de documentos hipermídia.

Além dessas vantagens, é possível destacar também:

- facilidade na criação de documentos: o autor usa os recursos gráficos disponíveis, sem se preocupar com aspectos de programação (Muchaluat-Saade, 2003), como, por exemplo, a sintaxe de uma linguagem textual de autoria. Isso favorece a utilização por parte de autores não-programadores; e

- facilidade no reuso: partes dos documentos ou até mesmo objetos de mídia são facilmente identificados e reaproveitados na elaboração de um novo hiperdocumento (Pinto, 2000).

No entanto, algumas desvantagens também podem ser destacadas quando se faz uso de editores gráficos, tais como:

- necessidade de maior poder de processamento (memória, CPU etc.) e de recursos de visualização (tela maior, por exemplo) para oferecer suporte às aplicações gráficas;
- uso de editores mais elaborados e de implementação mais complexa (Muchaluat-Saade, 2003), ou seja, dificuldade na criação do editor gráfico; e
- necessidade do software de edição específico, que muitas vezes, além de não ser gratuito, grava os documentos em formatos não abertos.

Uma outra alternativa para especificação de documentos hipermídia que vem sendo muito explorada é o uso de linguagens declarativas, através das quais o autor descreve seu documento utilizando apenas editores de texto, normalmente fazendo uso de linguagens de marcação (W3C, 2001) (HTML, 1999).

Alguns pontos positivos podem ser salientados na criação de documentos hipermídia utilizando a forma declarativa:

- necessidade de menos recursos no processamento de edição de documentos hipermídia, como, por exemplo, o uso de editores de texto simples para autoria sobre plataformas com pouco poder de processamento (Schwabe & Medeiros, 2001);
- baixo custo, pois existe a possibilidade de se utilizar um editor de texto básico e gratuito (Bulterman & Rutledge, 2004);
- o processo de desenvolvimento da aplicação tende a se tornar mais seguro, pois a especificação criada pode ser validada por um conjunto de regras pré-definidas e mapeada automaticamente para um ambiente de implementação (Schwabe & Medeiros, 2001);
- a especificação textual pode ser validada e depurada pelo próprio ser humano, facilitando, por exemplo, a recuperação de documentos que tenham sido corrompidos;

- qualquer editor, por mais simples que seja, pode oferecer todas as facilidades da linguagem hipermídia para construção do documento (Bulterman & Rutledge, 2004).

Novamente, como acontece com a autoria gráfica, existem desvantagens na edição declarativa:

- a autoria declarativa exige um domínio da sintaxe e semântica da linguagem que normalmente envolvem um tempo maior de aprendizado; e
- dependendo da forma como os documentos são especificados (quantidade de informação a ser descrita), a tarefa de autoria pode se tornar mais demorada e trabalhosa.

Para que um ambiente de autoria de documentos hipermídia agregue as vantagens oferecidas nas formas de edição gráfica e declarativa, buscando evitar, ou diminuir, as desvantagens apontadas em cada um dos paradigmas, é interessante que o ambiente ofereça suporte às duas abordagens e, com isso, possibilite que o autor mude, de maneira ágil, a forma de autoria de seus documentos conforme o seu interesse em um dado momento.

Voltando à autoria gráfica, alguns sistemas possuem ferramentas para a criação dos documentos que permitem ao autor explorar mais de uma visão para um dado documento. No caso do sistema HyperProp (Soares et al., 2000), um sistema para autoria e formatação de documentos hipermídia baseados no modelo conceitual NCM – *Nested Context Model* - (Soares et al., 2003), o subsistema de autoria fornece três visões diferentes, conforme ilustrado esquematicamente na Figura 1-2.

Uma das visões (estrutural) apóia-se na estrutura lógica do documento, fornecendo recursos para editar componentes, seus relacionamentos e definir o agrupamento desses componentes em estruturas hierárquicas (composições) (Pinto, 2000). Uma segunda visão (temporal) é responsável pela especificação dos relacionamentos de sincronização entre os componentes do documento ao longo do tempo (estrutura temporal), definindo suas posições relativas em um eixo temporal (Costa, 1996). Uma terceira (espacial) e última visão permite a definição de relacionamentos espaciais entre componentes de um documento, estabelecendo suas características de apresentação em um determinado dispositivo (Moura, 2001). Múltiplas visões gráficas são também exploradas em outros sistemas, como

por exemplo o GRiNS (Bulterman et al., 1998) e Kaomi (Jourdan et al., 1999), que serão discutidos mais detalhadamente no Capítulo 5 (Trabalhos Relacionados).

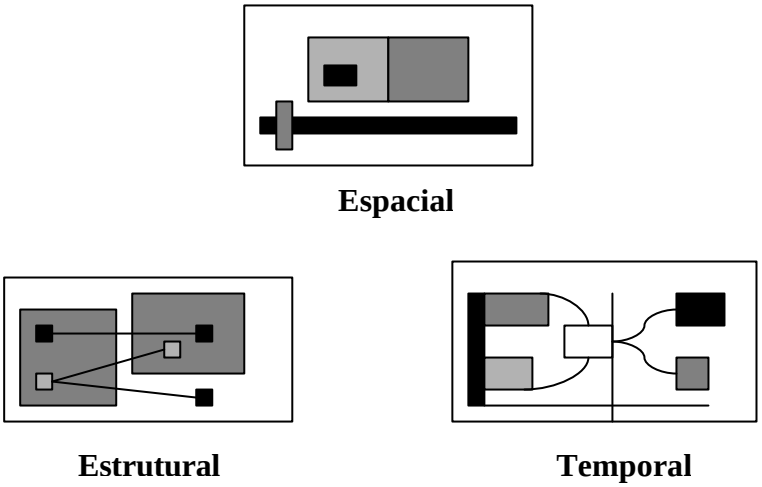


Figura 1-2 Visões de documentos hipermídia

Uma característica a ser destacada é que editores gráficos que oferecem múltiplas visões devem ter a capacidade de chavear entre uma visão e outra, de tal modo que um componente (objeto) em foco em uma determinada visão (temporal, por exemplo) seja também mostrado como foco, e com todas as suas características de apresentação, em uma outra visão (por exemplo, a visão espacial) (Muchaluat et al., 1997).

Nesse cenário, é possível entender a autoria declarativa como uma quarta opção de visão para criação dos documentos, devendo também estar sincronizada com as demais visões gráficas oferecidas pela ferramenta de autoria.

Não importa a visão utilizada, um problema conhecido enfrentado pelos usuários que trabalham com estruturas de dados grandes na fase de autoria é a desorientação na busca por determinada informação. Em documentos hipermídia, essa desorientação é causada principalmente pelo número elevado de nós e elos (relacionamentos) (Pinto, 2000).

Dentre as técnicas mais comuns para a apresentação de estruturas grandes, podem ser citadas o uso de recursos de *zoom* e *scroll* em partes da estrutura. Entretanto, a técnica que se mostra muito eficaz, mas também mais complexa, é a filtragem de partes da estrutura não relevantes para o usuário (Pinto, 2000). A dificuldade é identificar quais partes da estrutura realmente interessam ao usuário.

A visão olho-de-peixe proposta por Furnas (Furnas, 1986) atua como uma lente, preservando os detalhes próximos a um ponto escolhido e, à medida que o

usuário se afasta desse ponto, menos informação é exibida, mostrando apenas as referências mais importantes.

Nos mapas do espaço hipermídia (Muchaluat et al., 1998), a filtragem baseada na estratégia olho-de-peixe fornece, em uma única visão, os detalhes locais referentes à posição do usuário durante o processo de navegação (nó em foco), sem perder a noção da estrutura global dos documentos. Esta dissertação explora essa técnica, anteriormente aplicada à visão estrutural no sistema HyperProp, estendendo seu uso para as visões espacial e declarativa, de forma integrada à visão estrutural.

A criação de um editor e a implementação de suas funcionalidades (como técnicas de filtragens, sincronização entre as visões etc.) estão diretamente relacionadas com o modelo conceitual que rege a especificação dos documentos. Modelos com maior poder de expressão provêem mais recursos para autoria, isto é, permitem a criação de documentos mais elaborados. Todavia, editores gráficos com mais recursos devem ser implementados para dar suporte a todas as funcionalidades do modelo.

Muitos são os modelos hipermídia propostos na literatura. Basicamente, para cada modelo existe um ambiente de autoria a ele associado, tais como: NCM – *Nested Context Model* (Soares et al., 2003) para o ambiente de autoria do sistema HyperProp, AHM (*The Amsterdam Hypermedia Model*) (Hardman et al., 1993) – que influenciou o modelo usado na linguagem SMIL (SMIL, 2001) do ambiente GRiNS (Bulterman et al., 1998), o modelo Labyrinth (Jühne et al, 1998) da ferramenta Ariadne, o modelo CMIF (Hardman et al., 1993) do editor CMIFed e o modelo Madeus (Villard, 2000) utilizado pelo ambiente Kaomi (Jourdan et al., 1999). Vale destacar que o Kaomi tem como propósito ser um ambiente de autoria hipermídia flexível, ou seja, que pode ser utilizado para autoria de documentos pertencentes a diferentes modelos conceituais.

Conforme pode ser observado na Figura 1-2, as descrições visuais de um documento hipermídia podem ser entendidas como grafos, onde os nós dos documentos são vértices do grafo e os relacionamentos entre os nós são representados como arestas.

A maioria dos modelos hipermídia citados anteriormente, como o NCM, Madeus e SMIL, apresentam o conceito de composição na sua representação. Nesses modelos, um nó pode, na realidade, definir uma estrutura de composição

que contém outros nós (simples ou compostos), recursivamente. Dessa forma, mais do que grafos, as ferramentas de autoria para modelos com essas características devem lidar com grafos compostos (Noik, 1993).

Quando uma semântica é aplicada ao grafo composto, como a semântica dos modelos de documentos hipermídia, tanto as arestas quanto o aninhamento de composições de vértices podem representar diversos tipos de relações. Uma relação temporal, por exemplo, entre dois vértices diz respeito às suas ordenações no tempo. De forma análoga, uma relação espacial entre dois vértices diz respeito aos seus posicionamentos no espaço de recursos do sistema.

Assim como diferentes modelos hipermídia, muitas arquiteturas de sistemas também podem ter os seus componentes representados através de grafos compostos. Exemplos de tais arquiteturas de sistemas podem ser encontrados nas configurações de software modeladas por ADLs (*Architecture Description Languages*) (Allen, 1997) (Clements, 1996), em ferramentas de descrição formal de arquiteturas (Felix, 2004) e na construção de sistemas de *Workflows* (Hollingsworth, 2004). Assim, ao tratar a autoria de sistemas hipermídia como autoria de grafos compostos, pode-se estender as ferramentas desenvolvidas para que operem também na autoria das arquiteturas desses sistemas. Esse ponto será explorado posteriormente.

1.2. Objetivos

Esta dissertação tem como principal objetivo o desenvolvimento de uma ferramenta para especificação de arquiteturas de sistemas passíveis de serem modeladas por grafos compostos, em especial grafos com composicionalidade, conforme será melhor discutido no Capítulo 2. O ambiente de autoria baseia-se na integração de visões textual, estrutural, temporal e espacial das arquiteturas.

Como parte dos objetivos, trabalhos anteriormente realizados dentro do âmbito de ferramentas gráficas para especificação de documentos hipermídia no sistema Hyperprop (Soares et al., 2000) serão estendidos. As extensões envolvem tanto a incorporação da visão textual às antigas ferramentas, quanto à remodelagem das várias visões para aplicação em outros domínios de arquitetura que será detalhado no Capítulo 3.

A visualização (desenho) de grafos não é uma tarefa simples. Existem diversos algoritmos propostos na literatura para desenhar diferentes tipos de grafos (Carpano, 1980) (Kamada & Kawai, 1989), que serão abordados no Capítulo 5 (Trabalhos Relacionados), porém, a legibilidade de um grafo não é apenas obtida com um bom desenho do mesmo, mas também com técnicas de filtragem que melhoram o entendimento da estrutura. A fim de facilitar o processo de edição, a técnica de filtragem olho-de-peixe será aplicada nas visões gráficas (espacial e estrutural) e declarativa de forma sincronizada, para reduzir a quantidade de informações exibidas, tornando o documento mais legível para o autor. A aplicação desta técnica na visão temporal não se encontra implementada e faz parte dos trabalhos futuros.

1.3. Estrutura da Tese

Esta dissertação encontra-se organizada como a seguir. O Capítulo 2 discute os conceitos básicos de grafos compostos e grafos compostos com composicionalidade. O capítulo então comenta algumas arquiteturas de sistemas que podem fazer uso desses conceitos em suas concepções. Por fim, o capítulo apresenta os conceitos básicos do modelo NCM e da sua representação declarativa (a linguagem NCL), que serão usados no restante do trabalho, uma vez que o foco desta dissertação está na especialização dos editores de grafos para autoria de documentos hipermídia.

O Capítulo 3 apresenta o ambiente de edição desenvolvido, descrevendo as suas quatro visões (espacial, estrutural, temporal e declarativa). Nesse capítulo, o mecanismo de sincronização para a integração das várias visões é também apresentado.

O Capítulo 4 destaca as técnicas de filtragem implementadas para o ambiente de autoria desenvolvido.

O Capítulo 5 compara o editor elaborado nesta dissertação com outras ferramentas para autoria de documentos hipermídia e grafos, destacando os pontos positivos e negativos de cada ferramenta. Por fim, o Capítulo 6 tece as conclusões e descreve trabalhos futuros.

2
Conceitos Preliminares

O uso de grafos para modelagem de problemas encontra-se na literatura clássica desde a época de Leonhard Euler (1707-1783), quando o mesmo solucionou o conhecido problema das "Pontes de Königsberg" (Cormen et al., 2001).

A modelagem de problemas em grafos pode ser bastante útil, pois através delas é possível formalizar diferentes topologias em uma linguagem de representação padrão, para a qual já existem diversos teoremas que podem auxiliar na solução dos problemas. Além disso, sua representação gráfica pode ajudar na análise do problema. Porém, nem sempre essa visualização é plausível de ser utilizada, devido à sua dimensão e à sua complexidade. Conforme comentado no Capítulo 1, nesses casos, técnicas de filtragem podem ser utilizadas para reduzir tal complexidade.

Grafos podem ser usados em diversas áreas do conhecimento. Certos sistemas, como redes de computadores, sistemas hipermídia, mapas geográficos, arquiteturas de software, entre outros, apresentam características de estruturação que podem ser representadas através de grafos compostos.

Este capítulo está organizado da seguinte forma. A Seção 2.1 apresenta a definição de grafos compostos. A Seção 2.2 descreve algumas arquiteturas de sistemas modeladas através de grafos compostos. Por fim, na Seção 2.3, o conceito de grafos com composicionalidade é analisado em modelos hipermídia, quando então são resumidamente apresentados o modelo conceitual NCM – *Nested Context Model* (Soares et al., 2003) e a linguagem NCL – *Nested Context Language* (Muchaluat-Saade et al., 2003).

2.1.
Definição de Grafos Compostos

Um grafo composto G é definido por um conjunto de vértices V e um conjunto de arestas A, onde cada elemento de A define uma relação entre dois

elementos de *V* (Noik, 1993). O conjunto de vértices *V*, por sua vez, pode ser particionado em dois subconjuntos: *CV* e *AV*. Os vértices pertencentes a *CV* são chamados de vértices compostos e identificam, cada um, um subconjunto de vértices do grafo que são por ele contidos, formando um subgrafo de *G*. Os vértices pertencentes a *AV* são vértices que não podem conter outros vértices, sendo, por isso, chamados de vértices atômicos.

Mais precisamente, um grafo composto é definido por uma tupla G = (V, A, I), onde:

V = CV ? AV
A ? (V x V)
I ? CV x V é um conjunto de relações de inclusão.

Duas restrições ainda se fazem necessárias: um vértice composto *cv* pertencente a *CV* não pode conter a si mesmo e nem recursivamente conter algum vértice[1] que o contenha (direta ou recursivamente); e um mesmo vértice do grafo não pode estar diretamente contido em mais de um vértice composto (Noik, 1993).

Como a propriedade de *composicionalidade* é importante no tratamento de alguns grafos compostos, pode-se restringir a definição do conjunto de arestas *A*. Dessa forma, define-se, no contexto desta dissertação, um *grafo com composicionalidade*, um grafo composto em que o conjunto de arestas *A* é formado pela união dos conjuntos *B* e *M*, tais que:

a) B ? (V x V) e, para qualquer par de vértices (*v*,*c*) pertencentes a *B*, ou ? *cv* ? *CV* tal que *v* ? *cv* e *c* ? *cv*, ou ? *cv* ? *CV*, *v* ? *cv* e *c* ? *cv*. Em resumo, os vértices de uma aresta ou estão contidos em uma mesma composição ou não estão contidos em nenhuma composição. Essa restrição faz com que uma aresta definida em *B* não possa cruzar a fronteira de um vértice composto.

b) M ? (CV x V) e, para qualquer par de vértices (*cv*,*v*) pertencentes a *M*, *v* ? *cv*. Em outras palavras, *M* constitui um conjunto de arestas de pais para filhos, sendo, portanto, um subconjunto de *I*.

[1] Diz-se que um vértice composto *c* contém um vértice *v* recursivamente, se existe ao menos um vértice composto *r* contido em *c* tal que *r* contém *v* diretamente ou contém *v* recursivamente.

A Figura 2-1 ilustra um grafo composto com composicionalidade. Nessa figura, vértices cn_i são vértices compostos, vértices an_i e c_i representam vértices atômicos, as arestas b_i pertencem ao subconjunto B de arestas, e as arestas m_i pertencem ao subconjunto M. Note que relacionamentos entre vértices que não estejam diretamente contidos em um mesmo vértice composto são construídos através de concatenações de arestas definidas em B e em M (an_1 e c_2, por exemplo).

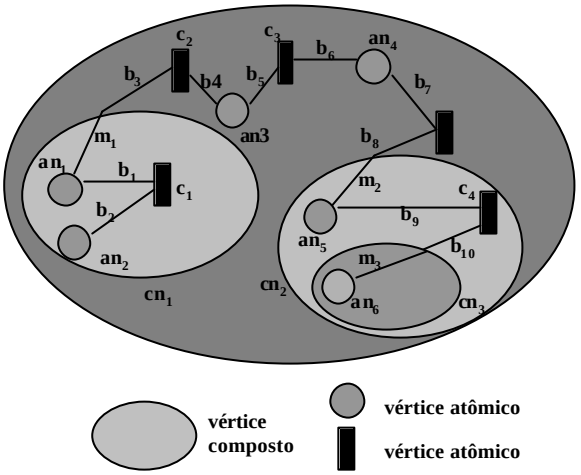


Figura 2-1 Grafo composto com composicionalidade

2.2. Sistemas Modelados através de Grafos Compostos

Conforme já comentado no Capítulo 1 e no início deste capítulo, a estrutura de dados de um grafo composto pode ser usada na modelagem de sistemas que apresentam o conceito de composição. Mais ainda, quando esses sistemas oferecem composicionalidade, grafos com composicionalidade podem ser aplicados na modelagem. Nas próximas subseções, alguns sistemas são descritos como exemplos.

2.2.1. ADLs (*Architecture Description Languages*)

Linguagens de descrição de arquitetura (ADLs) são linguagens formais que podem ser usadas para representar a arquitetura de um sistema de software, definindo seus componentes, como eles se comportam e os padrões e mecanismos para interações entre esses componentes (Clements, 1996).

Uma ADL pode também ser entendida como uma linguagem voltada para especificar a estrutura de alto nível de uma aplicação (ou seja, a arquitetura conceitual), ao invés de se preocupar com detalhes de implementação de um módulo de código específico.

Uma descrição arquitetural é composta de elementos básicos (ou blocos principais), que são os componentes, conectores e configurações arquiteturais.

- componentes são unidades de computação ou armazenamento de dados que, dependendo do nível de abstração, podem ser tão pequenos quanto um único procedimento ou tão grandes como uma aplicação inteira;
- conectores são usados para modelar interações entre componentes e definir as regras que governam essas interações;
- configurações arquiteturais são grafos conexos de componentes e conectores que descrevem a estrutura da arquitetura.

Diversas propostas de ADLs podem ser encontradas na literatura, entre elas ACME (Garlan et al., 1997), Aesop (Garlan, 1995), Armani (Monroe, 1999), CL (Paula, 1999), Darwin (Magee et al., 1996) e Wright (Allen, 1997).

Muitas ADLs admitem a existência de elementos (componentes ou conectores) compostos e garantem a composicionalidade da arquitetura. Assim, uma configuração em ADL pode ser representada por um grafo com composicionalidade, onde os vértices do grafo representam componentes e conectores. Pela definição da Seção 2.1, as arestas de B representam as ligações (*binds*) entre portas de componentes e papéis de conectores (seguindo a terminologia da ADL *Wright* (Allen, 1997)). Componentes e conectores compostos são vértices compostos que contêm componentes e conectores como elementos internos. Portas de um componente composto podem exportar interfaces (portas) de seus componentes internos, através de mapeamentos entre portas (representados por arestas definidas em M). Da mesma forma, papéis de um conector composto podem exportar papéis de seus conectores internos, através de mapeamentos entre papéis (também representados por arestas definidas em M). A Figura 2-2 apresenta uma arquitetura de ADL representada como um grafo composto com composicionalidade.

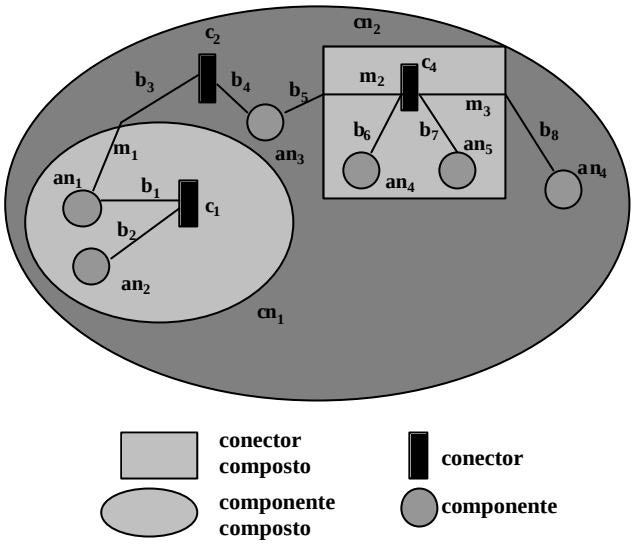


Figura 2-2 Representação de uma arquitetura de software descrita através de uma ADL utilizando grafos compostos

Uma forma de descrever ADLs textualmente é através da linguagem xADL (Dashofy et al., 2001), sendo descrita a seguir.

2.2.1.1. xADL - *XML-Based Architecture Description Language*

Em (Dashofy et al., 2001), é apresentada uma linguagem para a descrição de arquiteturas, denominada xADL - *XML-Based Architecture Description Language*. A linguagem xADL foi desenvolvida na Universidade da Califórnia e é extensível, pois todos os seus elementos e atributos são mapeados em módulos individuais, implementados através de XML *Schema* (XML, 2001), padrão W3C. A xADL possibilita, entre outras facilidades, a especificação dos relacionamentos dos elementos que compõem uma arquitetura.

A linguagem xADL adotou as entidades tradicionais, encontradas nas principais ADLs, para sua constituição, sendo formada por componentes, conectores, *links*, interfaces (pontos de entrada e saídas para conectores e componentes) e configurações (topologia da organização dos componentes e conectores).

Os componentes em xADL são identificados pelo elemento "*component*". Cada componente pode conter: *i)* interfaces, representadas pelo elemento "*interface*"; *ii)* mapeamentos, representados pelo elemento "*signatureInterfaceMapping*"; *iii)* conectores, representados pelo elemento "*conector*"; e *iv)* outros componentes.

As interfaces podem ser de três tipos: *i) "in"*, fluxo de comunicação apenas de entrada; *ii) "out"*, fluxo de comunicação apenas de saída ; e *iii) "inout"*, para representar tanto fluxo de saída como de entrada.

O elemento *"signatureInterfaceMapping"* (mapeamento) é constituído por dois outros elementos internos a ele: *"outerSignature"*, para referenciar uma das interfaces do elemento no qual o mapeamento está sendo definido (vértice composto) e *"innerInterface"*, para referenciar uma interface de um dos elementos filhos no qual o mapeamento está sendo definido.

O elemento *"link"* relaciona elementos (componentes e conectores) que tenham o mesmo pai ou dois elementos independentes (ausência de pai), respeitando assim o conceito de composicionalidade. Os *"links"* são formados por um conjunto de elementos *"point"*, que referenciam interfaces de componentes ou interfaces de conectores.

Os conectores, assim como os componentes, são constituídos por um conjunto de interfaces, mapeamentos, componentes e conectores internos a ele. É importante no entanto destacar que, em xADL, conectores só exportam interfaces de conectores filhos e componentes só exportam interfaces de componentes filhos.

A Figura 2-3 apresenta um documento HTML - *HyperText Markup Language* (HTML, 1999) com seu respectivo código mapeado em xADL na Figura 2-4.

```
<html>
 <head>
   <script language="JavaScript">
        function MsgBox (textstring) { alert (textstring) }
    </script>
 </head>
 <body>
   <form>
     <input name="text1" type= "text">
     <input name="Go!" type= "button" value="Go!" onClick="MsgBox (form.text1.value)">
   </form>
  </body> </html>
```

Figura 2-3 Documento HTML

```
<component id = "headComp" type = "Component">
        <description type = "Description">Head</description>
        <interface id = "headCompRight" type = "Interface">
```

```
            <description type = "Description">Head.Right </description>
            <direction type = "Direction"> inout </direction>
        </interface>
        <component id = "javaComp" type = "Component">
            <description type = "Description">function</description>
            <interface id = "javaCompRight" type = "Interface">
                <description type = "Description">function.Right </description>
                <direction type = "Direction" >in</direction>
            </interface>
            <interface id = "javaCompLeft" type = "Interface">
                <description type = "Description">function.Left</description>
                    <direction type = "Direction" >out</direction>
            </interface>
        </component>
        <signatureInterfaceMapping>
            <outerSignature href = "headCompRight"/>
            <innerInterface href= "javaCompLeft" />
            <outerSignature href = "headCompRight"/>
            <innerInterface href= "javaCompRight" />
         </signatureInterfaceMapping>
</component>
<component id = "bodyComp" type = "Component">
        <description type = "Description">Body</description>
        <interface id = "bodyCompLeft" type = "Interface">
            <description type = "Description"> Body.Left </description>
            <direction type = "Direction" >inout </direction>
        </interface>
</component>
<connector id = "con1" type = "Connector">
        <description type = "Description" >Connector</description>
        <interface id = "con1Left" type = "Interface">
            <description type = "Description">Con1.Left</description>
            <direction type = "Direction:>inout</direction>
        </interface>
         <interface id = "con1Right" type = "Interface">
            <description type = "Description">Con1.Right</description>
            <direction type = "Direction:>inout</direction>
        </interface>
</connector>
<link id = "link1" type = "Link">
```

```
    <description type = "Description">Head-Con1</description>
    <point type = "Point">
        <anchorOnInterface href = "con1Left" type = "XMLLink" />
    </point>
    <point type = "Point">
        <anchorOnInterface href = "headCompRight" type = "XMLLink"/>
    </point>
</link>
<link id = "link2" type = "Link">
    <description type = "Description">Body-Con1</description>
    <point type = "Point">
        <anchorOnInterface href = "con1Right" type = "XMLLink"/>
    </point>
    <point type = "Point">
        <anchorOnInterface href = "bodyCompLeft" type = "XMLLink"/>
    </point>
</link>
```

Figura 2-4 Documento xADL

Inicialmente é criado um componente *"headComp"*, para representar o elemento *"head"* da página HTML, contendo uma interface *"headCompRight"* do tipo *"inout"* (o fluxo pode ser de entrada e saída), um componente filho denominado de *"javaComp"*, e mapeamentos *"signatureInterfaceMapping"* relacionando a interface de *headComp* com as interfaces de *javaComp*.

O componente *"javaComp"* representa o elemento *"script"* da página HTML e contém duas interfaces: *"javaCompRight"* do tipo *"in"* e *"javaCompLeft"* do tipo *"out"*.

Os mapeamentos entre os componentes *"headComp"* (vértice composto) e *"javaComp"* (vértice atômico interno a *"headComp"*) são feitos através do elemento *"signatureInterfaceMapping"*, onde a interface do elemento externo é referenciada no elemento *"outerSignature"*, e a interface do elemento interno é referenciada através do elemento *"innerInterface"*.

O componente *"bodyComp"* (vértice atômico), representado pelo elemento *"body"* da página HTML, é constituído pela interface *"bodyCompLeft"* do tipo *"inout"*.

O conector *"con1"* foi criado para possibilitar a comunicação entre dois componentes. Para isso, duas interfaces do tipo *"inout"* foram criadas no conector:

"*con1Left*" e "*con1Right*". Nesse documento xADL, a finalidade do conector é permitir a comunicação entre os componentes "*bodyComp*" e "*headComp*".

Os "*links*" fazem associações entre a interface de um conector e a interface de um componente. É importante destacar que quando um "*link*" referencia a interface de um nó, e este nó apresenta mapeamentos exportando interfaces de seus elementos internos, o "*link*" juntamente com esses mapeamentos permitem uma conexão entre elementos de níveis diferentes, respeitando assim o conceito de composicionalidade.

2.2.2. Forma

Ferramentas de especificação formal de arquiteturas têm como uma de suas bases a composicionalidade. Algumas dessas ferramentas apresentam um conceito de composicionalidade mais amplo do que as ADLs (descritas na subseção anterior) e os modelos hipermídia (expostos na Seção 2.3). Forma (Felix, 2004), por exemplo, define composições como contendo conectores, componentes e outras composições. No entanto, ao contrário das ADLs, que restringem os mapeamentos apenas entre componentes ou entre conectores, interfaces de uma composição em Forma podem exportar interfaces de qualquer um de seus componentes filhos (exportações que, novamente utilizando a definição da Seção 2.1, podem ser representadas por arestas de *M*). Uma arquitetura pode ser representada por um grafo composto tendo como vértices composições, componentes e conectores. Ligações entre os vértices preservam a composicionalidade do grafo (representadas por arestas de *B*). A Figura 2-5 apresenta uma arquitetura segundo Felix (Felix, 2004).

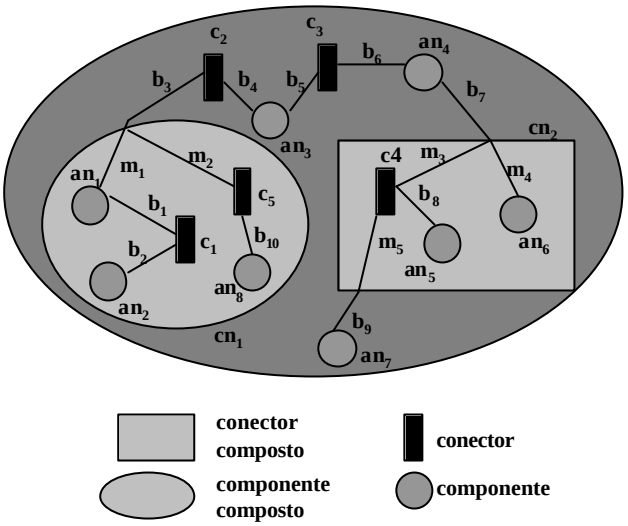


Figura 2-5 Arquitetura de forma em grafos compostos

2.2.3. Workflow

De acordo com Hollingsworth (Hollingsworth, 2004), *Workflow* é a automação de um processo de negócio, em parte ou como um todo, onde documentos, informações e tarefas são passadas de um participante ao outro, de acordo com um conjunto de regras definidas.

Em outras palavras, *Workflow* é fluxo de trabalho que descreve o que deve ser executado e como as atividades são encadeadas. As características de um *Workflow* são, basicamente, possuir um fluxo de atividades com um início e um fim definidos e, em alguns casos, envolver processos repetitivos.

O fluxo de trabalho é composto de atividades encadeadas e baseado em regras de processo de negócio. Conforme as atividades são realizadas, regras de negócio são aplicadas sobre essas atividades, direcionando-as para cada grupo de trabalho com base nas regras pré-determinadas, ou seja, na lógica de processo. As regras são atributos que definem de que forma os dados que trafegam pelo fluxo de trabalho devem ser transformados, direcionados e monitorados.

Segundo (Cruz, 1998), rotas são caminhos lógicos que, definidas sobre regras específicas, têm a função de transferir a informação dentro do processo, ligando as atividades do fluxo de trabalho. Os tipos de rotas são:

- serial - a atividade possui apenas uma atividade posterior. Cada atividade deve ser completada para ocorrer o início da seguinte;

? paralelo - grupo de atividades que ocorrem ao mesmo tempo onde as atividades são independentes entre si. As atividades de um grupo paralelo possuem a mesma atividade anterior que dispara o início das mesmas. Além disso, as atividades de um grupo paralelo possuem a mesma atividade posterior que aglutina os seus finais; e

? caminho condicional: a escolha de uma rota é determinada por uma regra.

A representação gráfica de um *Workflow* pode ser mapeada em um grafo composto, onde cada grupo de atividades pode ser representado como um vértice (composto) e os relacionamentos entre eles podem ser representados como arestas do grafo (*binds* e mapeamentos).

A Figura 2-6 ilustra a representação em grafo de um *Workflow* onde podem-se perceber seis vértices atômicos (an_x) e um vértice composto (cn_1), com lógica paralela na execução de três atividades. Nesse caso, as atividades são modeladas como vértices atômicos e os grupos de atividades como vértices compostos.

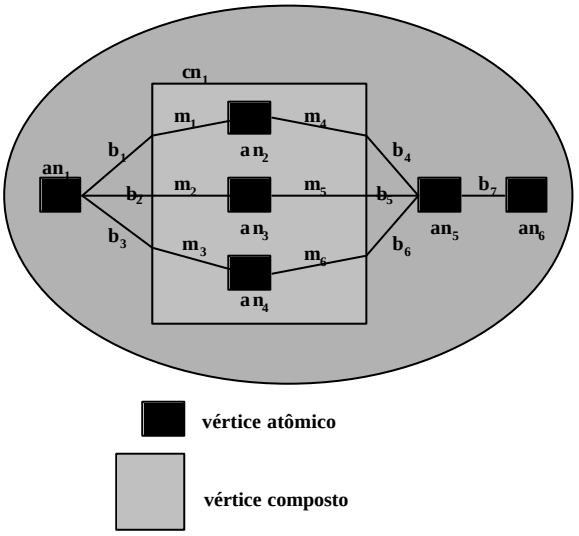


Figura 2-6 *Workflow* mapeado em grafos compostos

Existem diversas linguagens baseadas em XML para representação de sistemas de gerência de *Workflow*. Três dessas propostas vêm da indústria: XPDL, Wf-XML e WSFL. Outras são acadêmicas, tais como: XRL e WQM (Zschornack, 2003).

Na próxima subseção, a linguagem XRL é explicada como exemplo de representação declarativa de *Workflow*.

2.2.3.1. XRL - *Exchangeable Routing Language*

A linguagem XRL - *Exchangeable Routing Language* - (Aslst & Kumar, 2000), criada na Universidade do Colorado, tem seu formalismo semântico baseado nas Redes de Petri.

XRL possui elementos definidos na linguagem com semânticas específicas para a definição dos trajetos a serem percorridos para a conclusão de um determinado processo, sendo eles:

- tarefa (vértice atômico): ação a ser realizada, possuindo uma série de atributos, como nome, documentos, tempo de início e fim da tarefa etc. O elemento tarefa é representado na linguagem através da *tag "task"*;
- seqüência (vértice composto): os elementos internos ao elemento seqüência devem ser executados na ordem em que estão dispostos no documento, ou seja, os primeiros elementos definidos deverão ser os primeiros a serem processados, O elemento seqüência é representado na linguagem através da *tag "sequence"*;
- paralelismo (vértice composto): os elementos definidos dentro de um elemento paralelo **são iniciados** ao mesmo tempo. Três tipos de paralelismo são definidos na linguagem com relação **ao término** do processamento dos elementos: *i)* paralelismo com sincronismo total: todos os elementos em paralelo devem terminar suas execuções para que o fluxo continue, representado na linguagem pela *tag "Parallel_sync"*; *ii)* paralelismo com sincronismo parcial: o fluxo pode continuar quando alguns elementos terminam seus processamentos, representado na linguagem pela *tag "Parallel_part_sync"*; e *iii)* paralelismo sem sincronismo: não existe espera entre o término de processamentos dos elementos para que o fluxo continue, basta que um elemento termine a tarefa para o fluxo continuar seu processamento, representado na linguagem pela *tag "Parallel_no_sync"*;

- condicional (vértice composto): somente os elementos que satisfizerem determinada condição serão executados, representado na linguagem pelo elemento *"condition"*; e
- repetição (vértice composto): construção que permite a execução repetida de seus elementos internos. Nesse caso, uma condição para o laço deve ser testada, representada na linguagem pela *tag* "*repetition*".

A fim de explicar a funcionalidade dos principais elementos presentes na linguagem XRL, a Figura 2-7 ilustra um exemplo de uso da linguagem.

```
<route>
  <sequence>
     <task name = "an1" … >
          <parallel_sync name = "cn1" …>
        <task name = "b" name = "an2"… >
          <task name = "c" name = "an3" …>
        <task name = "d" name = "an4"… >
      </parallel_sync>
       <task name = "an5" … >
       <task name = "an6" … >
    </sequence>
</route>
```

Figura 2-7 Um exemplo de W*orkflow* em XRL

2.3. Modelos Hipermídia

De forma análoga às ADLs e às ferramentas de especificação formal, alguns modelos hipermídia preservam a noção de composicionalidade, embora quase sempre limitada a apenas uma de suas entidades (um subconjunto de seus vértices): os nós. Mesmo modelos bastante semelhantes às ADLs, como o modelo NCM (*Nested Context Model*) - e conseqüentemente a linguagem NCL (*Nested Context Language*) (Muchaluat-Saade et al., 2003) baseada nesse modelo - embora apresentem a noção de conector como entidade de primeira classe, não permitem em suas versões atuais conectores compostos.

A maioria dos modelos, entretanto, como o da linguagem SMIL (SMIL, 2001), não modela os relacionamentos entre nós como entidades de primeira classe. Desse modo, não existe a noção de conectores, mas apenas de elos interligando nós. Independente dessas diferenças, em todos os casos, um documento hipermídia pode ser representado por um grafo composto, onde seus nós (composições ou não) são representados pelos vértices do grafo.

Em modelos hipermídia com conectores, os conectores também são representados por vértices no grafo. Similar às ADLs, portas de um nó de composição podem exportar interfaces (no caso âncoras ou portas de composições filhas) de seus componentes internos. Essa exportação é realizada através de mapeamentos (representados pelas arestas de *M*, conforme as definições da Seção 2.1), mantendo assim a propriedade de composicionalidade para o documento. Nesses modelos, também similar às ADLs, ligações (*binds*) são estabelecidas entre os papéis de um conector e interfaces dos nós (representadas pelas arestas de *B* no grafo composto). A Figura 2-8 apresenta um exemplo de documento, seguindo o modelo NCM, descrito através de um grafo composto.

Elos hipermídia são definidos por um conector, juntamente com todas as arestas que o tocam. Um elo, dessa forma, pode ser multiponto, ou seja, ligar mais de duas interfaces de nós. Quando as interfaces que se ligam a um conector são portas de um nó composto, os mapeamentos dessas portas em outras interfaces de nós internos, recursivamente, caracterizam os pontos terminais de um elo.

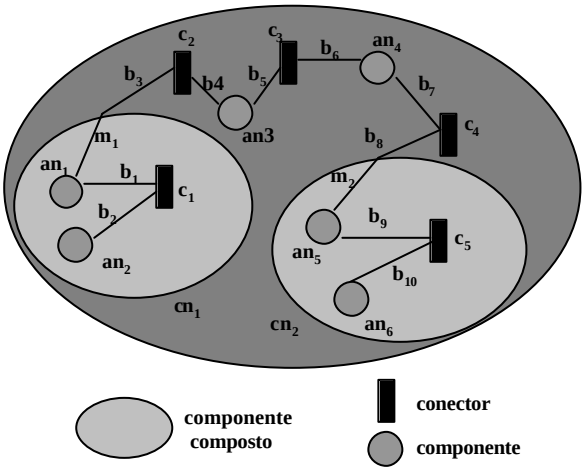


Figura 2-8 Modelo NCM representado em grafos compostos

Em modelos hipermídia onde não existe a noção de conector, mas apenas a de elo (podendo esse elo ser multiponto), um documento pode ainda ser representado por um grafo composto, onde é criado, para cada elo, um vértice

"falso" (como se o vértice representasse um conector inexistente), convergindo para ele todas as arestas ligadas a vértices (representando nós) definidos pelos pontos terminais das extremidades do elo. A Figura 2-9 ilustra o tratamento de elos multipontos sem conectores em grafos compostos.

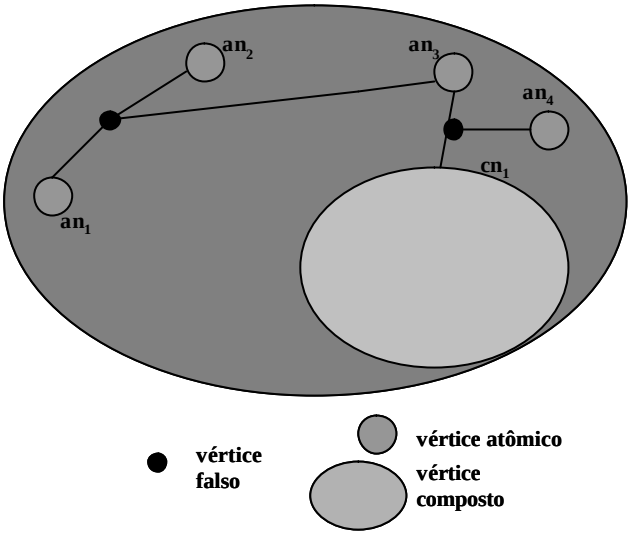


Figura 2-9 Elo multiponto em grafos compostos

Conforme já mencionado no Capítulo 1, os exemplos de documentos hipermídia usados nesta dissertação são, em sua maioria, baseados no modelo NCM e na linguagem NCL (focos iniciais da ferramenta apresentada nesta dissertação). Sendo assim, as próximas duas seções descrevem resumidamente esses conceitos (modelo e linguagem), a fim de oferecer os subsídios básicos para a leitura do restante da dissertação. Maiores detalhes do modelo (Soares et al., 2003) e da linguagem (Muchaluat-Saade, 2003) podem ser obtidos nas referências citadas.

2.3.1. Modelo Conceitual NCM (*Nested Context Model*)

O modelo conceitual NCM, *Nested Context Model* (Soares et al., 2003), representa um documento hipermídia através de um nó. O nó NCM pode ser um objeto de mídia (nó terminal ou de conteúdo) ou um nó de composição. Um nó de composição contém um conjunto de nós, podendo esses nós internos serem objetos de mídia (nós terminais) ou outros nós de composição, recursivamente. Os nós de composição NCM podem conter também elos relacionando esses nós. Uma restrição do modelo impede que um nó contenha, recursivamente, a si próprio, similar à restrição de grafos compostos apresentada no início deste capítulo.

Para permitir o relacionamento entre partes internas do conteúdo de um nó, o modelo define pontos de interface, que podem, por sua vez, ser uma âncora ou uma porta. Âncoras são pontos de interface que podem ser definidos para qualquer tipo de nó, representando intuitivamente um subconjunto marcado de unidades de informação do conteúdo. Portas são pontos de interface que só podem existir em nós de composição. Uma porta de um nó de composição especifica um mapeamento para um ponto de interface de um dos nós internos da composição.

Os tipos de âncoras variam de acordo com o tipo de mídia do nó. No caso de um nó de texto, uma âncora poderia ser uma palavra, já em um nó de áudio contendo uma música poderia ser um determinado intervalo de tempo da música. Uma âncora também pode referenciar um atributo do nó, permitindo que relacionamentos também sejam estabelecidos com base nas características do nó.

Para estabelecer relacionamentos entre nós, é necessária a criação de elos que são agrupados nas bases de elos pertencentes aos nós de composições, conforme comentado anteriormente.

Um elo faz referência a um conector hipermídia (Muchaluat-Saade, 2003) e a um conjunto de associações (*binds*). Um conector representa uma relação sem definir quais nós fazem parte do relacionamento. As relações podem ser de vários tipos, por exemplo: relações de sincronização, referência, derivação, relações entre tarefas de um trabalho cooperativo etc. O conector exporta, através de *papéis*, as interfaces para que os objetos tomem partido na relação, identificando, como o próprio nome sugere, o papel de cada objeto no relacionamento. Dessa forma, os *binds* têm como finalidade associar papéis de um conector a pontos de interface de nós do documento.

A especificação de apresentação de um nó no modelo NCM é feita independente da definição do nó, sendo representada por outro objeto do modelo, chamado descritor (Soares et al., 2000) (Rodrigues, 2003). Os descritores reúnem as informações referentes às características de exibição dos objetos do documento. Os principais atributos de um descritor são: *i)* ferramenta utilizada na exibição do nó; *ii)* região espacial de apresentação, que identifica onde o objeto vai ser exibido; *iii)* informações para ajuste temporal da apresentação do objeto; *iv)* parâmetros diversos que possam ser úteis para caracterizar a exibição (por exemplo, volume de um áudio, sotaque a ser aplicado a uma voz sintetizada etc.).

2.3.2. Linguagem NCL (Nested Context Language)

A NCL (*Nested Context Language*), atualmente na versão 2.0 (Muchaluat-Saade, 2003), é uma linguagem declarativa para especificação de documentos hipermídia.

Devido ao fato de a linguagem estar baseada no modelo NCM, suas características são similares às do modelo, tais como: o uso de composições (nós de composição) para a estruturação lógica do documento, especificação de relacionamentos temporais através de elos e a especificação de relacionamentos espaciais por meio de descritores.

Além disso, a linguagem introduz o conceito de *template* de composição hipermídia (Muchaluat-Saade, 2003), que tem como objetivo facilitar a autoria de documentos NCL, pois possibilita ao autor reutilizar uma estrutura já definida anteriormente. Dessa forma, tipos pré-definidos de composições podem ser tratados como *templates* NCL.

Vale destacar que a linguagem NCL 2.0 foi especificada através de XML *Schema* (XML, 2001) e dividida em módulos, assim como a linguagem xADL e a linguagem SMIL em sua versão 2.0. Tal abordagem facilita a interoperabilidade entre linguagens, permitindo que módulos de uma linguagem sejam incorporados a outra. Por exemplo, o módulo *BasicMedia* da linguagem SMIL é diretamente utilizado pela linguagem NCL. Esse módulo tem como finalidade definir os tipos básicos de mídia que existem em um documento.

A Figura 2-10 ilustra o código de um documento NCL para um melhor entendimento da linguagem.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ncl id="foodNcl" ...>
 <head>
  <layout>
        <topLayout id="w1" ...>
          <region id="image" top=”0” left=”0” .../>
          ...
        </topLayout>
  </layout>
  <descriptorBase>
   <descriptor id=" imagtextD2" region=" image "/>
   ...
  </descriptorBase>
 </head>
 <body>
    <composition id="food">
      <port id="entradaFood" .../>
```

```
        <video descriptor="imagtextD2" id="foodVideo" ...">
        <audio descriptor="audio" id="ingredientesAud" ...">
          <area id="part01Aud" begin="2.0s" end="3.2s"/>
           ...
        </audio>
        <img descriptor="logoDescriptor" id="logoTelemidia" .../>
        ...
        <text descriptor="textD1" id="ovoTxt" .../>
        ...
        <linkBase>
          <link id="link01-01" xconnector="starts.xml" >
            <bind component="ingredientesAud" role="on_x_presentation_begin" />
            <bind component="logoTelemidia" role="start_y"/>
          </link>
           ...
        </linkBase>
    </composition>
  </body>
</ncl>
```

Figura 2-10 Exemplo de documento NCL 2.0

A linguagem NCL 2.0 oferece um elemento *composition*, que permite criar nós de composição (representados por vértices compostos em uma modelagem em grafo). As composições podem conter nós de mídia (vértices atômicos), como por exemplo textos (*text*), imagens (*img*), vídeos (*video*), áudio (*audio*), outros nós de composição (*composition*) e bases de elos (*linkBase*).

As bases de elos contêm os elos (*link*), que, por sua vez, são formados por conectores, identificados na linguagem pelo atributo *xconnector* do elemento *link*, e pela associação entre nós e conectores (*bind*). Cabe comentar que, quando referenciado dessa forma, o conector encontra-se descrito em um outro arquivo.

A linguagem permite que sejam definidos mapeamentos entre as composições e os nós contidos na composição, através do elemento *port*. Os mapeamentos são feitos de portas de uma composição para âncoras (*area*) dos nós de mídia, ou para portas de nós de composição internos.

Para manter a composicionalidade, as associações dos elos (*binds*) devem referenciar exclusivamente portas/âncoras de nós diretamente contidos na composição que contém o elo.

A linguagem NCL também define as características de apresentação dos nós em uma entidade separada, representada pelo elemento *descriptor*. O *descriptor* pode especificar tanto os parâmetros temporais de apresentação como o posicionamento espacial do nó, por exemplo associando um nó a um dispositivo de saída (*region)*.

Com base nos elementos da linguagem NCL definidos anteriormente, as visões temporal, espacial e estrutural dos documento, mencionadas no Capítulo 1, podem ser geradas.

Cabe comentar que, para cada visão gráfica, diferentes elementos da linguagem NCL ganham destaque. Por exemplo, os elementos *topLayout* (conjunto de *regions*), *region* e *descriptor* são os elementos fundamentais utilizados na representação espacial do documento. Já na visão temporal, os elos juntamente com as âncoras presentes nos nós de mídias (por exemplo, nó de áudio) são entidades-chave para a visualização do documento no eixo do tempo. No caso da visão estrutural, os nós de composição ganham destaque, pois são responsáveis pelo agrupamento lógico das entidades presentes no documento.

3
Ferramentas para Edição de Arquiteturas de Sistemas Baseadas em Grafos Compostos

O sistema HyperProp (Soares et al., 2000) foi projetado inicialmente como um sistema para autoria e formatação de documentos hipermídia baseados no modelo NCM – *Nested Context Model* (Soares et al., 2003). Nos últimos anos, o sistema passou por diversas modificações, necessárias para incorporar novas características do modelo. Esta dissertação modificou o sistema de autoria, proporcionando a inclusão de uma nova visão, a visão textual. A visão espacial, apesar de ter sido definida anteriormente em (Costa, 1996) e (Moura, 2001), foi novamente implementada e incorporada à versão mais nova do sistema. Além disso, o ambiente de autoria foi remodelado para a edição de sistemas baseados em grafos compostos de uma forma geral (não apenas documentos hipermídia). Mais ainda, mecanismos de filtragem baseados na técnica olho-de-peixe foram implementados para as visões textual e espacial e atualizados para a visão estrutural.

Neste capítulo, o ambiente de autoria do sistema HyperProp será apresentado com suas quatro visões e, em seguida, o seu mecanismo de sincronização entre as visões será analisado.

3.1.
Edição Textual

Na nova versão do sistema HyperProp foi acrescentada a visão declarativa ao ambiente de autoria, permitindo ao autor editar arquiteturas de sistemas na forma textual (arquivos baseados na meta-linguagem XML) e não apenas na forma gráfica, como anteriormente disponível (Pinto, 2000). A Figura 3-1 ilustra o editor textual integrado ao sistema exibindo um documento NCL.

A interface do editor textual é dividida em duas partes. A área esquerda apresenta a árvore XML (XML, 2000) do documento, na qual os vértices compostos podem ser expandidos (exibindo todos os vértices diretamente

pertencentes a ele) ou colapsados (ocultando todos os vértices diretamente pertencentes a ele). Já a área direita apresenta o documento na forma textual.

O texto apresentado no lado direito do editor pode ser exibido com o recurso de *highlight*, no qual o nome dos elementos ficam em negrito, enquanto os atributos do elemento e seus valores aparecem em cores diferentes. O uso do *highlight* é opcional, de acordo com a preferência do autor.

As alterações feitas em um dos lados do editor são refletidas no outro. Por exemplo, ao se criar um novo elemento na área textual (parte direita do editor), o mesmo será refletido como um novo nó na visão em árvore do documento (parte esquerda do editor). Além disso, ao clicar sobre um determinado nó da árvore na parte esquerda do editor, a linha textual referente ao nó é automaticamente destacada na área direita, como exemplificado com o elemento *"composite id=coisaPele"* na Figura 3-1.

Para sincronizar as alterações realizadas no lado textual com a árvore XML do documento o usuário pode pressionar a tecla *"F5"* do teclado ou clicar sobre o botão na barra de *menu* representado por duas setas.

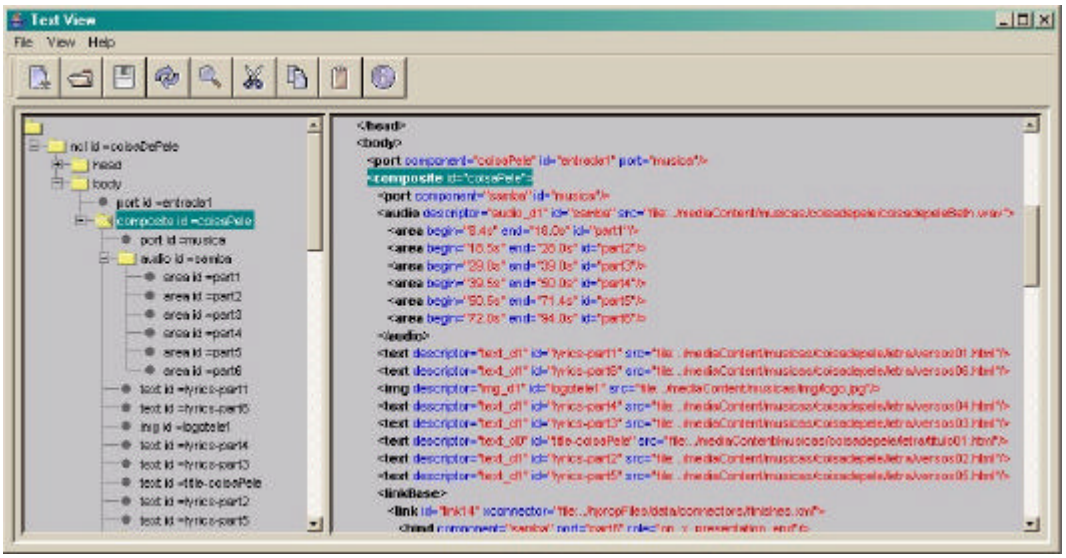


Figura 3-1 Visão declarativa do sistema HyperProp

É importante destacar que, no momento da sincronização da parte textual com a parte esquerda (visão em árvore), a especificação do documento textual é verificada de acordo com seu *Schema*, passado como parâmetro na declaração do documento e, caso nenhuma inconsistência seja detectada, a sincronização é realizada. Se, por acaso, uma inconsistência for detectada, a sincronização é cancelada e uma mensagem de erro é enviada ao usuário.

Uma outra possibilidade de verificar se o documento XML editado pelo usuário está de acordo com a especificação do seu *Schema* é clicando sobre a opção de *menu View* e, em seguida, clicando sobre a opção de *menu Validate.* A Figura 3-2 ilustra um exemplo de validação, no qual o usuário definiu dois elementos *descriptor* com mesmo atributo *"id"*, obtendo uma mensagem de erro identificando o tipo de erro encontrado e a linha na qual o erro se encontra. Caso nenhum erro seja encontrado na especificação, uma mensagem também é exibida ao usuário informando que o documento está de acordo com sua especificação.

Se o usuário solicitar uma validação de documento e este não apresentar a localização do seu *Schema*, uma mensagem de erro será emitida informando que a validação não poderá ser realizada pois o documento não possuiu um *Schema* definido.

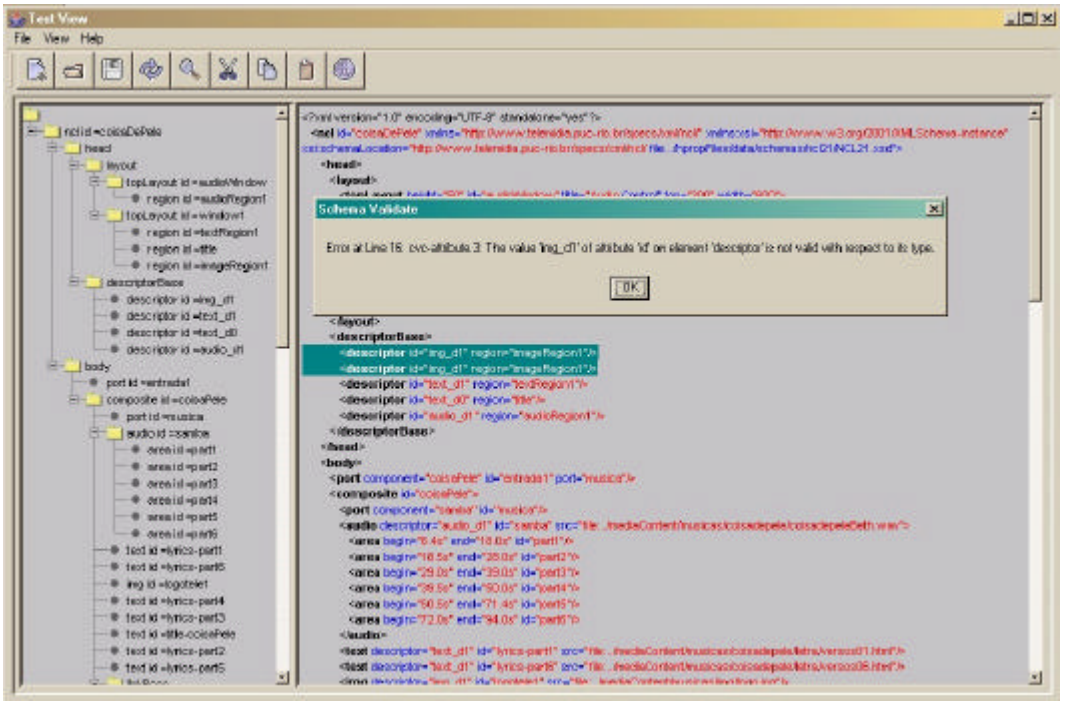


Figura 3-2 Validação de documentos XML

A validação do documento XML foi implementada usando o pacote JAXP (*Java API for XML Processing*) (JAXP, 2003) através da instanciação de um objeto da classe *DOMParser* que disponibiliza o método *parse* para validar documentos XML. O *Schema* do documento a ser validado deve ser indicado dentro do próprio documento XML, no elemento raiz da árvore através do atributo *"xsi:schemaLocation"*.

Para que o método *parse* seja executado, uma *String* deve ser passada como parâmetro do método, indicando a localização do arquivo a ser validado. Neste

momento, a parte textual do editor declarativo é salva em um arquivo temporário e sua localização passada como referência para o método *parse*. As informações de erros encontradas pelo método são tratadas e enviadas para o usuário.

Opções básicas como: copiar, colar e recortar, presentes nos editores tradicionais, também estão disponíveis na ferramenta. Caso algum elemento XML apresente o atributo *"id"*, este será refletido na árvore esquerda do editor. Observa-se na Figura 3-2 que os elementos *"head"*, *"layout"* e "body" da visão em árvore do editor não possuem o atributo *"id"*.

O editor declarativo do sistema HyperProp foi projetado para editar qualquer arquivo no formato XML e não apenas documentos NCL. Como exemplo da generalidade da ferramenta, a Figura 3-3 ilustra um documento GXL - *Graph eXchange Language*, uma linguagem declarativa para edição de grafos compostos (Winter et al., 2002) (Apêndice A), no qual o usuário está editando seu documento.

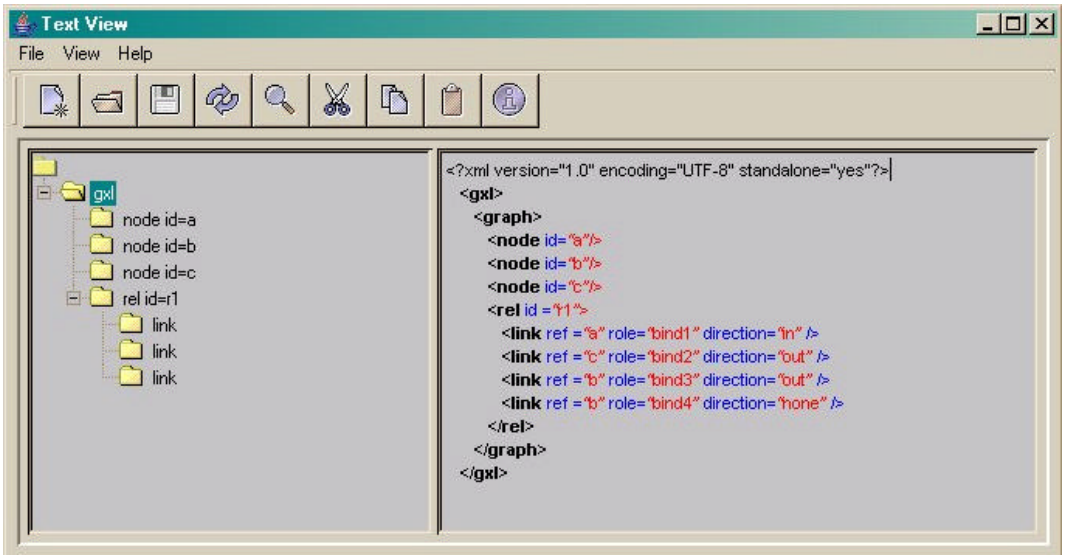


Figura 3-3 Editor declarativo com documento GXL

3.2. Edição Gráfica

Conforme comentado no Capítulo 1, uma outra alternativa para a autoria de arquiteturas de sistemas baseadas em grafos compostos é o uso de ferramentas gráficas. Cada uma das três ferramentas de edição gráfica disponíveis no sistema HyperProp será analisada separadamente nas próximas subseções.

3.2.1. Visão Gráfica Estrutural

A visão estrutural do ambiente de autoria do sistema HyperProp permite ao autor criar a estrutura lógica do grafo composto, ou seja, nela o autor pode criar, editar e apagar vértices (atômicos ou compostos) e arestas.

A interface da visão estrutural está dividida em duas partes, conforme ilustrado na Figura 3-4. O lado esquerdo da visão estrutural apresenta uma árvore com os vértices do grafo composto. O usuário pode escolher os tipos de vértices (atômico, composto ou ambos) a serem visualizados na árvore. No exemplo da Figura 3-4, apenas os vértices compostos são mostrados na visão em árvore do editor. Os vértices compostos podem ser expandidos, mostrando todos os vértices diretamente nele contido, ou colapsados, ocultando todos os vértices diretamente ligados a ele.

O lado direito da visão estrutural mostra todos os vértices (compostos e atômicos) contidos em um vértice composto selecionado na árvore (no exemplo da Figura 3-4, o vértice *coisaPele*). Além disso, o lado direito exibe as arestas que relacionam esses vértices.

Quando o vértice composto selecionado na árvore contém outros vértices compostos, esses últimos podem aparecer expandidos (por exemplo *MusicaeLetra)* ou colapsados (por exemplo, *Cantor*) na área da direita. Quando expandidos, os vértices internos e suas arestas são também exibidas. O processo de expandir vértices compostos pode se repetir nos vários níveis de aninhamento existentes na arquitetura representada.

Sempre que dois vértices são unidos por mais de uma aresta, a ferramenta exibe uma única aresta e coloca um rótulo identificando a quantidade de arestas entre estes vértices. Por exemplo, na Figura 3-4 pode-se observar a existência de uma aresta ligando os vértices *samba* e *lyrics-part6* com o símbolo “(2)”. Neste caso, o símbolo “(2)” indica que existem duas arestas entre os vértices, apesar de apenas uma aresta ter sido desenhada. Já na aresta que liga os vértices *samba* e *photo*, não existe nenhum símbolo sobre a mesma, logo apenas uma aresta relaciona os vértices.

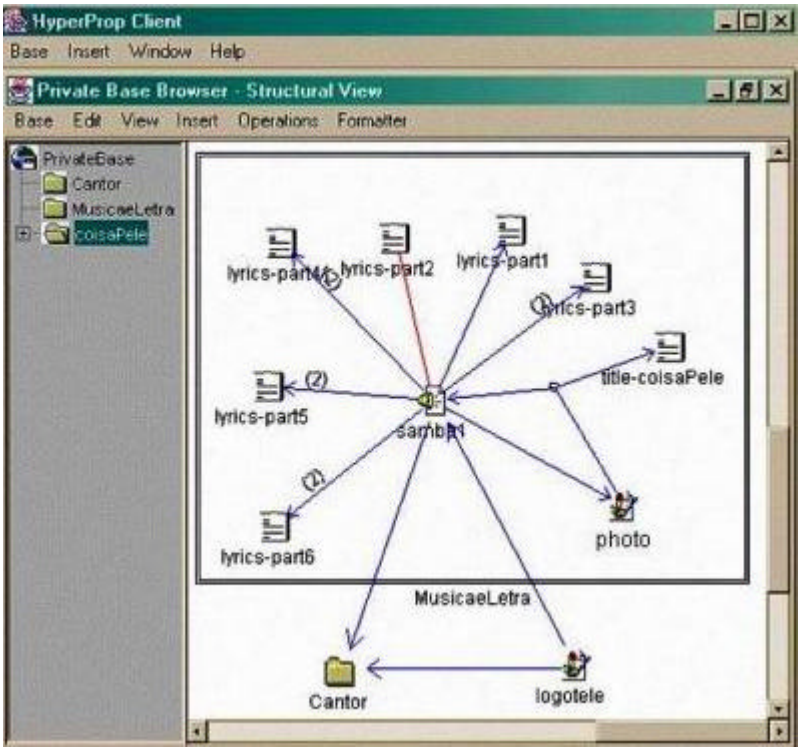


Figura 3-4 Visão estrutural

É importante destacar que a visão estrutural atual não ilustra a representação de portas existentes nos vértices compostos para exportar seus vértices internos. Observa-se, na Figura 3-4, que o vértice composto *MusicaeLetra* não ilustra nenhuma porta no seu desenho (retângulo) quando ocorre um relacionamento entre um vértice interno (*samba*) com um vértice externo (*Cantor*). Conforme a definição de composicionalidade da Seção 2.1, a aresta que relaciona os vértices *samba* e *Cantor* deveria ser ilustrada na Figura 3-4 pela união de duas arestas, uma pertencente ao conjunto M (Seção 2.1) relacionando o vértice composto *MusicaeLetra* com o vértice interno *samba* e outra aresta pertencente ao conjunto B (Seção 2.1) relacionando o vértice *Cantor* com o vértice *MúsicaeLetra*.

Apesar da visão estrutural ilustrada na Figura 3-4 apresentar algumas arestas cruzando a fronteira do vértice composto, na estrutura de dados da visão são definidas as arestas do conjunto M e B (Seção 2.1) para relacionar vértices em diferentes níveis de aninhamento, preservando assim o conceito de composicionalidade.

No modelo NCM, cada elo criado referencia um conector (causal, de restrição etc), podendo este conector ser reaproveitado na criação de novos elos. Para o desenho da visão estrutural (grafo composto), o reaproveitamento de conectores viola uma das restrições de grafos compostos - um mesmo vértice do

grafo não pode estar diretamente contido em mais de um vértice composto (Noik, 1993). A solução adotada foi a definição do "vértice conector", utilizado para a representação do conector referenciado pelo elo. A identificação do "vértice conector", na visão estrutural, é formada pela identificação do elo que o referenciou mais a identificação do conector, garantindo, assim, sua unicidade de representação no grafo composto.

Quando o usuário deseja estabelecer um relacionamento entre apenas dois vértices, o "vértice conector" fica implícito no elo (aresta), não sendo exibido graficamente. No entanto, quando o autor especifica um relacionamento entre três ou mais vértices (vértices *samba*, *photo* e *title-coisaPele*), um "vértice conector" é desenhado e uma aresta é criada de cada um dos nós para o "vértice conector".

Um raciocínio semelhante pode ser aplicado no reaproveitamento de nós no modelo NCM. Quando um nó é reusado em diferentes perspectivas (níveis de aninhamento em vértices compostos), a visão estrutural cria um vértice para cada nó reaproveitado e a identificação desse vértice criado é feita pela junção da perspectiva na qual o nó se encontra mais a identificação do nó.

A visão estrutural também permite que as arestas sejam definidas como sendo direcionadas ou não. No caso do elo causal do modelo NCM, existem condições nos conectores relacionando vértices de origem do elo que devem ser satisfeitas para disparar a execução de ações sobre vértices de destino do elo. A navegação de páginas *Web* é um exemplo de relação causal clássico. Quando o usuário seleciona uma âncora em uma página *Web* (vértice de origem), ele é direcionado para outra página *Web* (vértice de destino). Nesse caso, definir a direção do elo torna-se interessante no desenho do grafo.

Já os elos de restrição do modelo NCM não apresentam nenhuma causalidade envolvida. Por exemplo, um elo de restrição pode especificar que um vértice (por exemplo, nó de áudio) deve terminar sua apresentação ao mesmo tempo que outro vértice começa a dele. Nesse caso, a direção do elo relacionando os dois vértices não faz sentido. Um elo sem direção (elo de restrição no modelo NCM) pode ser visto entre os vértices *samba* e *lyrics-part2* na Figura 3-4.

Várias operações, tais como excluir, editar, expandir vértice composto, fechar vértice composto etc, podem ser realizadas sobre o desenho do grafo. As operações podem ser realizadas tanto com auxílio do *mouse* como através de caixas de diálogo (acionadas a partir da barra de *menu*). Para a criação de novas

entidades (vértices compostos, vértices atômicos, arestas etc.), uma caixa de diálogo é aberta solicitando ao usuário informações específicas da entidade a ser inserida.

A visão estrutural do sistema HyperProp disponibiliza algumas características particulares para visualização de grafos. Por exemplo, os vértices podem ser especializados podendo associar ícones (imagens) aos vértices. No caso do uso do sistema HyperProp para autoria específica de documentos hipermídia baseado no modelo NCM, os ícones dos vértices atômicos que representam objetos de imagens apresentam a cor vermelha, já os vértices atômicos que representam objetos de áudio aparecem com uma caixa de som no ícone, os vértices compostos sempre são representados no formato de pastas amarelas etc.

A biblioteca anteriormente utilizada para representação de grafos na visão estrutural é a VGJ (*Visualizing Graphs with Java*) (VGJ, 1988). Ela foi inicialmente adotada para o desenvolvimento do sistema pois, de todas as bibliotecas estudadas, era a que provia recursos mais apropriados para o desenvolvimento da visão estrutural contemplando grafos compostos. Os principais recursos utilizados da biblioteca VGJ na implementação foi o algoritmo de *Spring* (Kamada & Kawai, 1989), utilizado para desenho dos grafos, e a representação de aninhamento de vértices (vértices compostos), através da qual um subgrafo pode ser representado por um único vértice com o uso da VGJ (Pinto, 2000). A Figura 3-4 foi desenhada usando essa biblioteca.

Atualmente, existem novas bibliotecas para desenhos de grafos que disponibilizam mais recursos para edição de grafos compostos que a biblioteca VGJ. A biblioteca JGraph (Alder, 2003), que trabalha com o padrão de projeto MVC (Model-View-Controller) (Gamma et al., 2002). Em JGraph, o "modelo" são os objetos utilizados para representar as entidades existentes no grafo, a "visão" são objetos responsáveis pela apresentação do grafo para o usuário (características do leiaute de exibição) e o "controlador" são objetos que definem a maneira como a interface do usuário interage com usuário. Tal característica de implementação na biblioteca JGraph facilita a manutenção do código, pois os objetos estão bem divididos e definidos, diferentemente da biblioteca VGJ, onde as características de exibição das entidades estão representadas como atributo das mesmas. Nos próximos parágrafos, algumas diferenças encontradas nas bibliotecas JGraph e VGJ serão apresentadas.

A biblioteca JGraph disponibiliza as principais entidades de grafos presentes na biblioteca VGJ (*graph*, *nodes* e *edge*) e algumas outras entidades interessantes para representação de grafos compostos, tal como a entidade *Port* – objetos pertencentes aos vértices (compostos ou atômicos) que são referenciados pelas arestas (*edge*) para relacionar os vértices. Ou seja, uma aresta não referencia diretamente um vértice e sim uma de suas portas.

A entidade *Port* presente na biblioteca JGraph pode ser facilmente ilustrada nos desenhos de grafos, pois a biblioteca disponibiliza métodos para sua representação nos vértices, diferentemente da biblioteca VGJ que não apresenta a entidade porta como uma de suas entidades, sendo necessário sua extensão para a representação de portas.

A biblioteca VGJ disponibiliza apenas três algoritmos para desenhos de Grafos: *Tree Algorithm, Spring Embedder e Directed Graphs* (VGJ, 1988). Já a biblioteca JGraph disponibiliza oito algoritmos para desenhos de grafos: *Radial Tree, Circle, Moen's Algorithm, Simulated Annealing* e *Sugiyama* além dos mesmos três fornecidos pela biblioteca VGJ.

Uma nova versão da visão estrutural está sendo implementada de forma a torná-la mais flexível e portável na representação de grafos compostos. Para isso, a biblioteca JGraph (Alder, 2003) foi adotada.

A nova visão estrutural que está em desenvolvimento já fornece ao usuário diferentes algoritmos para o desenho de grafos, tais como: o algoritmo de *Spring*, Sugiyama (Sugiyama et al., 1981) e Circular (Wills, 1997). No entanto, alguns pontos ainda se encontram pendentes no desenvolvimentos e serão apresentados nos trabalhos futuros.

3.2.2. Visão Gráfica Temporal

A visão temporal é responsável pela especificação dos relacionamentos temporais entre vértices de um grafo composto, definindo suas posições relativas no tempo. Os vértices de um grafo composto na visão temporal são representados por retângulos, cujos comprimentos indicam suas durações de exibição/execução, conforme ilustrado na Figura 3-5.

A interface desenvolvida é composta por dois eixos: o horizontal, que representa a escala temporal à qual os vértices estão associados, e o vertical, responsável por alocar os vértices a recursos da arquitetura sendo especificada. Um exemplo de utilização da ferramenta para definir arquiteturas de sistemas, seria aquele em que os recursos poderiam representar os processadores de um sistema distribuído, com os componentes alinhados no tempo representando o revezamento na utilização dos recursos.

Para o caso particular da autoria hipermídia, os recursos representam dispositivos (monitor, placa de som etc.) utilizados pelos nós durante a apresentação. No modelo NCM e na linguagem NCL, esse relacionamento entre nós (vértices) e os dispositivos de saída é realizado através dos descritores dos nós, conforme explicado anteriormente na Seção 2.3.1.

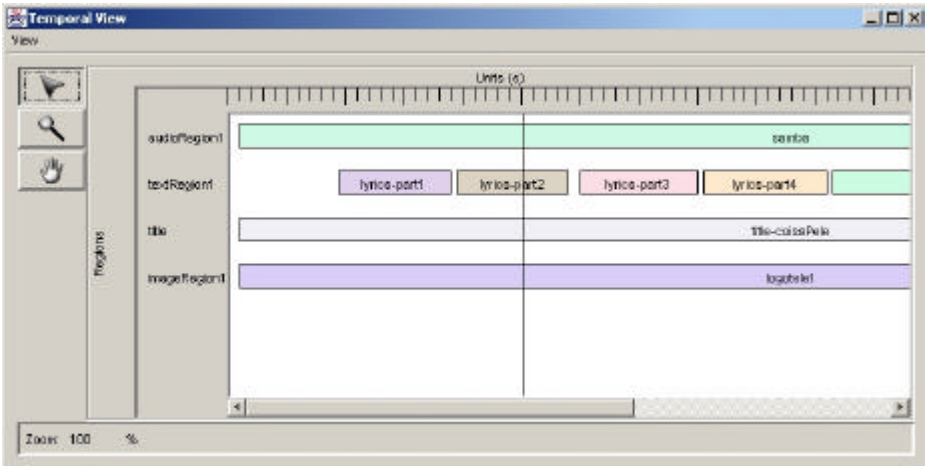


Figura 3-5 Visão Temporal

Na visão temporal, é possível que o autor aplique Zoom (*in* e *out*) no desenho, possibilitando uma visão geral de todo o sistema ou uma visão específica em um determinado ponto da visão. Além disso, a visão temporal possui um eixo vertical móvel, que indica quais vértices estão visíveis no instante de tempo por ele demarcado, refletindo esses mesmos vértices na visão espacial do sistema, conforme será visto na Seção 3.3 (Moura, 2001).

Assim como a visão estrutural, uma nova visão temporal, ilustrada na Figura 3-5, está em desenvolvimento. Entretanto, alguns problemas como: *i)* representação de vértices compostos; *ii)* desenho das arestas entre os vértices; *iii)* a edição dos vértices; e *iv)* integração do filtro olho-de-peixe, ainda não foram resolvidos.

3.2.3. Visão Gráfica Espacial

A visão espacial possibilita ao autor determinar graficamente a disposição dos vértices de um grafo composto com relação aos recursos utilizados na arquitetura de sistema, e como as características espaciais de exibição dos vértices devem variar durante a apresentação de uma arquitetura (Moura, 2001). Ou seja, a visão espacial pode ser definida em dois momentos: o primeiro momento ocorre durante a definição do leiaute espacial da apresentação de uma arquitetura de acordo com os recursos disponíveis e o segundo momento ocorre na especificação de como os vértices são alterados (posicionamentos, tamanhos etc) ao longo da apresentação da arquitetura.

Os recursos disponíveis podem ser analisados como um grafo composto. Por exemplo, um dispositivo de saída "monitor" pode ter sua área de visualização dividida em diferentes regiões, onde cada região pode apresentar diferentes tipos de vértices da arquitetura.

No caso da linguagem NCL, a especificação do leiaute da apresentação do documento hipermídia pode ser representada por um grafo composto onde os vértices compostos são representados pelos elementos *layout*, *topLayout* e *region*. Já os vértices atômicos são representados apenas pelo elemento *region*. Observe que o elemento *region* pode ser representado tanto como um vértice atômico quanto como um vértice composto. Isso ocorre porque a linguagem permite que uma região contenha outras regiões. Maiores detalhes sobre a linguagem podem ser obtidos em (Muchaluat et al., 2003).

Um exemplo de edição de um leiaute espacial para apresentação de um documento hipermídia baseado na NCL pode ser visualizado na Figura 3-6. Nessa figura, o autor selecionou a região *r7* e, em seguida, solicitou a visualização das propriedades da região selecionada. Todas as propriedades são exibidas numa caixa de diálogo. Caso o autor altere o valor de algum parâmetro, por exemplo o parâmetro *Height*, a figura desenhada no editor é automaticamente redesenhada com o novo valor atribuído.

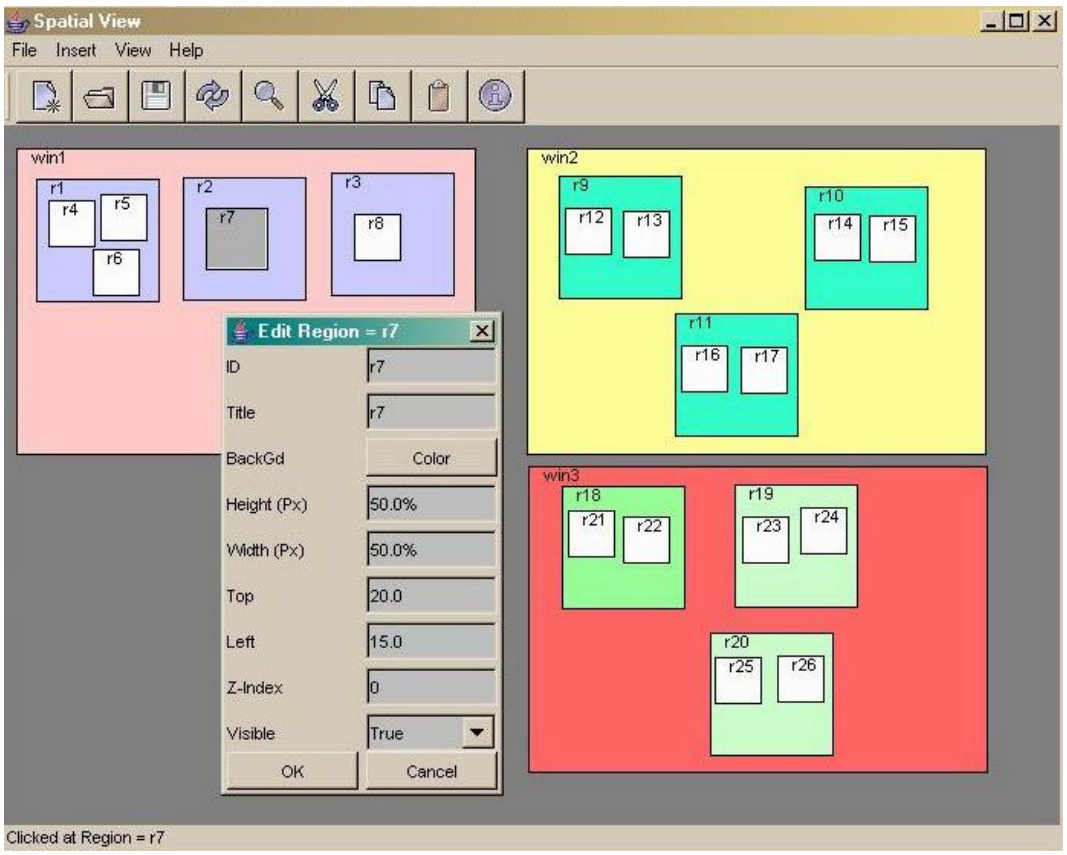


Figura 3-6 Visão Espacial para edição de *Layout* de apresentação.

Quando o símbolo *"%"* aparece em alguma das propriedades *Height*, *Width*, *Top* e *Left* significa que o valor dessa propriedade é proporcional ao valor do elemento espacial que contém a região. Por exemplo, no caso da região *r7*, a propriedade *Height* equivale a "5*0%"* da propriedade *Height* de seu vértice pai (a região *r2*).

A validação dos dados passados pelo usuário é sempre realizada e, caso alguma inconsistência seja detectada, uma mensagem de erro é enviada ao usuário. Por exemplo, se o usuário escrever o valor de um número por extenso ao invés de digitar apenas os números, ou até mesmo, digitar duas vezes o símbolo "**,**" no campo cuja entrada de dados deve ser apenas um valor numérico, uma mensagem de erro será enviada para o usuário informando a incompatibilidade dos dados.

Quando um vértice composto é movimentado, todos os vértices internos a ele são movimentados também, mantendo as posições relativas dos vértices filhos ao vértice pai. Se, durante a edição do leiaute, o usuário movimentar um vértice para fora da área delimitada por seu vértice pai, uma mensagem é enviada para o usuário informando que o vértice está fora da área delimitada por seu pai. Por

exemplo, se o usuário posicionar o vértice *r7* para uma área fora do vértice *r2*, uma mensagem de alerta é enviada.

É importante destacar que, durante a apresentação de uma arquitetura, se um vértice estiver posicionado fora da área delimitada por seu vértice pai, todos os objetos relacionados a esse vértice (recurso) não serão exibidos. Por exemplo, na apresentação de documentos hipermídia, o usuário pode posicionar uma determinada região (destinada para exibição de vídeos) para fora da janela (monitor) impossibilitando sua visualização.

O autor pode, através da interface gráfica, criar, editar e apagar os elementos do grafo composto destinados à apresentação do leiaute da arquitetura, assim como alterar as posições dos vértices com relação a seu vértice pai apenas com o auxílio do mouse.

Em (Moura, 2001), os relacionamentos espaço-temporais dos vértices ao longo da apresentação são analisados. Os relacionamentos espaço-temporais envolvem, além das relações de atributos espaciais, os atributos temporais e/ou ações associadas à apresentação dos vértices do grafo (inicia, interrompe ou termina exibição).

Um exemplo de relacionamento espaço-temporal seria determinar que quando o vértice *"x"* (por exemplo, um nó de imagem) estiver posicionado na posição (20, 30), sua exibição deve ser encerrada. Outro exemplo, seria que, durante a apresentação dos vértices *"x"* e *"y"*, eles devem permanecer alinhados à direita enquanto o terceiro vértice *"z"* estiver sendo apresentado.

Neste trabalho, apenas a especificação do leiaute foi desenvolvido no editor espacial, deixando-se para trabalhos futuros a edição de relacionamentos espaço-temporais.

3.3. Sincronização entre as visões

Para permitir a sincronização entre as várias formas de autoria gráfica, e também o funcionamento das visões gráficas com a edição declarativa de maneira integrada, adotou-se a arquitetura ilustrada na Figura 3-7.

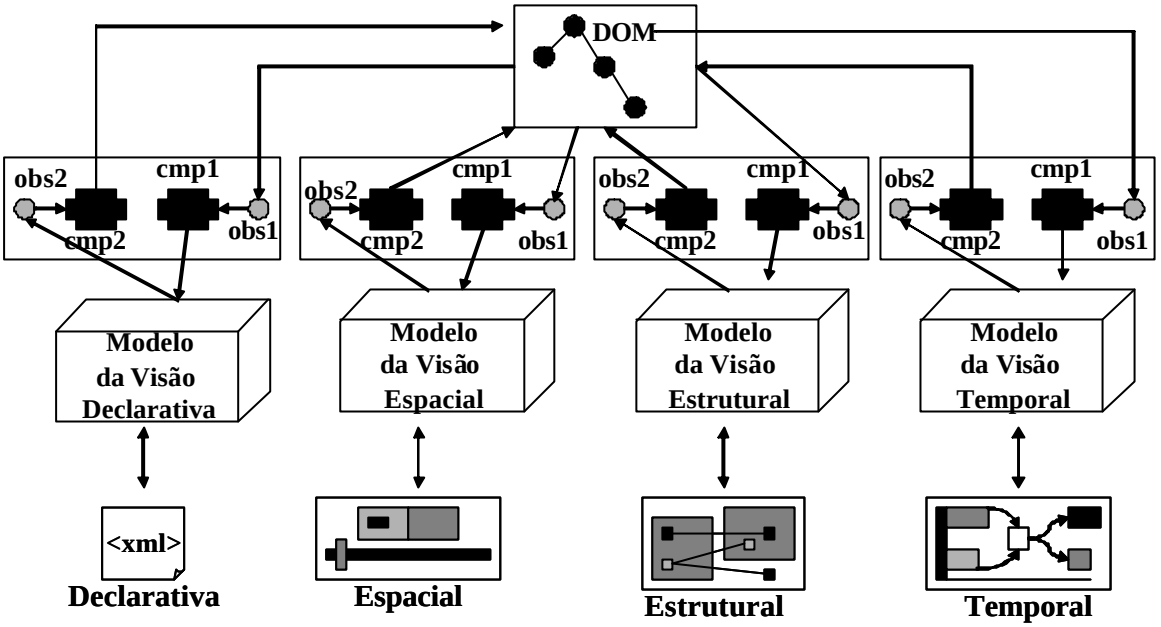


Figura 3-7 – Arquitetura de integração das visões

O modelo de dados utilizado para integração das visões é o DOM – *Document Object Model* (DOM, 2004), padronizado pelo W3C para armazenar e manipular árvores XML, no caso, a árvore que descreve a estrutura do grafo.

Para cada uma das visões, existe um par de compiladores responsáveis, respectivamente, por traduzir a árvore DOM para a estrutura de dados utilizada para exibição do grafo, e por converter essa estrutura de dados na representação do modelo de integração, no caso o DOM.

Alguns compiladores foram desenvolvidos e integrados ao sistema HyperProp neste trabalho, tais como os compiladores que convertem objetos Java do modelo NCM em diferentes visões na estrutura DOM. É importante destacar que o sistema HyperProp já disponibilizava um compilador capaz de converter arquivos NCL (formato declarativo) em objetos Java NCM (Silva et al., 2004). No entanto, somente com o desenvolvimento dos novos compiladores foi possível disponibilizar um ambiente de autoria sincronizado entre as formas de edição gráfica e declarativa no ambiente HyperProp.

Além dos compiladores, cada visão tem um par de observadores (um para cada compilador). Sempre que a árvore DOM é modificada, os *observadores1*, na Figura 3-7, são notificados e acionam os respectivos compiladores para solicitarem as atualizações dos modelos das visões.

De forma análoga, os *observadores2* recebem eventos de mudança do modelo da respectiva visão (resultantes de ações do usuário sobre a ferramenta de

edição) e requisitam que os compiladores correspondentes reflitam essas mudanças no modelo de integração (DOM).

No exemplo do sistema HyperProp, os compiladores de documentos XML foram implementados usando o pacote JAXP (*Java API for XML Processing*) (JAXP, 2003).

Para a visão textual, os compiladores são triviais, uma vez que o modelo da visão textual é também baseado na árvore DOM de documentos XML. Para as visões gráficas, foi necessário desenvolver compiladores apropriados para tradução dos modelos (Silva et al., 2004), uma vez que suas estruturas de dados são baseadas na implementação Java do modelo NCM.

Os módulos que formam os compiladores devem ser específicos para cada arquitetura. Por exemplo, para adaptar o sistema para trabalhar com especificações de arquiteturas de sistemas usando os conceitos de ADL (Capítulo 2), os modelos das visões e os compiladores devem ser adaptados. Evidentemente, como a base da edição são as entidades elementares de grafos compostos, tal adaptação modifica apenas os atributos das entidades básicas (vértices compostos, atômicos, arestas etc.).

A sincronização das diversas visões da ferramenta se dá não apenas pela integração das estruturas de dados dos diversos modelos, mas também pelas suas exibições. Por exemplo, se um vértice é escolhido como foco de exibição em qualquer das visões, ele automaticamente vira o foco em todas as outras visões. Particularmente, se um vértice é escolhido como foco na visão estrutural, ele vira o foco para a exibição de toda a cadeia de acontecimentos, dele derivada, na visão temporal (Costa, 1996).

A Figura 3-8 ilustra um exemplo de sincronização entre a visão espacial atual e a declarativa, onde a região *textRegion1* foi selecionada pelo usuário e automaticamente a linha referente a essa região foi destacada no editor declarativo. Conforme o usuário altera o posicionamento da região *textRegion1*, selecionada no editor espacial, os valores dos atributos referentes ao posicionamento da região no editor textual são imediatamente alterados.

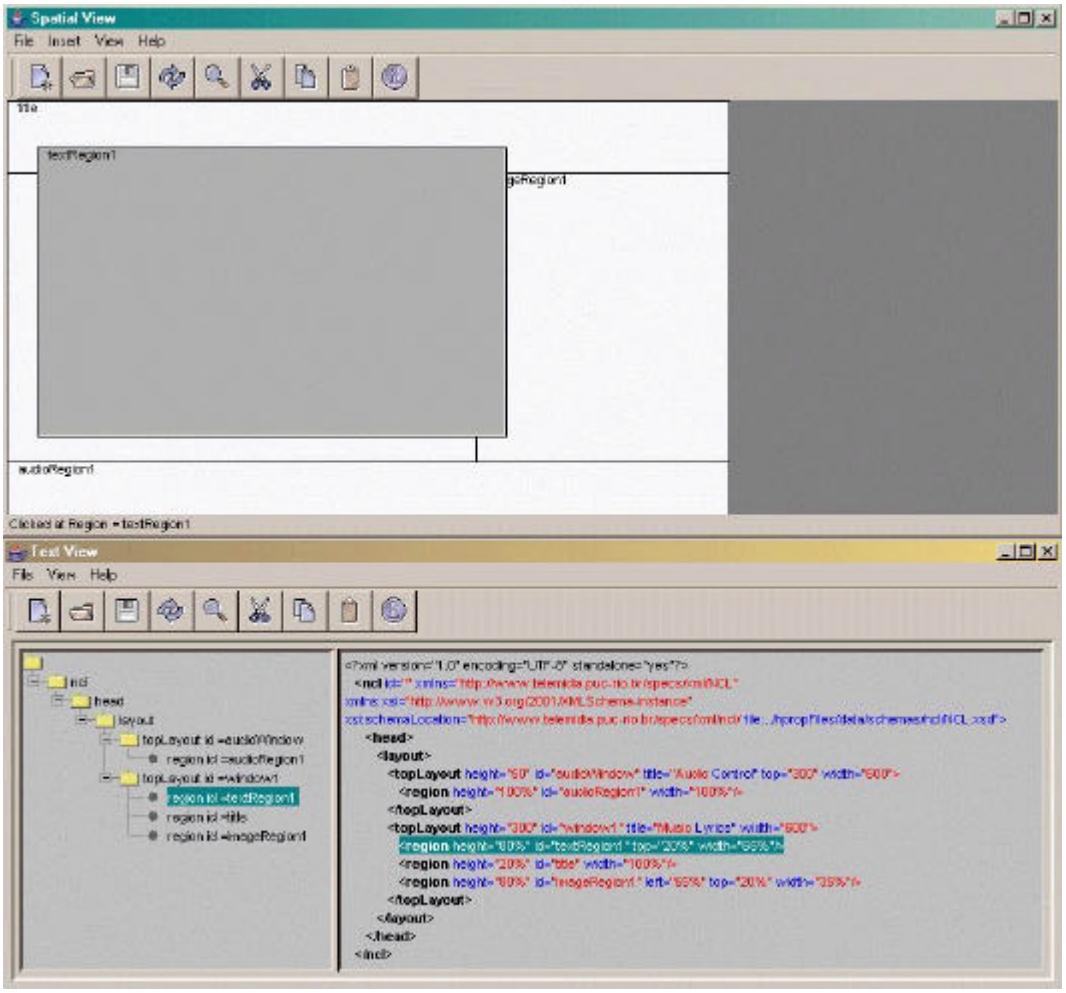


Figura 3-8 Visão Espacial sincronizada com Visão Textual

Um outro exemplo de exibição sincronizada se dá entre as visões temporal e espacial. A visão temporal possui uma barra vertical móvel (vide Figura 3-5), que permite ao autor fixar um instante de tempo para estabelecer alguma análise do sistema sendo especificado. Esse eixo móvel pode ser útil para analisar o comportamento espacial da apresentação em um dado instante (Moura, 2001). Por exemplo, quando o eixo vertical está sobre determinados vértices no instante de tempo *"t1"*, esses vértices são exibidos em seus respectivos dispositivos de saída na visão espacial.

Ainda outro exemplo de visões sincronizadas é discutido no próximo capítulo, quando filtragens na exibição de vértices em uma visão implicam em filtragens nas demais visões.

Uma dificuldade encontrada na sincronização das visões gráficas com a declarativa foi com relação ao texto gerado no editor declarativo, pois os elementos do documento XML exibidos no editor não apresentam nenhuma ordem de classificação (por exemplo, alfabética). Vale destacar que a estrutura

hierárquica dos nós da árvore é mantida. Por exemplo, o usuário pode organizar o documento XML no editor declarativo seguindo a ordem alfabética dos nomes dos elementos. No entanto, quando um evento de sincronização for disparado por uma das visões gráficas, essa organização criada pelo usuário é perdida. Isso ocorre pois a forma na qual o DOM organiza seus os objetos na sua árvore é refletida diretamente na visão textual.

4
Técnicas de Filtragens Aplicadas às Visões do Ambiente de Autoria do Sistema HyperProp

Um problema enfrentado pelos usuários que trabalham com estruturas de dados grandes é a desorientação na busca por determinada informação. Essa desorientação é causada pelo número elevado de vértices e arestas.

Dentre as técnicas mais comuns para a apresentação das estruturas, podem ser citadas o uso de recursos de *zoom* e *scroll* em partes da estrutura. Entretanto, a técnica que se mostra mais eficaz, embora mais complexa, é a filtragem de partes da estrutura não relevantes para o usuário. A dificuldade é identificar quais partes da estrutura realmente são relevantes. Uma solução muito aplicada nos editores de estruturas complexas para filtrar informações é a técnica olho-de-peixe (Furnas, 1986).

Este capítulo está dividido da seguinte forma: na Seção 4.1 será explicada a técnica de filtragem olho-de-peixe e sua extensão para grafos compostos. Para um melhor entendimento da técnica olho-de-peixe utilizada no sistema HyperProp, a Seção 4.2 está dividida em três subseções. A Subseção 4.2.1 apresenta a técnica olho-de-peixe sobre a visão estrutural do sistema HyperProp com base no exemplo ilustrado na Seção 4.1. A Subseção 4.2.2 demonstra o funcionamento da visão olho-de-peixe na visão espacial na especificação de *layout* sincronizada com a visão declarativa. Por fim, a Subseção 4.2.3 analisa os cálculos realizados para filtragem na especificação de *layout* na visão espacial.

4.1.
Técnica de Filtragem Olho-de-Peixe aplicada em Grafos Compostos

A visão olho-de-peixe, proposta por Furnas (Furnas, 1986) para estruturas hierárquicas, atua como uma lente, preservando os detalhes próximos a um ponto escolhido (vértice em foco) e, à medida que se afasta desse ponto, exibindo menos informação (apenas as informações mais importantes segundo critérios que serão explicados no próximo parágrafo).

A estratégia do olho-de-peixe é definir uma função de grau de interesse, que atribui a cada elemento do grafo um valor que representa o grau de interesse do usuário em relação ao elemento em foco. A idéia principal é que essa função grau de interesse (*DOI(x,y)*) aumente com a importância *a priori* (*API(x)*) do elemento *"x"* e diminua com a distância (*D(x,y)*) entre os elementos *"x"* e *"y"*, sendo o elemento *"y"* o foco. Assim, tem-se: ***DOI(x,y) = API(x) – D(x,y).***

A filtragem dos vértices é realizada escolhendo um determinado valor *"k"* para o corte, de modo a exibir somente os elementos *"x"* que possuírem um *DOI(x,y) ? k*. Variando o valor de *"k"*, que pode ser interpretado como o nível de detalhe desejado, obtém-se diferentes visões olho-de-peixe para o mesmo grafo.

A proposta original de Furnas é aplicada em estruturas onde a função *DOI(x,y)* pode ser facilmente definida, como em listas e árvores. Para a utilização da técnica olho-de-peixe no sistema Hyperprop, a mesma teve de ser estendida para ser aplicada em grafos compostos (Muchaluat-Saade et al., 1998), levando em conta tanto as relações de aninhamento como relações de arestas entre os vértices para calcular quais os vértices e arestas são visíveis.

No caso de grafos compostos, a importância *a priori API(x)* de um vértice *"x"* é dada pelo valor negativo do nível de aninhamento do vértice observado em relação ao vértice composto mais externo (no caso, o próprio grafo ou um vértice composto do grafo que tenha sido eleito como o escopo da filtragem[2]). Assim, ***API(x) = -i*** onde " ***i*** " é o nível de profundidade do vértice em relação ao vértice composto eleito como escopo. A Figura 4-1 ilustra um exemplo de cálculo da função *API(x)* para grafos compostos.

[2] Se um vértice composto é eleito como sendo o escopo para filtragem significa que apenas os vértices diretamente ou recursivamente contidos no vértice composto podem vir a ser exibidos. No caso *default*, o próprio grafo pode ser entendido como o vértice composto eleito como escopo.

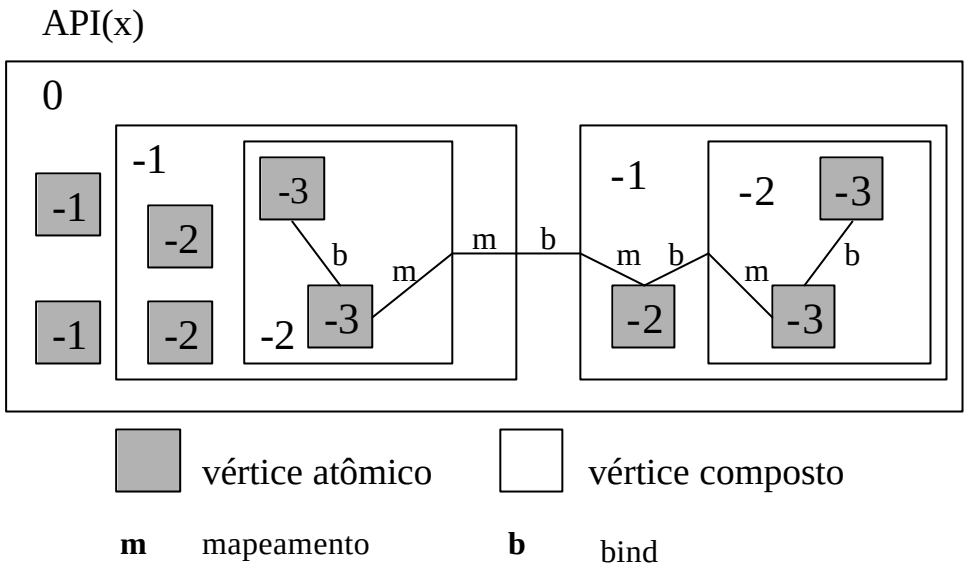


Figura 4-1 Exemplo de cálculo de API(x) para grafos compostos

A segunda componente da função *DOI*, a distância *D(x,y)*, é computada entre um vértice *"x"* e o vértice *"y"* (vértice em foco) da seguinte forma:

D(x,y) = min(Dc(x,y) + wc, De(x,y) + we).

A função *Dc(x,y)* é a distância mínima entre os vértices *"x"* e *"y"* considerando somente a relação de aninhamento das estruturas de composição, que será explicada melhor nos próximos parágrafos, e a função *De(x,y)* é a distância mínima entre *"x"* e *"y"* considerando somente os relacionamentos por arestas. Se não há um caminho de arestas ligando os dois vértices, a distância *D(x,y)* é equivalente a *Dc(x,y)*. As constantes *"wc"* e *"we"* representam os pesos que estes dois métodos de navegação (*Dc(x,y)* e *De(x,y)*) têm no cálculo da visão olho-de-peixe. A diferença entre esses pesos determina uma prioridade na exibição dos vértices relacionados ao vértice em foco pela estrutura de composição ou por arestas.

A distância *Dc(x,y)*, considerando-se a navegação em profundidade (baseada apenas no aninhamento de vértices compostos), do foco *"y"* para o vértice *"x"* é dada por: $\boldsymbol{Dc(x,y) = dist_a + dist_d}$.

O valor de $\boldsymbol{dist_a}$ **é a distância ascendente**, calculada adicionando-se uma unidade para cada passo (passagem de um vértice para seu respectivo pai (vértice composto)) no caminho ascendente desde o vértice "y" (foco) até que o primeiro vértice composto ancestral comum aos vértices *"x"* e *"y"* seja alcançado.

O valor de $dist_d$ é a distância descendente, calculada adicionando-se uma unidade para cada passo (passagem para o vértice filho (composto ou atômico))

no caminho descendente entre o vértice composto ancestral comum encontrado no cálculo da *distância* $dist_a$ e *"y"*. Ao final, se a distância do caminho descendente é maior que zero, soma-se o valor calculado com o valor de $dist_a$. Em outras palavras, $\boldsymbol{dist_d = caminho\ descendente + dist_a}$, **se caminho descendente > 0**. O valor $dist_a$ é usado no cálculo de $dist_d$ para manter a relação de distância do vértice ancestral comum e o vértice com foco. A Figura 4-2 ilustra um exemplo de cálculo da função *Dc(x,y)* para grafos compostos.

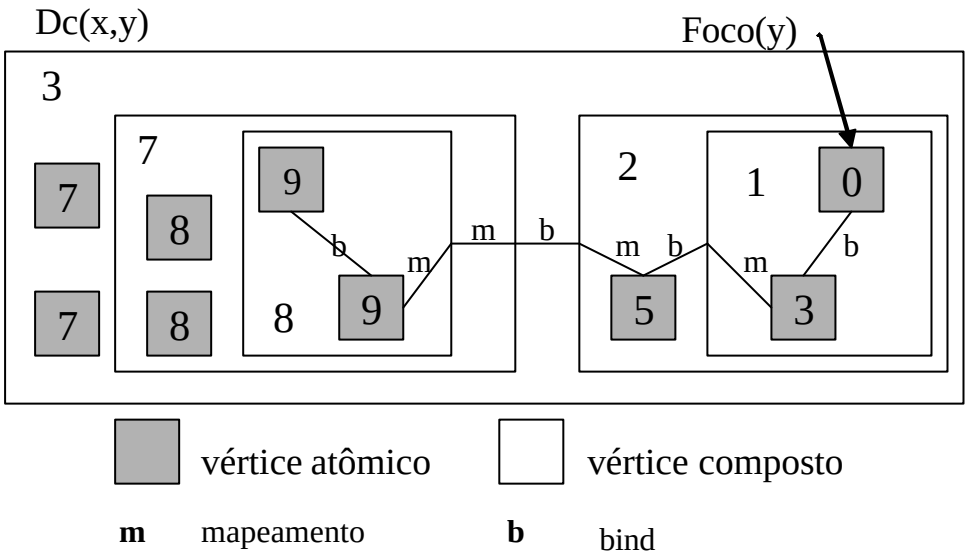


Figura 4-2 Exemplo de cálculo de *Dc(x,y)* para grafos compostos

Para calcular a distância considerando a navegação por arestas *De(x,y)*, deve-se, para cada aresta (*binds* e mapeamentos, vide Seção 2.1) percorrida entre os vértices *"x"* e *"y"*, adicionar uma unidade.

A distância *De(x,y)* será igual ao valor mínimo encontrado entre todos os caminhos possíveis entre *"x"* e *"y"*. Note que pode existir mais de um caminho relacionando os vértices *"x"* e *"y"* por arestas. Para tal cálculo de *De(x,y)*, pode ser usado o algoritmo de Dijkstra (Cormen et al., 2001), que calcula o menor caminho entre dois vértices num grafo não-orientado. A Figura 4-3 ilustra um exemplo de cálculo da função *De(x,y)* para grafos compostos.

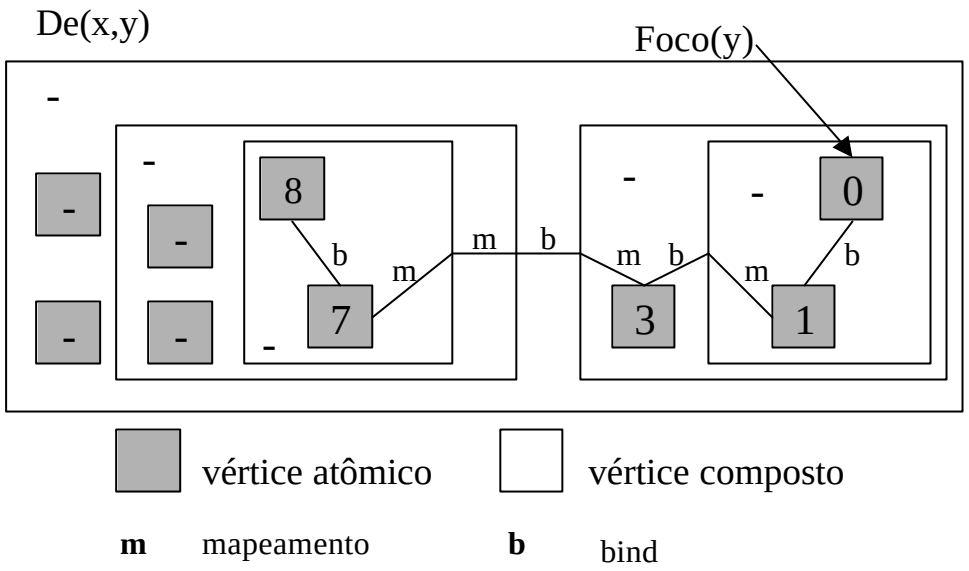


Figura 4-3 Exemplo de cálculo de *De(x,y)* para grafos compostos

Apenas a título de ilustração, a Figura 4-4 apresenta os valores dos vértices considerando a função *D(x,y) = min(Dc(x,y) + wc , De(x,y) + we)* e as distâncias das Figuras 4-2 (*Dc(x,y)*) e 4-3 (*De(x,y)*). Para esse caso em particular, as constantes *wc* e *we* foram atribuídas com valor zero.

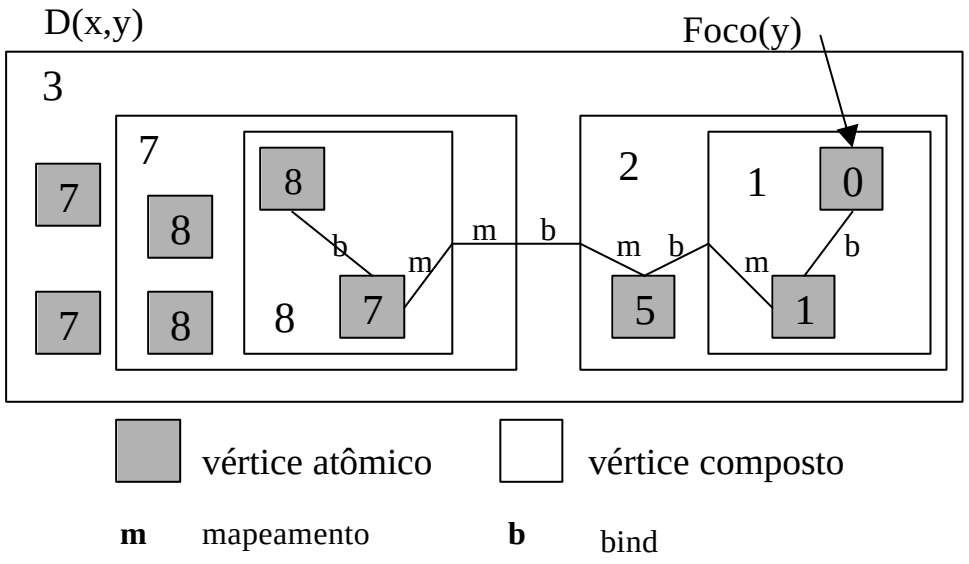


Figura 4-4 Exemplo de cálculo da função *D(x,y)* para grafos compostos baseado nas Figuras 4-2 (*Dc(x,y)*) e 4-3 (*De(x,y)*)

A Figura 4-5 apresenta os valores dos vértices para a função *DOI(x,y) = API(x) – D(x,y)* baseados na Figuras 4-1 (*API(x)*) e 4-4 (*D(x,y)*).

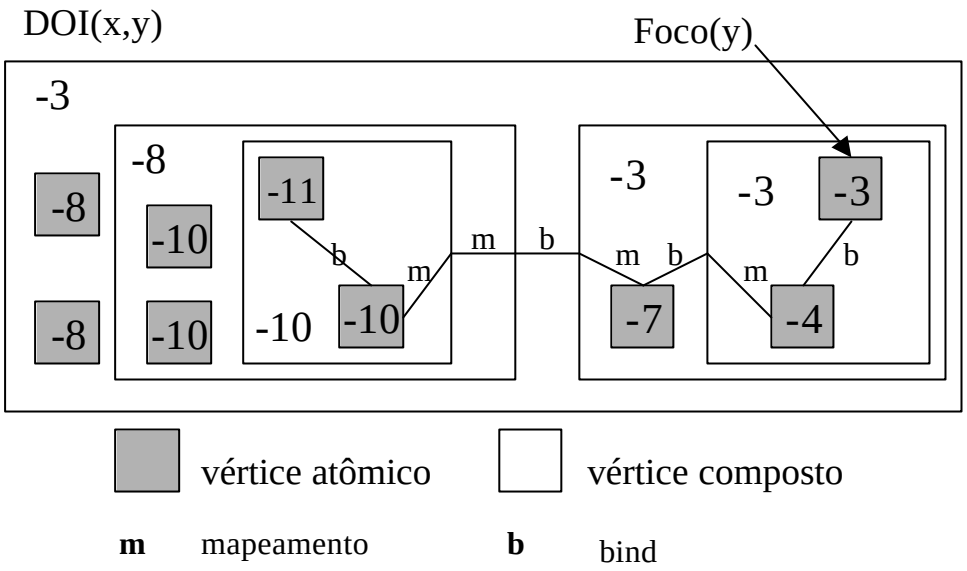


Figura 4-5 Exemplo de cálculo da função *DOI(x,y)* para grafos compostos baseado nas Figuras 4-1 (*API(x)*) e 4-4 (*D(x,y)*)

Caso o usuário escolha o valor de *"k"* igual a menos cinco, por exemplo, somente os vértices com grau de interesse maior ou igual a menos cinco serão visualizados pelo usuário.

4.2. Filtragem Olho-de-Peixe no Sistema HyperProp

O sistema HyperProp, na implementação atual, utiliza a filtragem olho-de-peixe nas visões estrutural, declarativa e na especificação do leiaute na visão espacial, podendo a mesma ser aplicada a qualquer momento durante a edição e navegação dos documentos. Conforme o usuário altera o parâmetro *"k"* ou seleciona um novo vértice como foco, os demais vértices são exibidos e/ou excluídos nas visões espacial, estrutural e declarativa, de forma sincronizada.

4.2.1. Filtragem olho-de-peixe na Visão Estrutural

A técnica de filtragem olho-de-peixe foi reintegrada à visão estrutural do sistema HyperProp e sincronizada com a visão declarativa. Para usar a visão olho-de-peixe na visão estrutural, o usuário deve inicialmente ativar a técnica de filtragem clicando na opção *"View"* da barra de *menu* da visão estrutural e, em

seguida, selecionar na opção *"FishEye"*. Nesse instante, todos os vértices compostos são fechados na visão do grafo.

A Figura 4-6 ilustra a visão estrutural do sistema HyperProp apresentando um grafo composto com todos seus vértices desenhados. Nesse momento, a técnica de filtragem olho-de-peixe ainda não foi aplicada sobre o grafo.

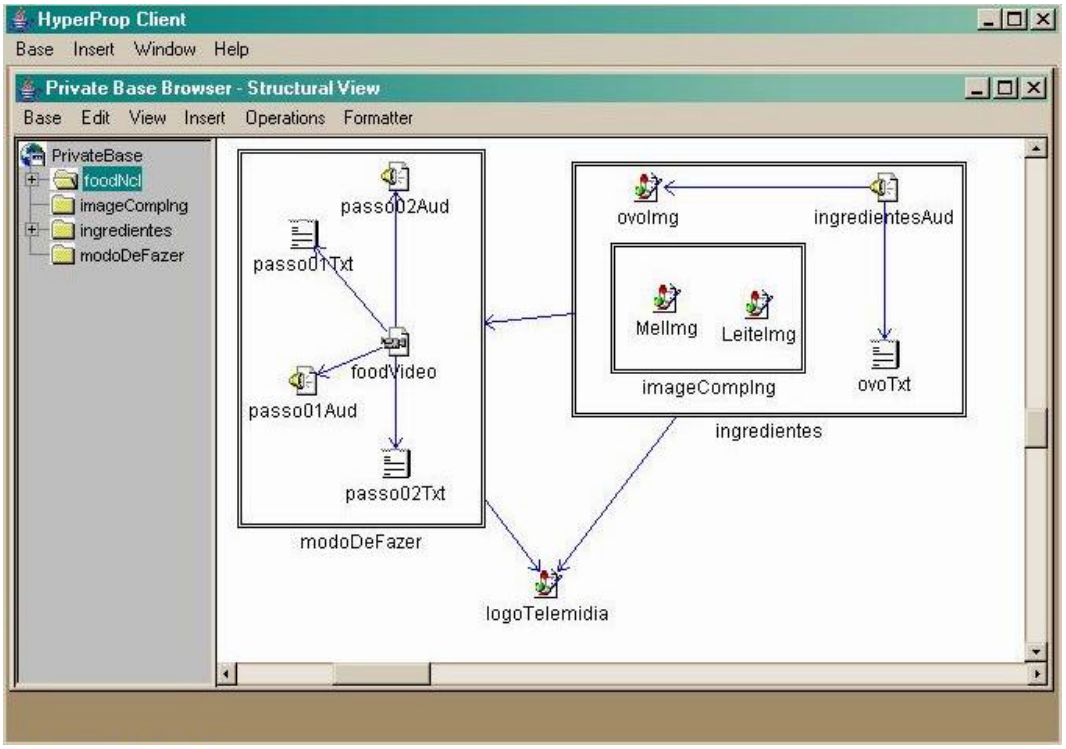


Figura 4-6 Visão estrutural sem a técnica olho-de-peixe

A escolha do vértice a ser utilizado como foco é realizada após a ativação da filtragem. Para definir um vértice como foco, o usuário deve selecioná-lo com o botão direito do *mouse* e, em seguida, escolher a opção *"Focus"* entre as alternativas de *menu* exibida sobre o vértice. Os parâmetros de configuração (*k*, *wc* e *we*) da visão olho-de-peixe podem ser alterados através da janela *"FishEye Configuration"*.

Para exibir a janela *"FishEye Configuration"* o usuário deve selecionar a opção *"View"* da barra de *menu* da visão estrutural e depois selecionar a alternativa *"Configure"*.

Observe na Figura 4-7 que em um determinado momento da navegação o usuário escolheu como foco o vértice *"melImg"*. Em seguida, ativou a janela *"FishEye Configuration"* para configurar a visão olho-de-peixe com nível de detalhe no valor de *"25%"* de sua visualização inicial. Isto fez com que apenas os vértices *melImg*, *imageCompImg* e *ingredientes* fossem exibidos.

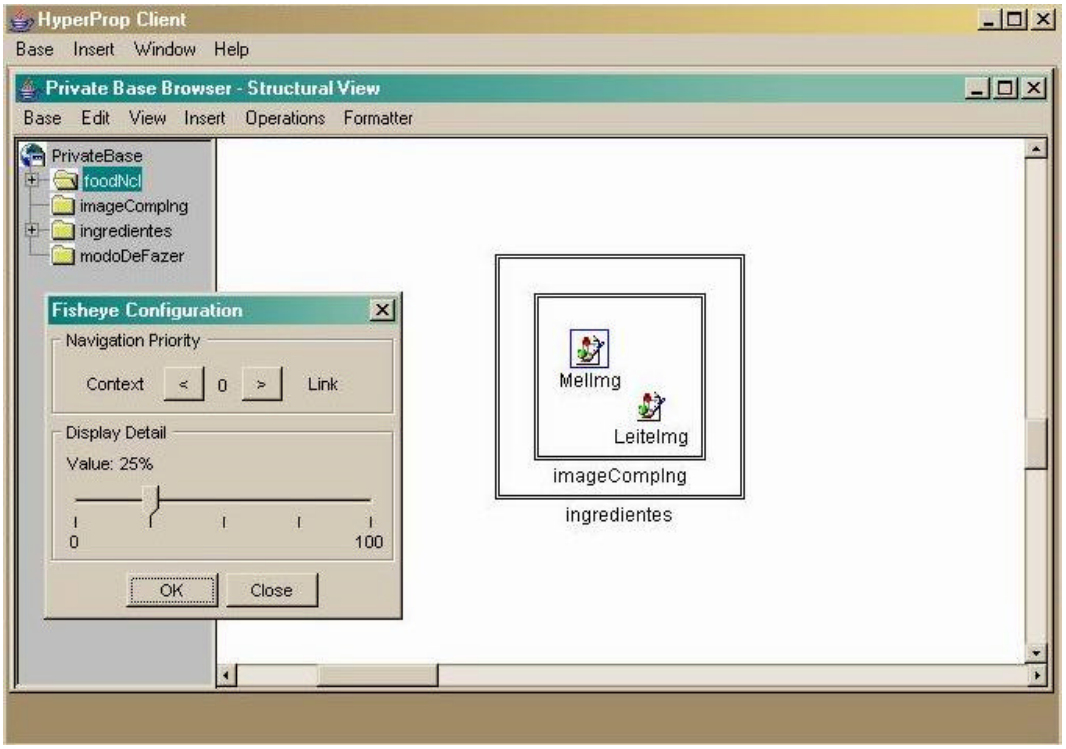


Figura 4-7 Visão olho-de-peixe com 25% de *Nível de detalhe*

Quando o usuário ativa visão olho-de-peixe, os vértices exibidos na visão estrutural são automaticamente apresentados na visão declarativa (na parte direita do editor) com tamanhos variados de texto, diferenciando-os de acordo com a respectiva distância ao foco.

Já a parte esquerda do editor declarativo apresenta apenas os vértices exibidos na visão estrutural. A Figura 4-8 mostra a visão declarativa sincronizada com a visão estrutural.

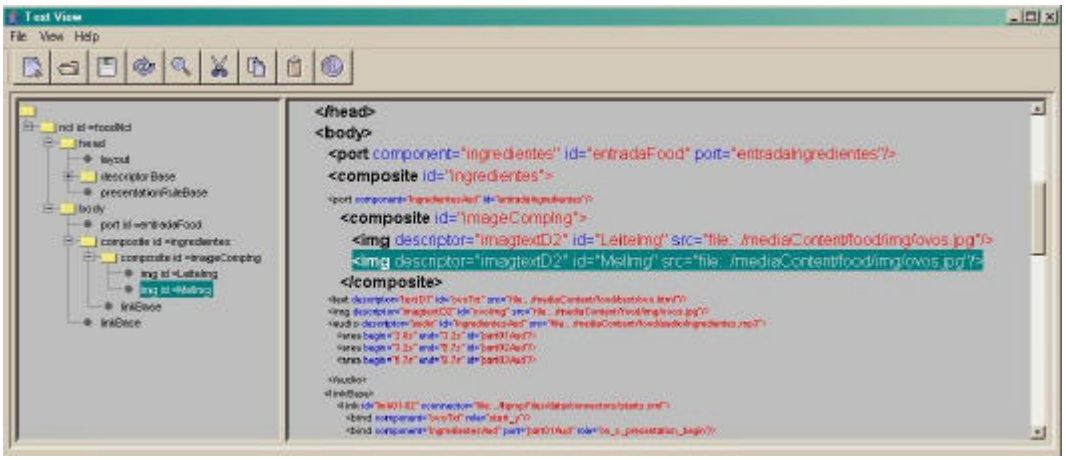


Figura 4-8 Visão declarativa sincronizada com a visão estrutural (olho-de-peixe 25%)

Conforme o usuário navega no grafo ou altera os parâmetros de configuração da visão olho-de-peixe na janela *"FishEye Configuration"*, as novas

informações exibidas na visão estrutural são refletidas na visão declarativa de forma sincronizada.

A Figura 4-9 ilustra o grafo na visão estrutural com *nível de detalhe* no valor de *"75%"* de visualização e a Figura 4-10 mostra a visão declarativa sincronizada com a visão estrutural da Figura 4-9. Observe a diferença de tamanho nas fontes de texto na parte direita do editor.

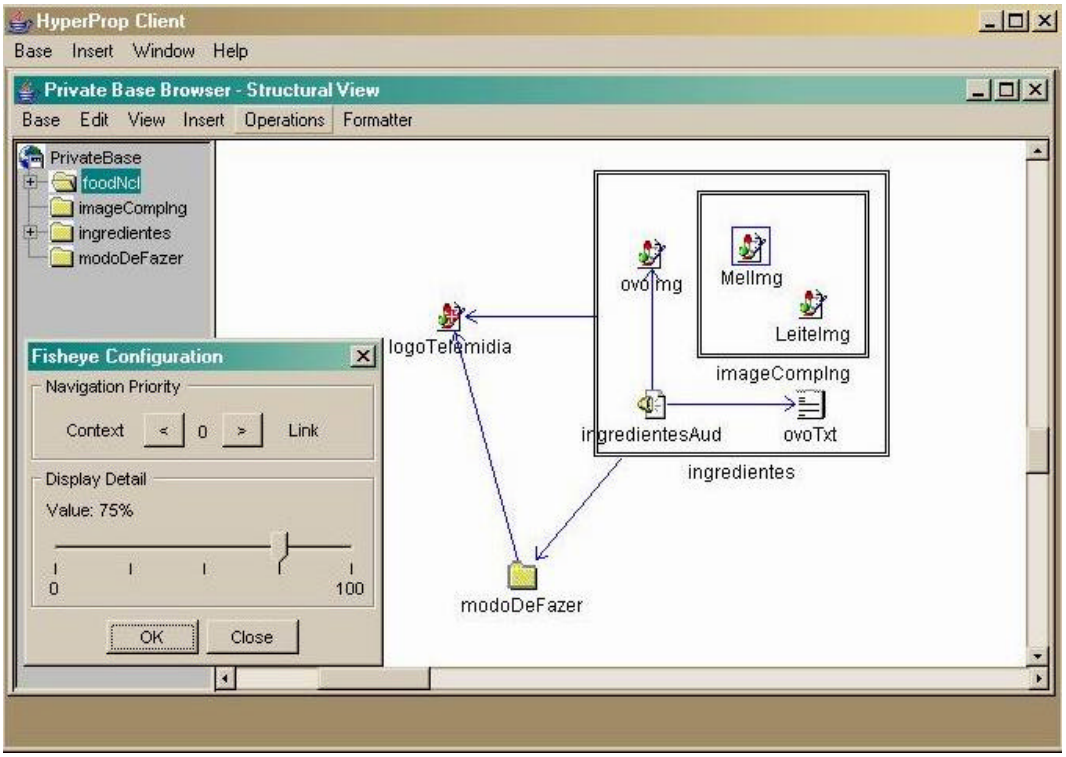


Figura 4-9 Visão olho-de-peixe com 75% de *Nível de detalhe*

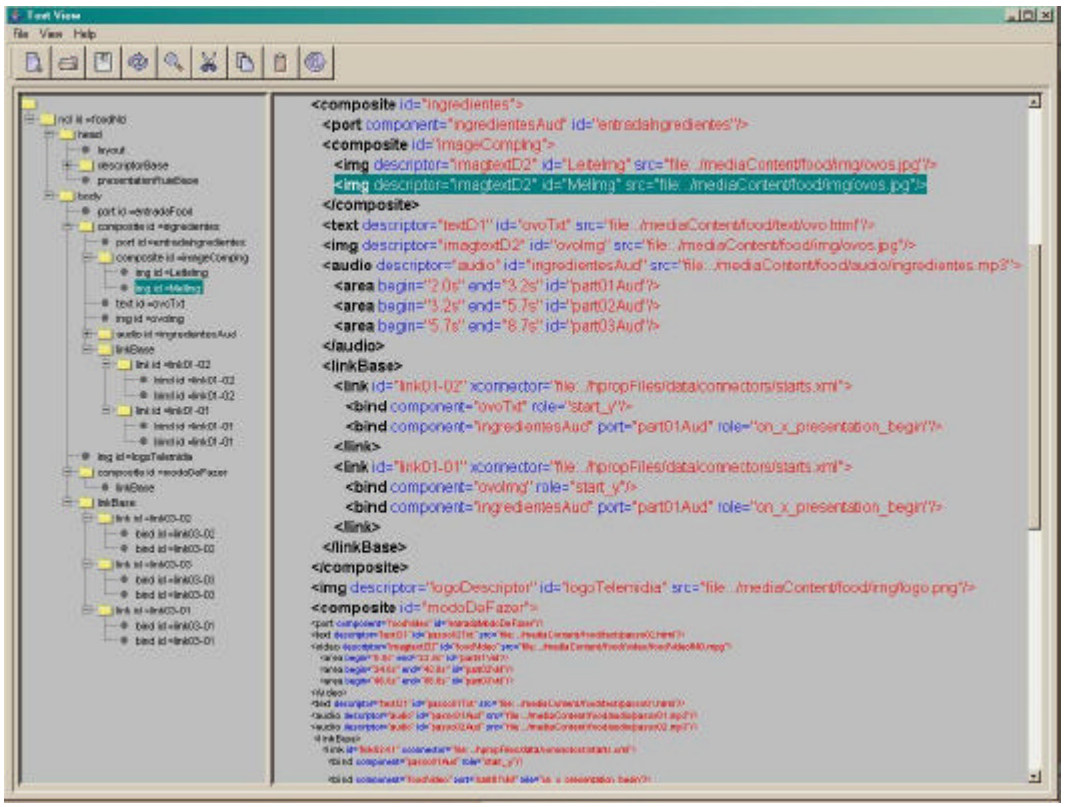


Figura 4-10 Visão declarativa sincronizada com visão estrutural (olho-de-peixe 75%)

4.2.2. Visão olho-de-peixe na Visão Espacial e Textual

Conforme já apresentado na Seção 3.4, durante a edição espacial, o usuário pode especificar o leiaute da apresentação espacial de uma determinada arquitetura de acordo com os recursos disponíveis.

A Figura 4-11 ilustra o editor espacial na especificação de um leiaute para apresentação de documentos hipermídia baseado na linguagem NCL. Nessa figura, o autor definiu três elementos *topLayout* (“*w1*”,“*w2*” *e* “*w3*”) com vários filhos internos a eles (*regions*). Além disso, escolheu como foco o elemento *region* “*r8*”. A mesma informação apresentada na visão espacial da Figura 4-11 pode ser visualizada na visão textual conforme a Figura 4-12.

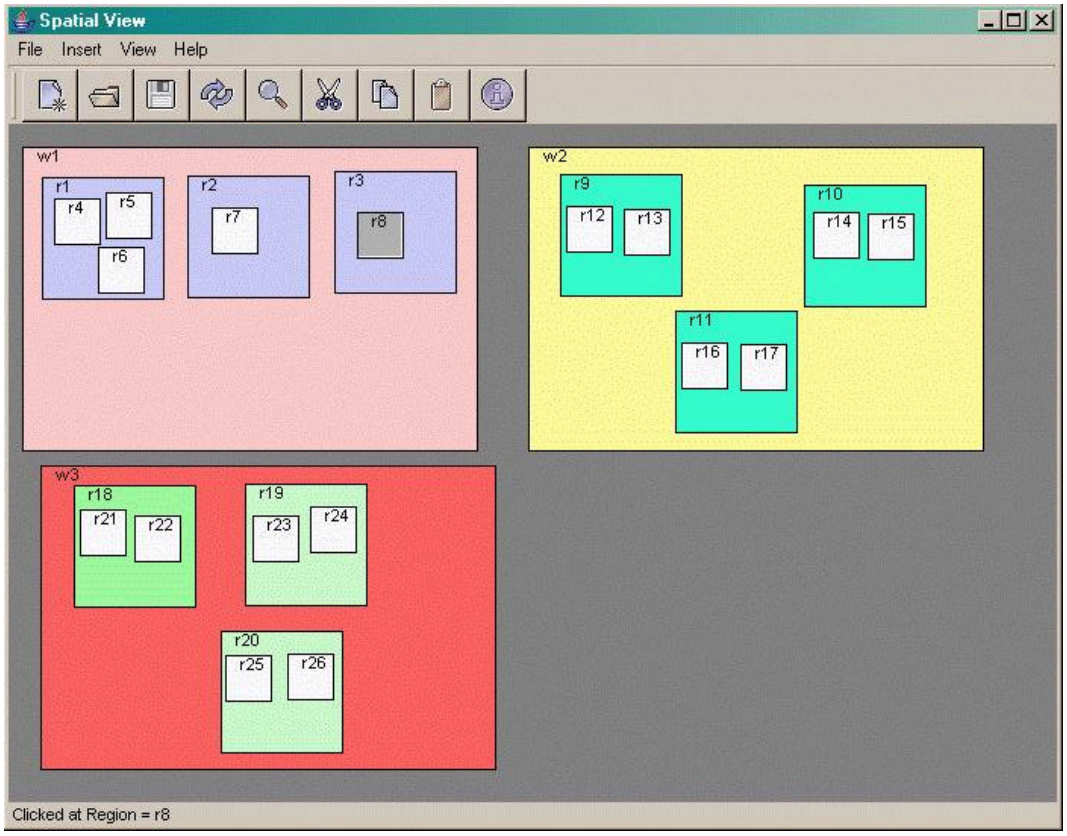


Figura 4-11 Visão Espacial da especificação do elemento *layout* na linguagem NCL

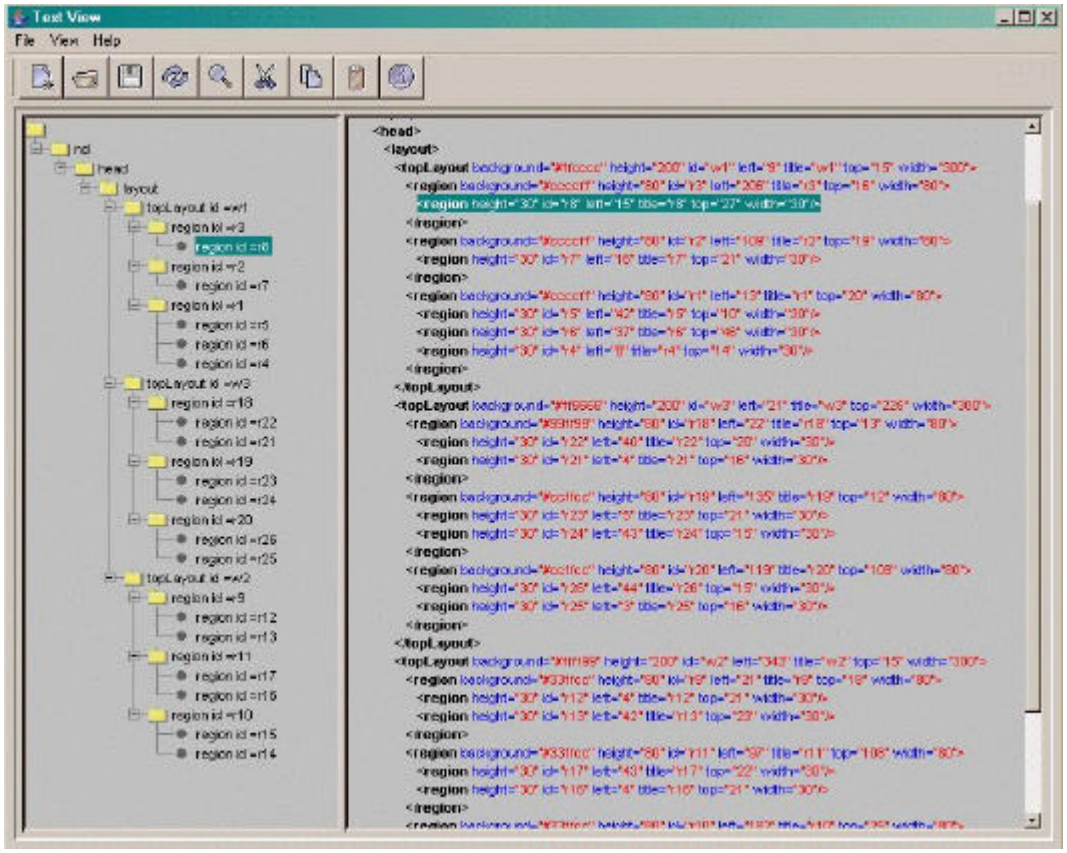


Figura 4-12 Visão Textual da especificação do elemento *layout* refletida da Figura 4-11

Para aplicar a técnica olho-de-peixe na edição do leiaute na visão espacial, o usuário inicialmente seleciona um vértice como foco e, em seguida, ajusta o parâmetro *nível de detalhe* (parâmetro *"k"*) na janela *"FishEye Configuration"*. O filtro aplicado na visão espacial é refletido na visão textual. A Figura 4-13 ilustra a visão espacial com a filtragem olho-de-peixe mantendo a região *"r8"* como foco e usando um *nível de detalhe* em aproximadamente 50%.

Conforme o usuário altera o *nível de detalhe* (parâmetro *"k"*) ou seleciona um novo vértice como foco, a filtragem é automaticamente reavaliada e a visão atualizada.

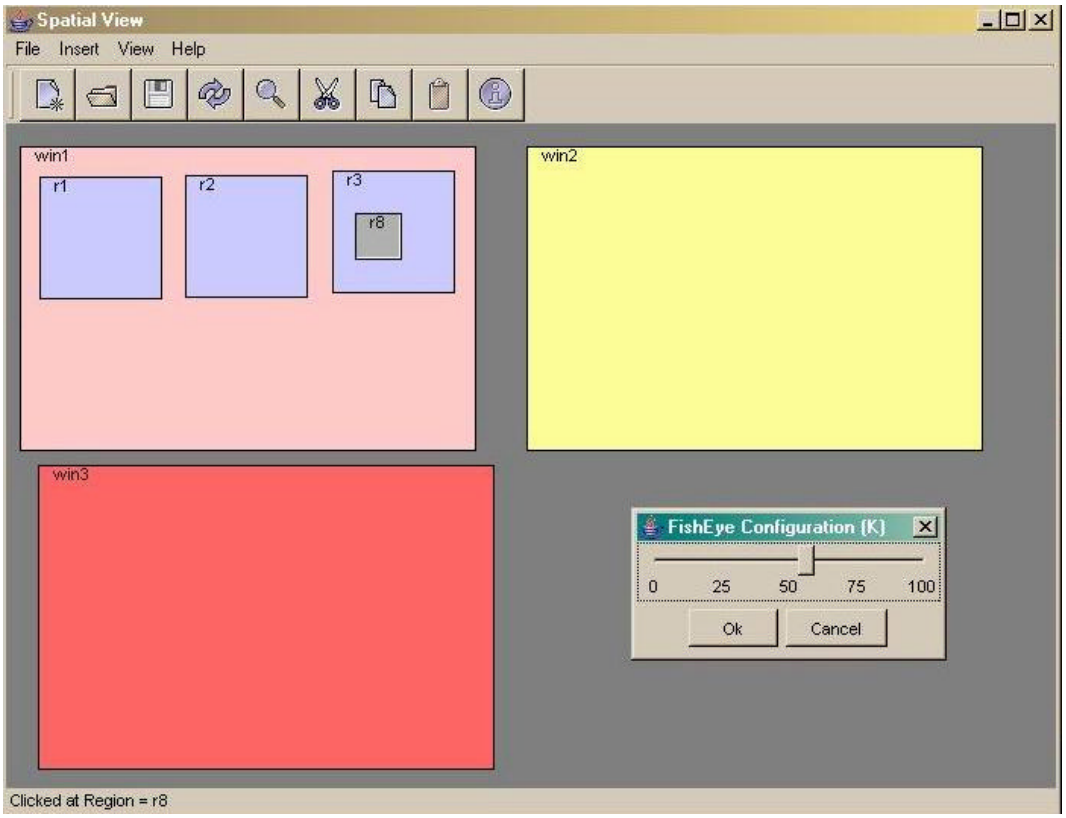


Figura 4-13 Visão olho-de-peixe aplicada na visão espacial

A Figura 4-14 ilustra a visão textual sincronizada com a visão olho-de-peixe aplicada sobre o editor espacial da Figura 4-13. Observe que, na visão textual, os vértices não ilustrados no editor espacial da Figura 4-13 são excluídos da parte esquerda do editor (visão em árvore), enquanto que, na área direita (texto), os elementos são mostrados com tamanhos variados de texto, diferenciando-os de acordo com a respectiva distância ao foco. Quanto mais próximo do foco o vértice estiver, maior será o tamanho do texto exibido no editor.

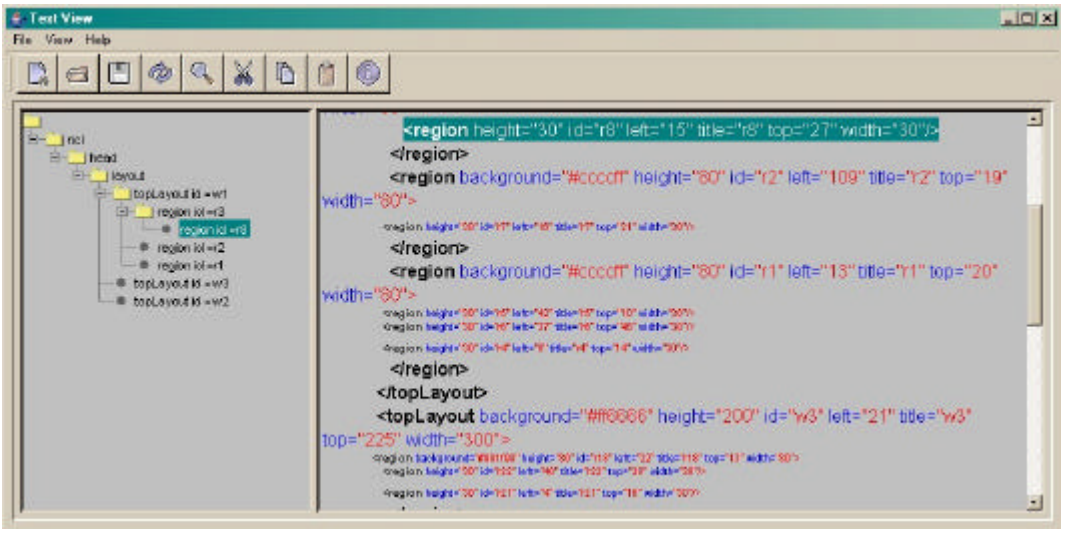


Figura 4-14 Visão olho-de-peixe aplicada na visão textual

4.3. Cálculos Realizados na Filtragem Olho-de-Peixe Sobre a Visão Espacial

Conforme mencionado na Seção 4.1, a técnica olho-de-peixe pode ser aplicada na forma original proposta por Furnas (Furnas, 1986) para estruturas simples como **árvores** e listas.

Os relacionamentos de inclusão entre os elementos presentes na linguagem NCL (*layout*, *topLayout* e *region*) visualizados na Figura 4-12 podem ser tratados como uma **árvore**, onde a raiz da árvore (vértice composto) é o elemento *layout* da linguagem NCL.

É importante destacar que o algoritmo olho-de-peixe aplicado sobre a especificação de **um leiaute na visão espacial** foi aplicado na forma **original** proposta por Furnas, diferente da **visão estrutural**, em que o algoritmo foi adaptado para grafos compostos conforme discutido na Seção 4.1.

O cálculo realizado na especificação do leiaute da visão espacial da Figura 4-12 (foco sobre o vértice *"r8"*) procedeu nas seguintes etapas:

1. calcular os valores da função API(x) - no caso da estrutura em árvore, cada elemento pertencente à árvore recebe um valor **negativo** referente a seu nível de profundidade (**API(x) = - i**). A Figura 4-15 ilustra os elementos da visão espacial com seus respectivos valores calculados;

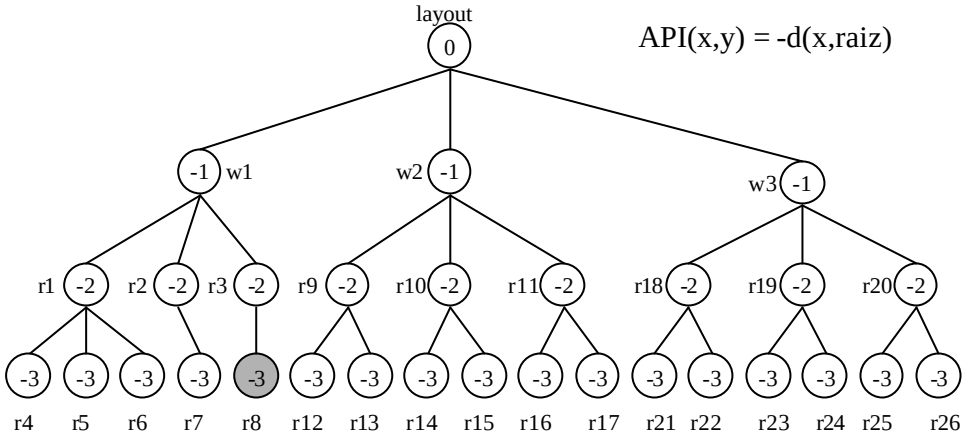


Figura 4-15 Cálculo da função API(x,y)

2. calcular os valores da função D(x,y) - no caso da distância do vértice *"x"* em relação ao vértice *"y"* (foco), soma-se uma unidade para cada aresta entre o vértice em foco (*"r8"*) e o vértice de destino. A Figura 4-16 ilustra os elementos com seus respectivos valores calculados;

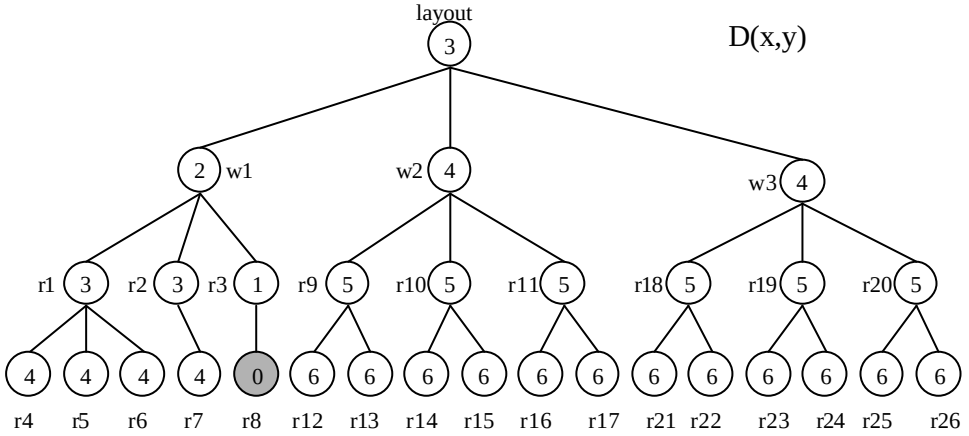


Figura 4-16 Cálculo da função D(x,y)

3. calcular os valores da função DOI(x,y). Para estruturas em árvores foi utilizado *DOI(x,y) = API(x) – D(x,y)*. A Figura 4-17 ilustra os elementos com seus respectivos valores calculados;

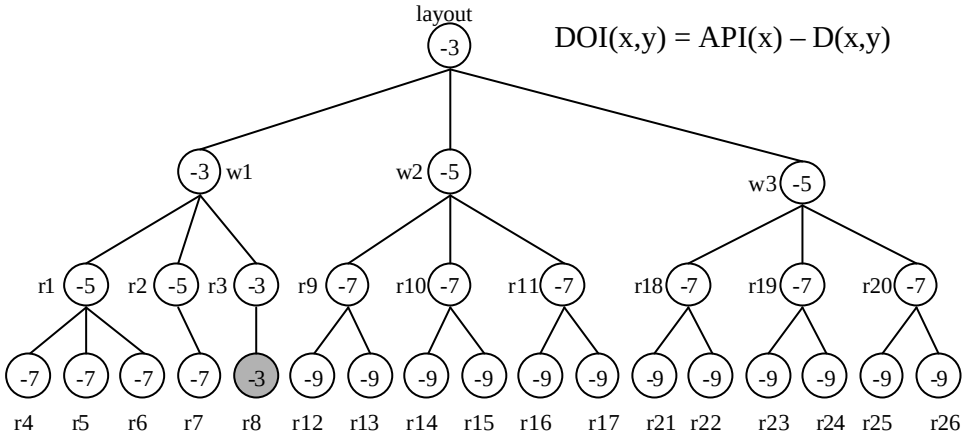


Figura 4-17 Cálculo da função DOI(x,y)

4. selecionar quais elementos devem ser exibidos: Conforme o usuário altera o *nível de detalhe* um novo parâmetro *"k"* é definido como corte. Para *"k"* igual *"-5"*, somente os elementos com valores *DOI (x,y)* maiores ou iguais a *"-5"* tornaram-se visíveis. A Figura 4-18 ilustra a árvore filtrada.

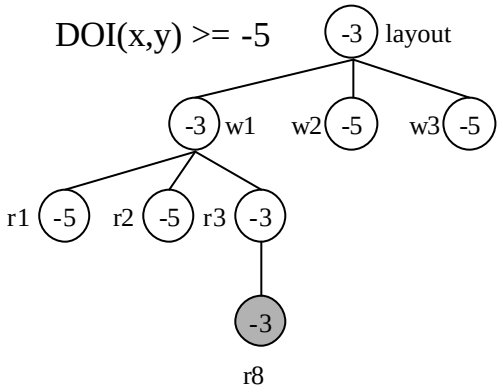


Figura 4-18 Elementos visualizados com *k* ? -5

5
Trabalhos Relacionados

Este capítulo compara vários trabalhos relacionados à autoria de grafos e documentos hipermídia com o editor proposto nesta dissertação. Com o objetivo de organizar as comparações, o capítulo está estruturado nas seguintes seções: a Seção 5.1 apresenta o sistema GraphViz (Gansner, 2003) para autoria e visualização de grafos. O editor XML (XML, 2000) Salix (Aoki & Seraphim, 2003) é analisado na Seção 5.2. A Seção 5.3 apresenta o sistema Kaomi (Jourdan et al., 1999) para autoria de documentos hipermídia. A Seção 5.4 discute as principais características do sistema GRiNS (Bulterman et al., 1998), ferramenta para autoria de apresentações multimídia baseada em documentos SMIL (SMIL, 2001). Por fim, a Seção 5.6 analisa as ferramenta Ariadne (Jühne et al., 1998) e o ambiente Arakne (Bouvin, 2000).

5.1.
GraphViz

GraphViz (Gansner, 2003) é um sistema para visualização de grafos desenvolvida pela *AT&T*, distribuído na comunidade através da licença "*AT&T License*". A linguagem de programação utilizada no desenvolvimento da ferramenta foi *C++ ANSI* juntamente com *TCL/Tk 8.3* (*Tool Command Language*) (TCL/TK, 2004), sendo assim, portável em outras plataformas como Unix, (Linux, Solaris, IRIX, AIX, BSD), Windows, Macintosh etc.

A ferramenta fornece três construtores (algoritmos) para desenhos de grafos. Basicamente, todos os construtores trabalham da mesma forma, tendo como entrada um arquivo textual na linguagem *dot* (Gansner et al., 2002) e como saída o desenho do grafo em um arquivo, podendo o usuário escolher o formato de saída (gif, png, jpeg, PostScript etc.) do mesmo.

O *parser* utilizado na ferramenta para converter documentos criados na linguagem *dot* em figuras encontra-se nas bibliotecas *libgraph* e *libagraph*. Caso alguma inconsistência seja encontrada durante o processamento do arquivo fonte

(arquivo *dot*), uma mensagem de erro é enviada ao usuário informando o número da linha do arquivo onde o erro se encontra.

Existem algumas ferramentas disponíveis na Internet (GraphViz, 2002) para converter arquivos textuais de representação de grafos em arquivos na linguagem *dot*, tal como a ferramenta *GXL2DOT*, que recebe como entrada um arquivo na linguagem GXL - *Graph eXchange Language* (Winter et al., 2002) (Apêndice A). A Figura 5-1 ilustra a interface principal da ferramenta GraphViz.

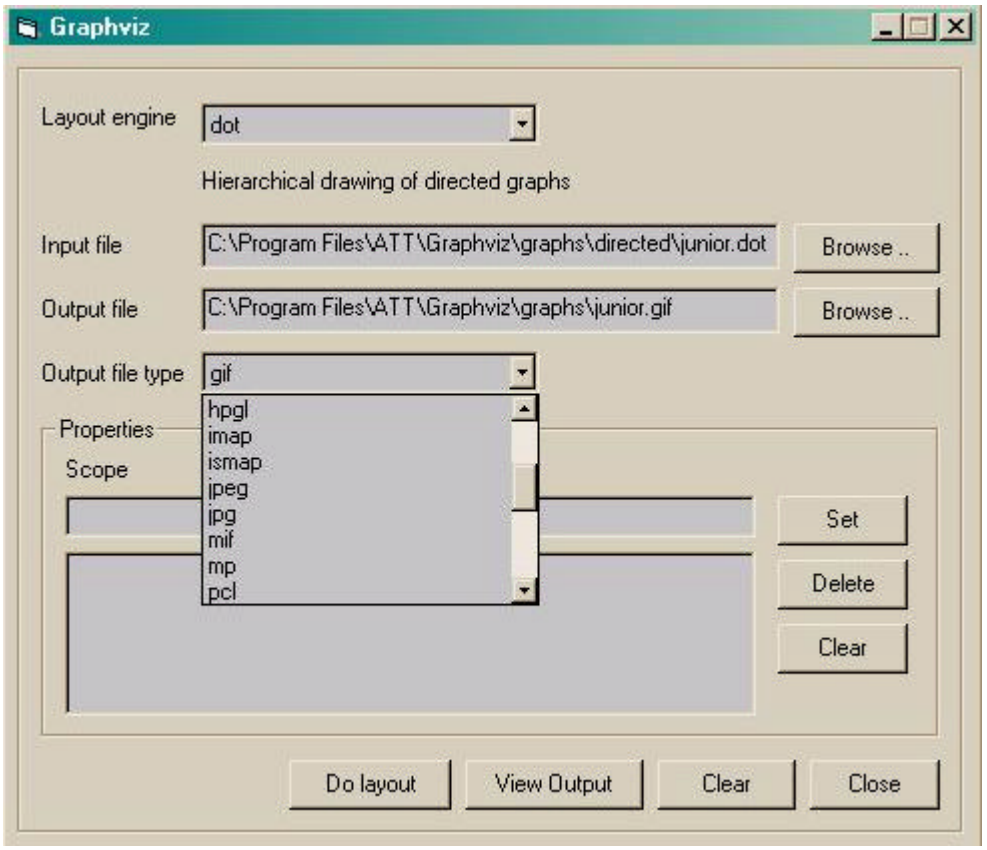


Figura 5-1 Sistema GraphViz

A utilização da ferramenta é simples. O usuário inicialmente escolhe qual construtor (*Layout engine*) será utilizado no desenho do grafo. Em seguida, o usuário deve informar o nome do arquivo de entrada (arquivo textual na linguagem *dot),* o nome do arquivo de saída e, por último, o formato do arquivo de saída (jpg, gif, jpeg etc).

Cada construtor (*dot, neato e towapi*) da ferramenta GraphViz está voltado para um determinado tipo de grafo. Conhecer o funcionamento desses construtores torna-se importante pois, de acordo com o tipo de grafo elaborado na linguagem *dot*, o usuário saberá escolher qual deles melhor se aplica à

representação do seu grafo. Nas próximas subseções cada construtor será analisado juntamente com a linguagem *dot*.

5.1.1. Construtor *dot*

O construtor *dot* (Gansner et al., 2002) é destinado para desenhar grafos hierárquicos e direcionados. Os algoritmos usados no desenho são baseados nos trabalhos de (Warfield, 1977), (Carpano, 1980) e (Sugiyama et al., 1981) que operam com a seguinte estratégia: os primeiros vértices definidos no arquivo fonte (linguagem *dot*) apresentam maior prioridade hierárquica com relação aos demais, ou seja, os primeiros elementos definidos no arquivo serão desenhados acima dos demais. Observe na parte direita da Figura 5-2 (fonte na linguagem *dot*) que o elemento *main* foi o primeiro vértice a ser definido no arquivo e conseqüentemente o primeiro a ser desenhado na figura do grafo, criando assim uma hierarquia no desenho conforme os elementos são definidos no arquivo fonte da linguagem *dot*.

O sinal "->" presente no arquivo fonte (linguagem *dot*), indica o sentido da aresta entre dois vértices. A Figura 5-2 ilustra um grafo hierárquico desenhado pelo construtor *dot*. Ao lado da figura, encontra-se a especificação do grafo na linguagem *dot*.

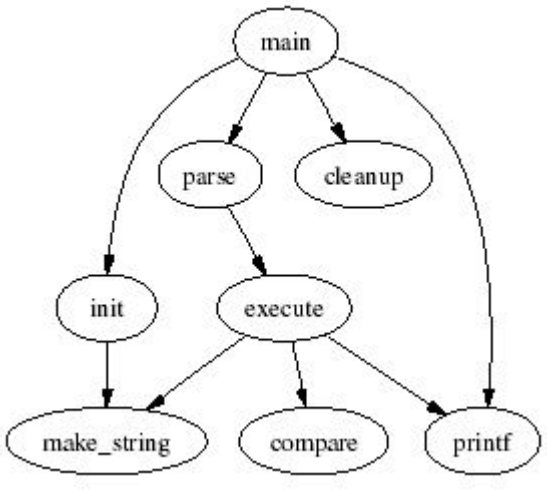


Figura 5-2 Grafo Hierárquico (*dot*)

```
digraph G {
        main -> parse -> execute;
        main -> cleanup;
        main -> init;
        execute -> make_string;
        execute -> printf
        execute -> compare;
        init -> make_string;
        main -> printf;
}
```

Uma deficiência encontrada no construtor *dot*, em alguns casos, é a sobreposição de arestas e vértices quando o grafo desenhado apresenta uma grande quantidade (número superior a 50) de vértices e arestas.

5.1.2. Construtor *neato*

O construtor *neato* é destinado para desenhar grafos não orientados, comuns na representação de redes de computadores, modelos físicos que identificam pontos de baixas energias, sistemas de software etc. (North, 2002).

Os algoritmos utilizados para o desenho dos grafos no construtor *neato* foram propostos por (Kamada & Kawai, 1989). O conceito de elasticidade (*Spring*) é aplicado entre todos os pares de vértices de forma que o tamanho do grafo desenhado seja o menor caminho entre os pontos terminais do mesmo. Maiores detalhes do algoritmo do algoritmo de *Spring* podem ser encontrados em (North, 2002) e (Pinto, 2000).

Com o construtor *neato* é possível configurar tamanhos de arestas em relação a outras arestas, por exemplo, o usuário pode especificar que a aresta entre os vértices *v1* e *v2* tenha o dobro do tamanho da aresta que une os vértices *v1* e *v3*.

Tanto o construtor *neato* como o *dot* oferecem as mesmas possibilidades de configurações de desenho de grafos, tais como: cor do vértice, formato do vértice (quadrado, elipse, triângulo etc.) e fontes dos textos para legenda de vértices e arestas. Além disso, o usuário pode determinar a posição exata de vértices no desenho do grafo com base nas coordenadas cartesianas da figura a ser criada. Nesse caso, o ponto de origem (0,0) do eixo cartesiano está localizado na parte superior esquerda da figura.

A Figura 5-3 apresenta um grafo não direcionado desenhado pelo construtor *neato,* tendo ao seu lado a especificação do grafo na linguagem *dot*. Observe que todas as arestas são representadas pelo símbolo *"--"*, utilizado na linguagem *dot* para representar uma aresta não orientada.

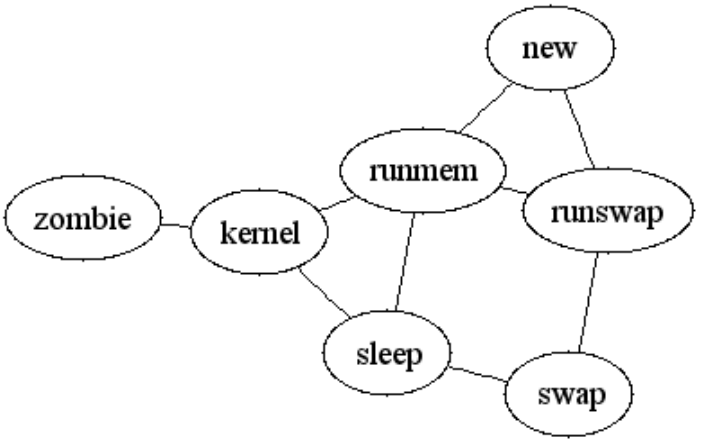


Figura 5-3 Grafo não direcionado (*neato*)

```
graph G {
kernel -- zombie;
kernel -- sleep;
kernel -- runmem;
sleep -- swap;
swap -- runswap;
runswap -- new;
runswap -- runmem;
new -- runmem;
sleep -- runmem;
}
```

Análogo ao construtor *dot*, uma deficiência encontrada no construtor *neato*, em alguns casos, é a sobreposição de nós em arestas quando o grafo desenhado apresenta uma grande quantidade (número superior a 50) de vértices e arestas.

5.1.3. Construtor *twopi*

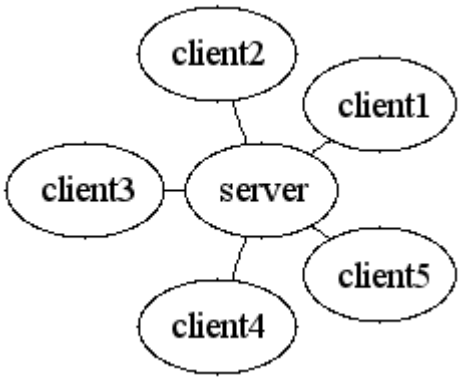
Destinado a desenho de grafos circulares (Wills, 1997), o algoritmo desse construtor procede da seguinte forma: um vértice é escolhido como centro da circunferência e todos os outros vértices são dispostos em volta dele. Caso a quantidade de vértices desenhados em volta do vértice central gere sobreposições de vértices e arestas, outros círculos podem ser criados com raios diferentes até que todos os vértices sejam desenhados.

Na linguagem *dot*, o usuário pode determinar a distância entre cada anel circular, a forma das figuras dos vértices (quadrado, triângulo, imagens etc.), cores de preenchimento, fonte de texto etc.

É importante destacar que a linguagem *dot* disponibiliza recursos para todos os construtores. No entanto, caso alguma informação seja passada no fonte do arquivo (linguagem *dot)* que o construtor não consiga interpretar será enviada uma mensagem de erro para usuário informando a linha na qual o erro foi detectado. Por exemplo, inserir a aresta "->" no fonte do arquivo para representar uma aresta no construtor *twopi*.

O construtor é pré-configurado para trabalhar apenas com um anel, podendo gerar sobreposição de vértices caso o número de vértices a serem desenhados seja grande (número maior que 30).

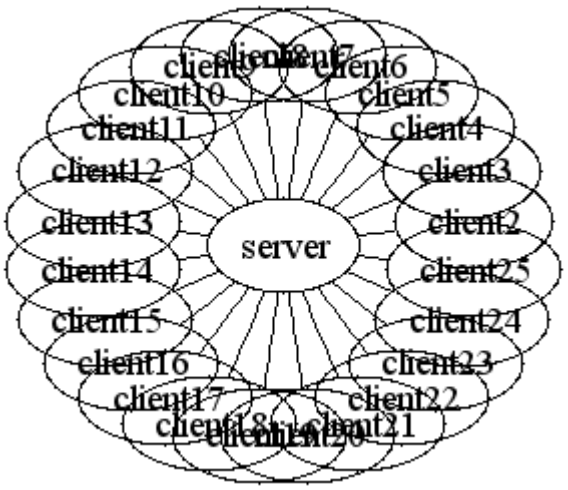
A Figura 5-4 ilustra um exemplo simples de grafo desenhado com o construtor *twopi*.




```
graph G {
    server -- client1;
    server -- client2;
    server -- client3;
    server -- client4;
    server -- client5;
}
```

Figura 5-4 Grafo circular simples (*twopi*)

A Figura 5-5 ilustra um grafo com um número maior de vértices que o da Figura 5-4. Devido à utilização de apenas um anel, houve sobreposição.




```
Graph G {
        server -- client2;
        server -- client3;
        server -- client4;
        server -- client5;
        server -- client6;
        …
        server -- client22;
        server -- client23;
        server -- client24;
        server -- client25;
}
```

Figura 5-5 Grafo circular com sobreposição (*twopi*)

A Figura 5-6 apresenta o mesmo grafo da Figura 5-5, porém sem sobreposição de vértices, através da utilização de dois anéis. Observe, no código, as linhas em negrito "**ranksep = 1.75"** e **"overlap = false"** que foram adicionadas ao código fonte do arquivo em relação ao código da Figura 5-5. A variável *ranksep* determina a relação de distância entre os raios circulares a serem

desenhados e a variável *overlap* com o valor igual a *"false"* impede a sobreposição de vértices. No caso da Figura 5-6 o valor de *ranksep* igual a 1.75 significa que o raio do próximo anel será 75% maior que o inicial.

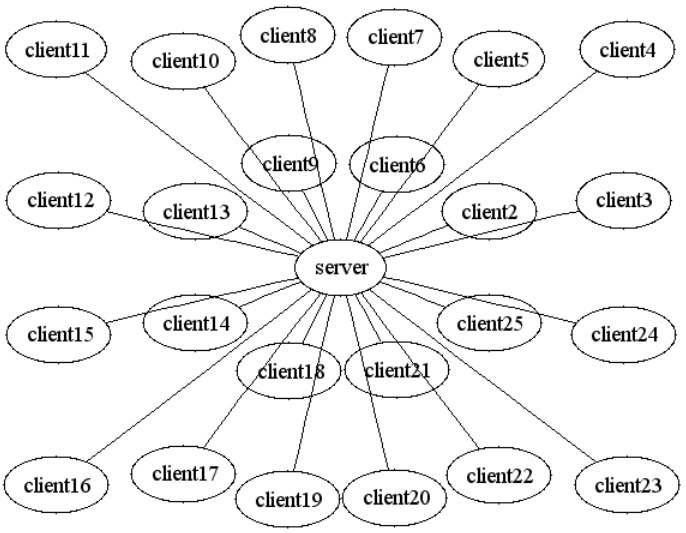


Figura 5-6 Grafo circular sem sobreposição (*twopi*)

```
graph G {
        ranksep = 1.75;
        overlap = false;
        server -- client2;
        server -- client3;
        server -- client4;
        …
        server -- client22;
        server -- client23;
        server -- client24;
        server -- client25;
}
```

5.1.4. Comparações entre o GraphViz e HyperProp

A visão estrutural atual do sistema HyperProp (Soares et al., 2003) apresenta apenas o algoritmo *Spring* (Kamada & Kawai, 1989) disponível para representação de grafos. Já a ferramenta GraphViz, além do algoritmo de *Spring*, disponibiliza ao usuário outros dois algoritmos, *Hierárquico* (Sugiyama et al., 1981) e *Circular* (Wills, 1997), para representação de grafos.

Na nova versão da visão estrutural do HyperProp (Seção 3.2.1), que está em desenvolvimento, o usuário pode escolher o algoritmo aplicado no desenho da visão estrutural. Atualmente, já estão implementadas as opções *Spring* (Kamada & Kawai, 1989), *Hierárquico* (Sugiyama et al., 1981) e *Circular* (Wills, 1997).

Uma deficiência encontrada na ferramenta GraphViz é não apresentar uma interface gráfica que permita uma interação com o usuário, para construção de grafos passo a passo. O uso da ferramenta limita-se a desenhar grafos descritos na linguagem *dot*. Já o HyperProp possibilita ao usuário o uso de interfaces gráficas na criação de vértices e arestas e técnicas de filtragens para visualização do grafo.

O problema da sobreposição de vértices e arestas encontradas na ferramenta GraphViz é melhor solucionado no HyperProp, embora também ocorra. No

HyperProp esse problema é parcialmente contornado, pois o desenho do grafo é mais flexível no que tange ao dimensionamento da figura. No entanto, para grafos grandes, o dimensionamento pode, às vezes, prejudicar o entendimento do desenho por parte do usuário. As soluções empregadas para resolver esse problema foram o uso de *scroll* e as técnicas de filtragem (olho-de-peixe explicada no Capítulo 4).

5.2. SALIX

SALIX (Aoki & Seraphim, 2003) é um ambiente para autoria de documentos XML (XML, 2000), desenvolvido na Universidade Estadual Paulista, que tem como característica principal o uso de uma interface gráfica (WYSIWYG - *What You See Is What You Get*) com recursos que agilizam a produção de documentos XML.

A ferramenta SALIX trabalha com gerenciamento de janelas MDI (*Multiple Document Interface*), permitindo que vários documentos sejam abertos ao mesmo tempo. As seguintes janelas estão disponíveis no editor:

- XML (texto) – para a edição do documento XML em modo texto.
- DTD (texto) – para edição textual da DTD - *Document Type Declaration* referenciada pelo fonte XML;
- XML (gráfico) – para edição do documento XML em modo gráfico.

Assim, duas formas de visualização de documentos XML são apresentadas. A primeira é puramente textual, como em um editor de texto convencional. Porém, por ser um editor de uma linguagem de marcação, as marcas (*tags*) dos elementos são diferenciadas (apresentadas em cores diferentes) do conteúdo do texto propriamente dito. Esse recurso é usado nas guias DTD e XML (texto), conforme ilustrado na Figura 5-7.

A segunda forma de visualizar o documento XML é através da interface gráfica criada com uma tabela de valores para cada elemento XML presente no documento e uma visão em árvore do documento XML. A Figura 5-8 apresenta o editor na forma gráfica.

O quadro de estrutura hierárquica, lado esquerdo do editor, apresenta os nomes dos elementos, nome de atributos de cada elemento do documento XML,

formando assim, a árvore do documento. Nesse quadro, o usuário pode adicionar e excluir elementos com auxílio do mouse, além de expandir e fechar uma folha na árvore.

O quadro grade de valores, lado direito do editor, apresenta o nome do elemento e o valor de cada atributo do elemento selecionado pelo usuário na árvore esquerda do editor. Nessa visão, o usuário pode alterar o conteúdo dos atributos, assim como o nome do elemento.

É importante destacar que a visão gráfica só está disponível para o usuário se o arquivo estiver bem formado, ou seja, respeitando as regras de formação definidas pela DTD.

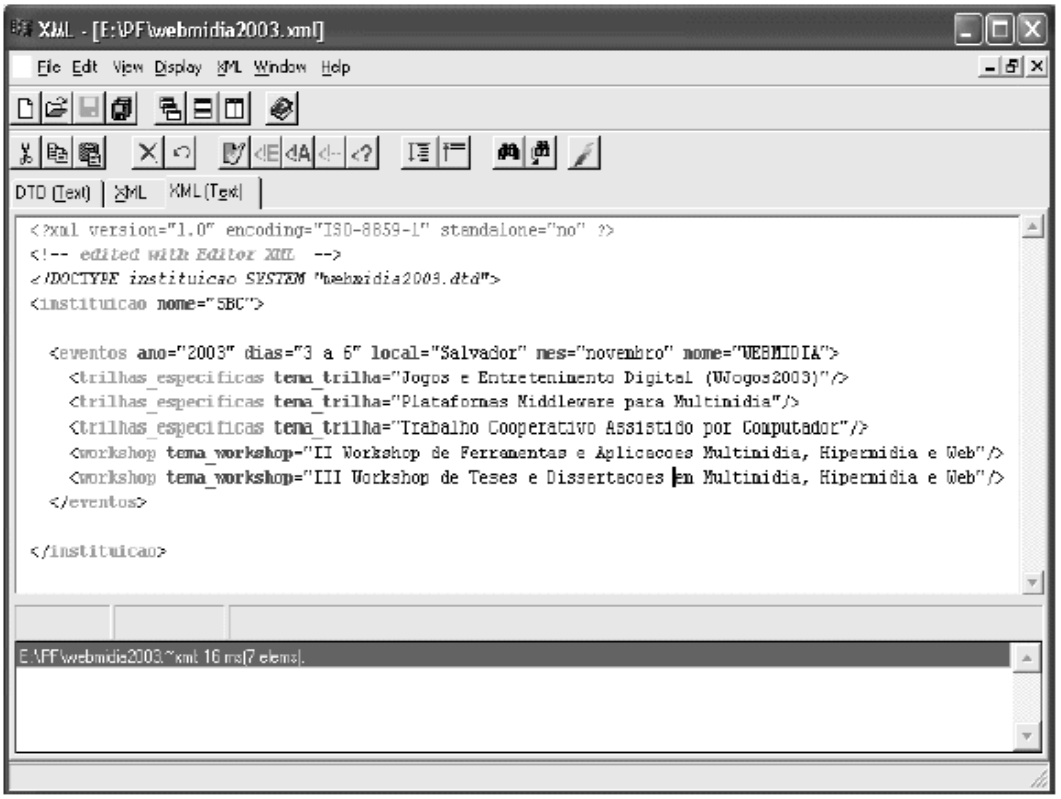


Figura 5-7 Ambiente SALIX (Autoria Declarativa)

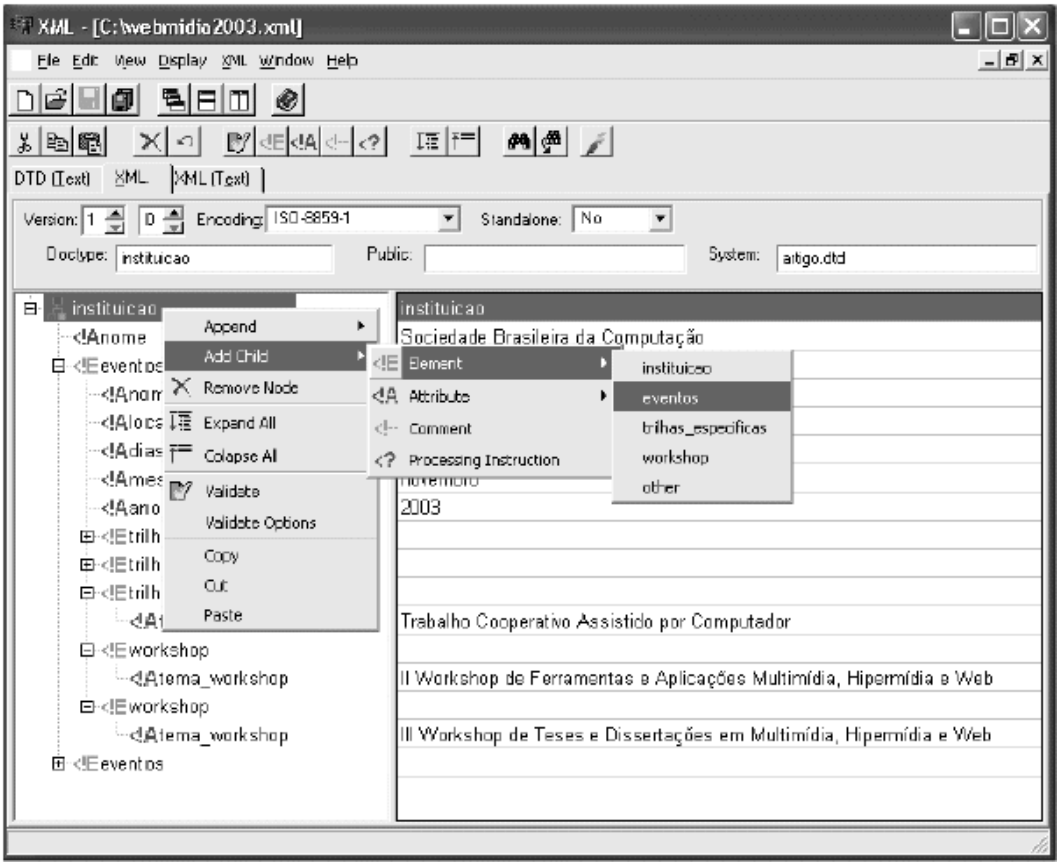


Figura 5-8 Ambiente SALIX (Autoria em árvore e tabelas)

O editor SALIX apresenta as seguintes funcionalidades, além da própria edição de documentos:

1. *Validação do documento* – Permite validar o documento XML editado pelo usuário caso exista alguma DTD associada ao arquivo;
2. *Depurador* – Permite ao usuário localizar os erros (caso existam) no documento editado depois da validação.

5.2.1. Comparações entre o SALIX e HyperProp

A visão declarativa do sistema HyperProp e o sistema SALIX são muito semelhantes. As funcionalidades de validação e depuração presentes no ambiente SALIX também estão disponíveis na visão declarativa do HyperProp, assim como a presença do *highlight* nos nomes e atributos de cada elemento do documento na forma textual.

As principais diferenças entre os editores são:

1. Visão em árvore do documento XML. A visão em árvore do editor declarativo do sistema HyperProp (parte esquerda do editor, Figura

3-1) informa apenas o nome dos elementos XML do documento. Já na árvore do editor SALIX, para cada elemento XML presente no documento, uma nova sub-árvore é criada, tendo como folhas todos os atributos do elemento. Note que, em alguns casos, o excesso de folhas criadas pode acabar dificultando o usuário na busca por uma determinada informação no documento.

2. Destaque para elementos selecionados na árvore do documento. No sistema SALIX, quando o usuário seleciona um determinado elemento na árvore XML do documento, a parte direita do editor é automaticamente atualizada mostrando apenas as informações do elemento selecionado. Já na visão declarativa do sistema HyperProp, quando o usuário seleciona um determinado elemento da árvore XML, a linha textual referente ao elemento selecionado é destacada em azul e o cursor do editor posicionado no início da linha.
3. Técnicas de Filtragens. O sistema SALIX não apresenta nenhuma técnica de filtragem associada ao editor. A visão declarativa do HyperProp apresenta a técnica de filtragem olho-de-peixe conforme apresentado no Capítulo 4.

5.3. Kaomi

O sistema Kaomi (Jourdan et al., 1999) tem, como base do processo de autoria, um conjunto de visões do documento, onde a sincronização entre as visões é realizada através da seleção de objetos ou por intervalos de tempo predefinidos, cabendo ao usuário do sistema configurar o valor desses intervalos.

É interessante destacar que o Kaomi é flexível pois trabalha com uma grande variedade de formatos declarativos de documentos multimídia, tais como documentos SMIL, XML, formato declarativo do modelo Madeus e etc.

O Kaomi fornece diferentes visões para trabalhar sobre o documento multimídia proporcionando ao usuário facilidades na navegação e edição de documentos multimídia baseada no paradigma WYSIWYG (*What You See Is What You Get*). As visões oferecidas pela ferramenta são:

1. *Visão de Apresentação* – oferece funções de controle básicas (tocar, parar, pausar) para determinados tipos de mídia.
2. *Visão Estrutural* – apresenta o conjunto de objetos que pertencem ao documento. A estrutura formada é uma árvore onde as folhas representam os objetos básicos de mídia e os nós não terminais são os objetos compostos. É interessante destacar que, nesse tipo de visão, é possível aplicar filtros de exibição da estrutura, tais como: ordem alfabética e tipo de mídia a ser exibida.
3. *Visão Declarativa* – exibe o documento fonte na forma textual em um editor semelhante ao programa *Notepad*, com recursos básicos de edição;
4. *Visão Temporal* – projeta, em um grafo, a execução de um documento no tempo, preservando a noção de sua estruturação hierárquica. Nessa visão, as arestas representam elos e, os vértices, os objetos de mídias.

Quando ocorre alguma alteração durante a edição no documento em uma determinada visão, essa alteração é propagada para todas as outras visões, garantindo com isso a consistência dos dados.

O sistema Kaomi também disponibiliza as funções elementares para a autoria de documentos tais como: abrir, fechar, criar e salvar. Além dessas, o Kaomi apresenta funções temporais (por exemplo, exibir em série objetos de mídia ou apresentar os objetos em paralelo, inserir atraso entre os objetos etc.) e funções de relação hipertexto, por exemplo, operações do tipo *goto* ou seja, caso algum objeto de mídia seja selecionado pelo usuário durante a apresentação do documento um outro objeto de mídia pode ser acionado.

O ambiente de autoria produzido pelo Kaomi pode ser ajustado para se adequar ao contexto da aplicação. A ferramenta oferece meios de ser expandida, permitindo que novas visões sejam adicionadas, mantendo as características de edição e sincronismo. É possível acrescentar ainda novas funções em uma visão, assim como definir novas relações temporais e ampliar o suporte a novos tipos de formato de dados.

5.3.1. Comparações entre o Kaomi e HyperProp

O sistema HyperProp apresenta observadores que atuam sobre as visões (Seção 3.4), responsáveis pelo controle da sincronização entre elas. Já o sistema Kaomi possibilita ao usuário determinar intervalos de tempo para a sincronização, além dos eventos de sincronização pré-configurados no ambiente, tais como: seleção de objetos, exclusão de elementos, inserção de elementos etc.

O Kaomi oferece a possibilidade de trabalhar com novos formatos de arquivos diferentes daquele para o qual a ferramenta vem pré-configurada (SMIL (SMIL, 2001)). Para isso, o usuário deve criar compiladores para converter seus documentos na base de dados utilizada pelo sistema (Jourdan et al., 1999). O HyperProp também pode ser usado por diferentes formatos de arquivos, desde de que o usuário crie compiladores que convertam o arquivo do usuário para a base de dados (objetos Java representando um grafo) utilizada pelo sistema HyperProp.

A visão declarativa do Kaomi é muito simples, apresentando apenas os elementos na forma puramente textual sem recursos de destaque (*highlight*), ausência da visão em árvore do documento interna ao editor etc. A visão declarativa do sistema HyperProp disponibiliza validação do documento baseado no *Schema* do arquivo XML, técnicas de filtragem, recursos de *highlight*, visão em árvore do documento interna ao editor declarativo etc.

A visão estrutural do Kaomi limita-se a apresentar apenas a estrutura do documento na forma de árvore. Além disso, a visão estrutural do Kaomi não disponibiliza nenhuma técnica de filtragem nas visões e nada é mencionado com relação ao algoritmos de desenho grafos. Já a visão estrutural do HyperProp mostra a árvore do documento na parte esquerda do editor (Figura 3-3), o desenho do grafo com todos os relacionamentos entre os vértices na parte direita da visão (Figura 3-3) e um conjunto de funcionalidades para edição de documentos já mencionados anteriormente na Subseção 3.2.1.

5.4. GRiNS

GRiNS é um ambiente para autoria de documentos hipermídia com suporte nativo à linguagem SMIL (SMIL, 2001), padrão W3C, que tem como

característica principal o uso da visão temporal no processo de autoria com recursos gráficos que facilitam a complexa tarefa de organização temporal dos objetos de mídia no documento.

As apresentações (documentos SMIL) criadas no ambiente GRiNS podem ser visualizadas no próprio ambiente e em diferentes sistemas (*players*) que são compatíveis com a linguagem SMIL, tais como: RealOne, IE-6 e 3GPP.

O sistema trabalha com arquivos de áudio (mp3, wav e aiff etc.), vídeo (avi, mpeg etc.), imagem (bmp, jpg, gif, png etc.) e texto (HTML, XHTML etc.) de forma integrada dentro da mesma apresentação hipermídia.

A Figura 5-9 apresenta a tela inicial do ambiente GRiNS com suas barras de ferramentas (*general toolbar*, *container toolbar*, *region/alignment toolbar*, *linking/interation toolbar e previewer control panel*) e barra de menu (*menu bar*). As funcionalidades das principais barras serão explicadas nas próximas subseções, juntamente com as cinco visões disponíveis no ambiente.

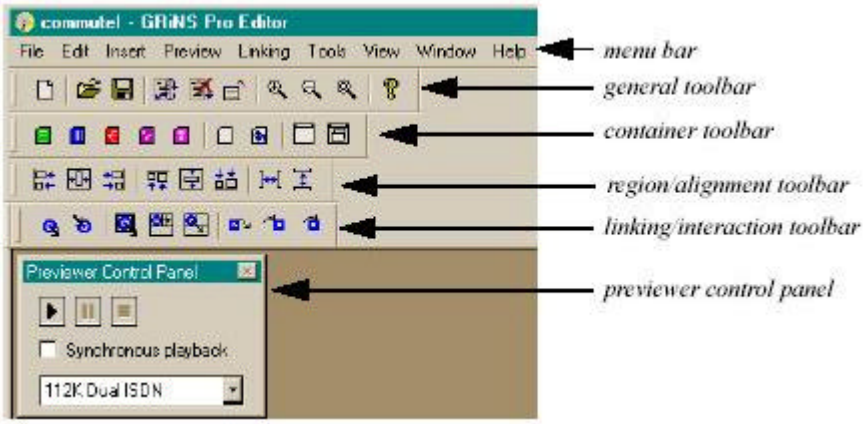


Figura 5-9 Janela principal do ambiente GRiNS

5.4.1. Visão de Apresentação

Permite ao autor verificar a apresentação do seu documento hipermídia antes do mesmo ser disponibilizado para diferentes sistemas (RealOne, IE-6 ou 3GPP).

A apresentação hipermídia pode ser simulada de acordo com a largura de banda selecionada pelo autor. Conforme o usuário altera o valor da largura de banda, os tempos de pré-busca (*prefetching*) dos objetos de mídia são recalculados na visão temporal e refletidos durante a apresentação do documento. A Figura 5-10 ilustra a visão de apresentação do sistema GRiNS.

Na visão de apresentação (“*Previewer Control Panel”*) é possível que o autor escolha partes do documento a serem apresentadas, por exemplo, o ponto de início e fim do documento.

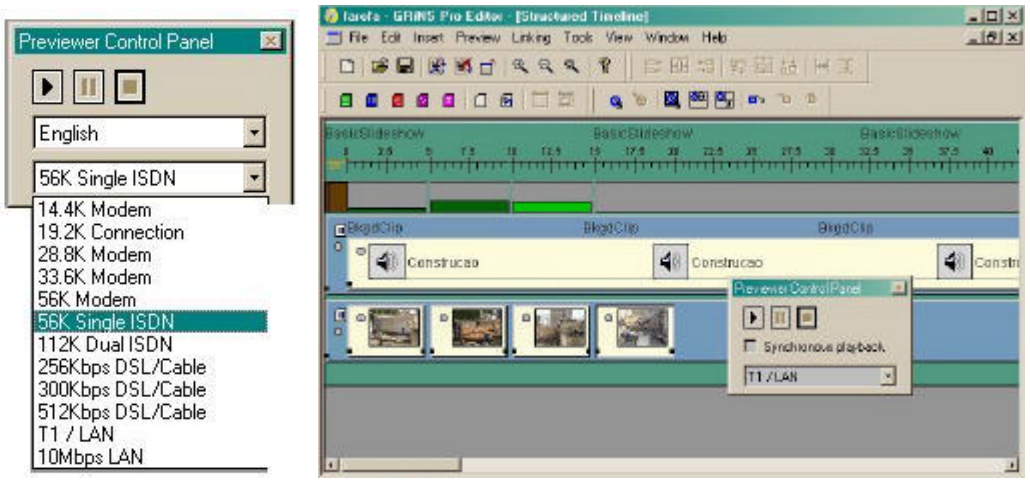


Figura 5-10 Visão de apresentação do sistema GRiNS

5.4.2. Visão Temporal

A visão temporal do sistema GRiNS permite ao autor criar documentos hipermídia baseados nos elementos de semântica temporal definidos na linguagem SMIL-2.0.

Os principais elementos temporais presentes no ambiente GRiNS são: paralelo, seqüencial e escolha (*switch*). Cada elemento temporal é representado no sistema como um contêiner de forma que todos os elementos internos a esse contêiner estão sujeitos à sua semântica temporal.

A barra de menu *“Container toolbar”* apresenta um conjunto de botões coloridos, onde cada botão está associado a um tipo diferente de contêiner. O botão verde cria um contêiner com semântica temporal paralelo. O botão azul cria um *contêiner* com semântica temporal seqüencial e o botão vermelho cria um *contêiner* com semântica temporal escolha (dentre todos os elementos internos a ele apenas um será escolhido durante a apresentação do documento).

A Figura 5-11 mostra a visão temporal com a apresentação *“Sample Slideshow”*, *contêiner* azul mais externo. Dentro desse *contêiner* existe um único *contêiner* verde (*“Audio-and-Image”*) com outros três *contêiners:* o primeiro, de cor azul (*“Image-Sequence”*), contendo três objetos de mídia imagem. O segundo, apenas com um nó de texto e, o terceiro *contêiner*, de cor vermelha (*“Audio-*

Switch"), com dois nós de áudio. Observe que os elementos estão organizados dentro dos *contêiners* de acordo com sua semântica temporal.

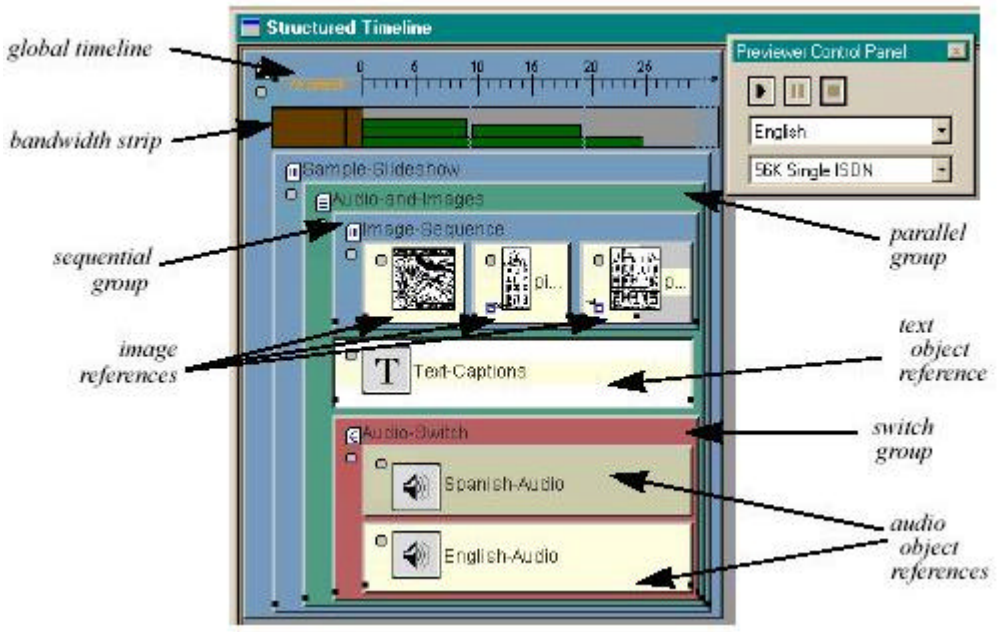


Figura 5-11 Visão Temporal do GRiNS

Na visão temporal o autor sabe quais são os tipos de objetos de mídia envolvidos na sua apresentação, os tipos de estruturas criadas na composição da apresentação e uma estimativa do tempo da apresentação do documento. Além disso, a visão temporal fornece recurso de integração com a plataforma *Windows*, onde arquivos selecionados no sistema operacional *Windows* são inseridos no ambiente através de *drag-drop* (arraste e copie).

5.4.3. Visão Espacial

Na visão espacial, o autor pode criar e editar regiões de apresentações nas quais os objetos de mídias serão exibidos durante a execução do documento.

A Figura 5-12 ilustra a visão espacial do GRiNS. Na área esquerda do editor, as regiões criadas no documento são exibidas na forma de árvore e todos os objetos de mídias que estão associados a essa região aparecem como suas folhas. Na janela da direita, é ilustrado o objeto de mídia selecionado pelo autor na árvore (parte esquerda do editor) dentro da região que o objeto de mídia está associado.

Na visão espacial, o autor pode alterar os parâmetros (altura, largura etc.) da região com auxílio do *mouse*, assim como inserir novos objetos de mídia.

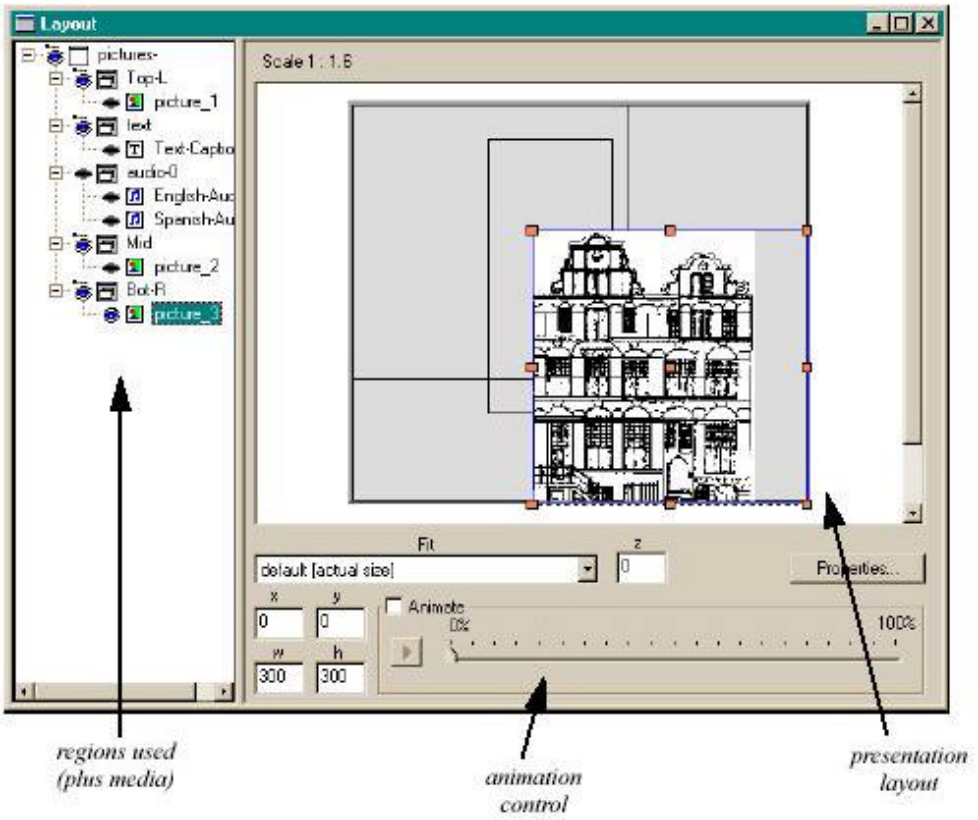


Figura 5-12 Visão Espacial do sistema GRiNS

5.4.4. Visão Declarativa

A visão declarativa do sistema GRiNS apresenta o fonte do documento hipermídia na linguagem SMIL-2.0, permitindo ao autor editar o documento na forma textual. As alterações feitas na forma textual são atualizadas nas outras visões (temporal e espacial) desde que o documento não contenha erros.

Para atualizar as alterações realizadas na visão declarativa com as demais visões gráficas, o usuário precisar selecionar o botão *"apply"* no editor declarativo.

A Figura 5-13 exibe a visão declarativa do ambiente GRiNS. Note que o editor é simples, sem grandes recursos para edição textual.

Os únicos elementos que recebem destaque (*"highlight"*) no editor textual são aqueles de semântica temporal (*par*, *seq* e *switch*).

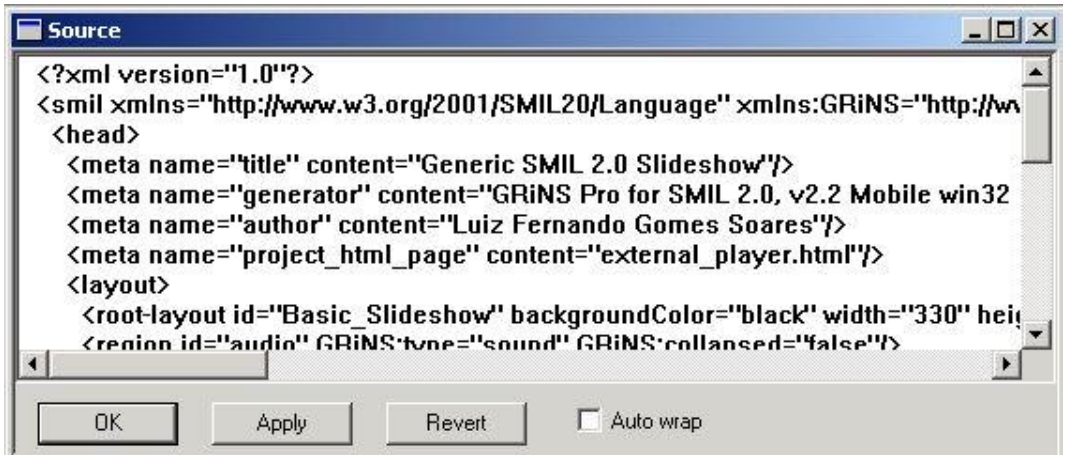


Figura 5-13 Visão Textual do GRiNS

5.4.5. Comparações entre o Sistema GRiNS e o HyperProp

O ambiente GRiNS não apresenta a visão estrutural para autoria de documentos hipermídia. A principal visão de autoria do sistema GRiNS é a visão temporal. Já o sistema HyperProp além de disponibilizar a visão temporal oferece também a visão estrutural para edição de documentos.

A visão espacial do GRiNS apresenta algumas funcionalidades interessantes com relação à visão espacial atualmente desenvolvida no HyperProp, tais como: pré-visualização do objeto de mídia na região (possibilitando ao autor alterar os parâmetros da região de acordo com a visualização do objeto de mídia na região) e visão em árvore dos elementos que compõem o *layout* da apresentação. Cada região espacial definida no documento forma um ramo da árvore tendo como folhas todos os objetos de mídia associados a essa região. Esse tipo de visão em árvore ainda não está implementada na visão espacial atual do HyperProp.

A visão declarativa do ambiente GRiNS apresenta sempre todo o documento na forma textual, tornando o documento de difícil entendimento quando o mesmo torna-se relativamente grande. A visão declarativa do HyperProp exibe apenas a parte textual do documento referente à visão gráfica que o usuário tem como foco. Por exemplo, quando o usuário ativa a visão gráfica espacial, apenas o texto referente à edição espacial é mostrado na visão declarativa.

Uma limitação do sistema GRiNS é o fato do mesmo ser executado apenas na plataforma *Windows*. Já o Hyperprop é multiplataforma por ser desenvolvido através da linguagem Java.

A opção de "*highlight*" no editor textual do GRiNS é simples, identificando apenas os nomes de elementos de semântica temporal (*par*, *seq* e *switch*). Já no editor declarativo do HyperProp, todos os nomes de elementos apresentam-se no formato negrito, o nome e o valor dos atributos são exibidos também em destaque. Por fim, o editor textual do GRiNS não exibe a visão em árvore do documento criado e nenhuma técnica de filtragem no editor é empregada.

5.5. Outras Ferramentas de Edição

As ferramentas apresentadas nas próximas subseções possuem funcionalidades que atualmente não estão disponíveis no sistema HyperProp. Entretanto, algumas delas, tal como a utilização de trilhas na navegação de documentos, já foram anteriormente implementadas no sistema (Muchaluat-Saade & Soares, 1995) (Muchaluat-Saade, 1996) e serão apresentadas como referências a outros trabalhos que permitem edição de documentos.

5.5.1. Ariadne

Ariadne é uma ferramenta que auxilia usuários da *WWW* (Berners-Lee et al., 1994a) a administrarem os caminhos (trilhas) percorridos durante sua navegação de páginas na *Web* (Jühne et al., 1998).

Com o sistema Ariadne, o usuário pode criar, editar e compartilhar trilhas sobre determinados assuntos na *Web*, assim como visualizar a trilha num formato gráfico.

Ao usar uma rota já criada no sistema Ariadne, o usuário passa a ter um mapa gráfico com os nós já percorridos, o ponto atual no qual ele se encontra e os nós ainda não percorridos pelo usuário.

É interessante destacar que, a qualquer momento, o usuário pode sair de uma rota pré-determinada e até mesmo adicionar novos nós na rota utilizada inicialmente. A Figura 5-14 ilustra o uso do mapa gráfico do sistema Ariadne.

Uma outra característica interessante do Ariadne é que ele não altera o conteúdo das páginas *WWW*, diferenciando-se das outras ferramentas com mesma funcionalidade, por exemplo: Walden's Path (Furuta et al., 1997).

O Ariadne torna-se muito útil quando o compartilhamento de rotas ocorre entre usuários. Por exemplo, ao se pesquisar um tema específico na *Web*, o usuário pode perder muito tempo filtrando informações relevantes à sua pesquisa, devido à grande quantidade de informações disponíveis na *Web* sobre o mesmo tema. Ao obter uma rota já criada, o usuário passa a ter um guia de navegação sobre seu tema, reduzindo com isso seu tempo de pesquisa.

As principais características do Ariadne são:

1. *Uso de interface independente* – O Ariadne mantém a apresentação das páginas *WWW* em seu formato original, ou seja, os conteúdos das páginas não são alterados;
2. *Ramificação de navegação* – Permite que o usuário navegue por rotas alternativas dentro de um guia de navegação;
3. *Composição de navegação* – Permite que uma navegação seja adicionada a outro guia de navegação;
4. *Orientação na navegação* – Provê uma representação gráfica da navegação do usuário e de seu histórico.

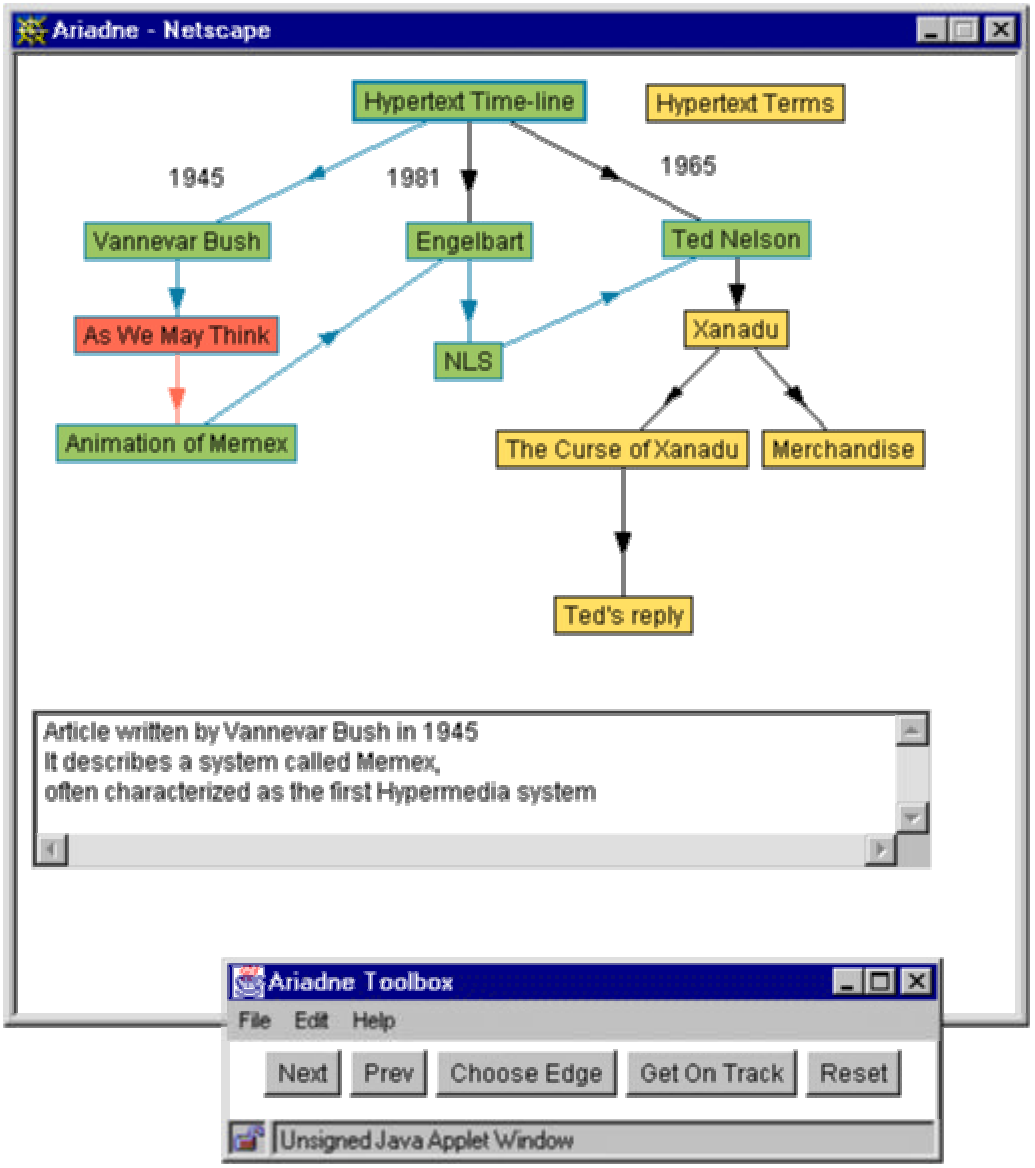


Figura 5-14 Mapa de navegação da ferramenta Ariadne

5.5.2. Arakne

No modelo atual da *WWW* é impossível criar relacionamentos (elos) ou anotações nas páginas da *Web* onde não se tem o direito de escrita (que é o que ocorre normalmente). Isso dificulta os usuários a organizarem as informações encontradas, haja vista que seria interessante adicionar comentários e novos elos em algumas páginas encontradas.

Uma alternativa para contornar esse problema foi através da criação de ferramentas que possibilitam o usuário modificar o conteúdo das páginas *WWW*

localmente (na própria máquina). Assim, cada usuário pode "moldar" as páginas *WWW* pesquisadas da forma que melhor lhe convier.

Essas ferramentas permitem, entre outras facilidades, criar elos e adicionar comentários e conteúdo nas páginas. Geralmente, a ferramenta é instalada na máquina do cliente, mas pode também estar associada a um *proxy*, de maneira que as alterações feitas por um usuário possam ser vistas por todos os usuários que compartilham o mesmo *proxy*. Assim, a ferramenta facilita o trabalho colaborativo na busca por determinadas informações.

Uma dessas ferramentas é o Arakne (Bouvin, 2000), elaborada na Universidade de Aarhus. Inicialmente, a ferramenta foi desenvolvida como *applets*, possibilitando que usuários criem comentários e elos nas páginas. Logo depois a ferramenta virou um aplicativo desenvolvido em Java com suporte a outros tipos de mídias como vídeo e áudio.

O usuário, ao assistir um vídeo através do sistema Arakne, pode adicionar elos em determinados intervalos de vídeo. A Figura 5-15 exibe um vídeo associado à ferramenta. Observe que o usuário criou o *"link 5"* com dois pontos de ativação (tempo de início e fim do elo). Caso o usuário (ou até mesmo outro usuário que assista o filme através da ferramenta Arakne) clique no vídeo durante o intervalo de tempo predeterminado pelo *"link 5"*, o elo será ativado, realizando um determinado evento, por exemplo, abrindo uma página HTML (HTML, 1999) com algumas informações sobre o ator principal da œna exibida nesse intervalo de tempo.

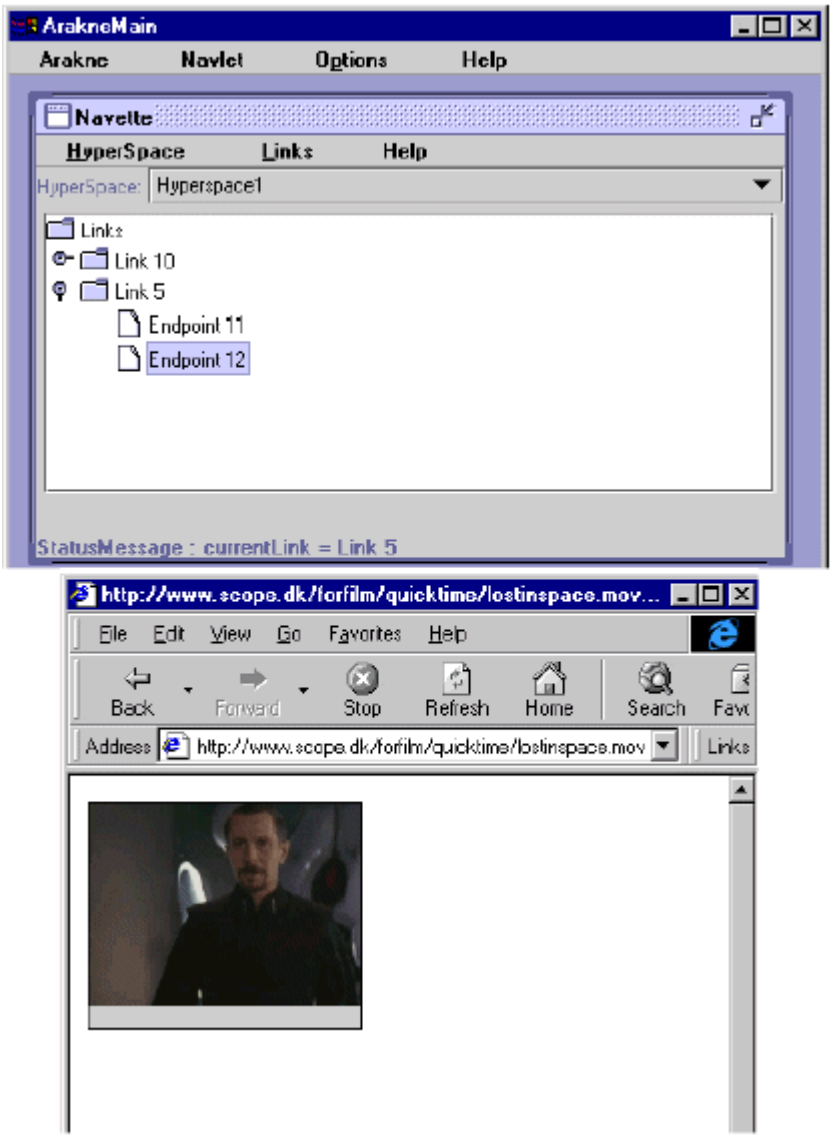


Figura 5-15 Ambiente Arakne

6
Conclusões

O objetivo principal deste trabalho foi desenvolver uma ferramenta para a especificação de arquiteturas de sistemas baseadas em grafos compostos, em especial grafos com composicionalidade. A ferramenta desenvolvida pode ser aplicada diretamente no domínio de autoria de documentos hipermídia, mas pode também ser utilizada em outros domínios, como na definição de arquiteturas de sistemas de software, ferramentas de especificação formal, projetos de *Workflows*, entre outros.

No Capítulo 2, a definição de grafos compostos e o conceito de composicionalidade foi apresentado. Em seguida, algumas arquiteturas de sistemas baseadas em grafos compostos foram analisadas, destacando-se os elementos de cada arquitetura que são passíveis de serem modelados em elementos de grafos compostos.

As visões espacial, estrutural e declarativa disponíveis no ambiente de autoria do sistema apresentado nesta dissertação trabalham de forma sincronizada, de maneira que as alterações realizadas em uma das visões sejam refletidas nas outras visões. No Capítulo 3, as visões do sistema desenvolvido foram analisadas destacando as funcionalidades disponíveis em cada visão. Em seguida, o funcionamento do mecanismo de sincronização entre as visões foi detalhado. Por fim, um estudo sobre os compiladores no sistema foi discutido levantando as questões que devem ser analisadas no desenvolvimento de novos compiladores para diferentes arquiteturas.

A principal técnica de filtragem aplicada sobre as visões estrutural, declarativa e na especificação de leiaute da visão espacial foi a estratégia olho-de-peixe. A técnica foi estendida para trabalhar com grafos compostos de modo que, em uma única visão, os detalhes do vértice focado pelo usuário sempre sejam mostrados sem perder-se a noção da estrutura global do grafo. O Capítulo 4 analisou a técnica olho-de-peixe na ferramenta com exemplos de uso da técnica sobre as visões, assim como os cálculos realizados em cada uma das visões.

As comparações entre o sistema HyperProp (na versão atual) com diferentes ferramentas de autoria destacando os pontos positivos e negativos de cada ferramenta foi abordado no Capítulo 5.

6.1. Trabalhos Futuros

Alguns pontos dos trabalhos realizados nesta dissertação ainda podem ser explorados como trabalhos futuros.

A utilização da ferramenta desenvolvida pode ser estendida através do desenvolvimento de novos compiladores para diferentes arquiteturas. Atualmente, apenas o compilador para documentos NCL está integrado ao sistema. Com novos compiladores implementados, o sistema ficará mais portável, pois diferentes arquiteturas poderão ser importadas e editadas no sistema.

Muitas ferramentas para autoria de grafos exportam seus documentos no formato GXL - *Graph eXchange Language* (Winter et al., 2002) (Apêndice A). É interessante o desenvolvimento de um compilador que possibilite aos usuários do sistema HyperProp exportar e importar documentos no formato GXL. Com esse compilador integrado ao sistema, a interoperabilidade na edição em diferentes sistemas de edição de grafos torna-se possível.

A versão atual da visão estrutural do sistema apresenta alguns problemas com relação à exibição de vértices. Por exemplo, o tamanho do quadrado desenhado para uma composição expandida fica demasiadamente grande em alguns casos, causando transtornos para usuário que acaba tendo que redimensionar o quadrado manualmente. Além disso, a visão estrutural disponibiliza apenas o algoritmo de *Spring* (Kamada & Kawai, 1989) para desenho de grafos.

Os algoritmos de *Spring*, Hierárquico (Sugiyama et al., 1981) e Circular (Wills, 1997) já estão implementados na nova versão da visão estrutural. No entanto, alguns pontos permanecem pendentes e ficam como trabalhos futuros, tais como: ausência da técnica olho-de-peixe, a possibilidade de edição de vértices e arestas e expandir e colapsar vértices compostos.

Com relação à técnica de filtragem olho-de-peixe, um trabalho futuro é a sua extensão de forma a possibilitar a definição de dois vértices como foco na

visão estrutural, facilitando assim a criação de relacionamentos. Nesse caso, uma alternativa seria a definição de uma função grau de interesse conjunta, composta pela soma algébrica de duas parcelas, onde cada uma é a aplicação da função grau de interesse considerando-se cada um dos focos separadamente.

A visão espacial para edição de leiaute pode ser aperfeiçoada acrescentando-se uma visão em árvore interna ao editor, semelhante à visão espacial do sistema GRiNS (Subseção 5.4.3). Com a árvore integrada à visão espacial, o usuário tem uma visão de hierarquia dos vértices de recursos do grafo. Muitas vezes, essa hierarquia não fica explicitada apenas através da janela em que o desenho do grafo é representado devido à sobreposição de vértices atômicos em vértices compostos.

No caso específico de documentos hipermídia, a implementação da árvore no editor espacial apresentado nesta dissertação deve mostrar para cada região de apresentação do documento o conjunto de folhas diretamente ligadas a ela, sendo cada folha um nó de mídia pertencente à região. Assim, o usuário teria uma visão imediata da relação entre nós de mídias e regiões de apresentação do documento.

Ainda com relação à visão espacial, outro trabalho futuro é a possibilidade de edição de relacionamentos espaço-temporais, pois a versão atual não disponibiliza tal funcionalidade.

A nova versão da visão temporal apresenta algumas pendências, tais como: ausência de representação de vértices compostos (apenas os vértices atômicos estão sendo representados na visão), inexistência de arestas relacionando os vértices do grafo, implementação dos formulários para edição de vértices e arestas e, por fim, a técnica de filtragem olho-de-peixe não se encontra implementada na visão.

Atualmente, o sistema apresentado nesta dissertação trabalha apenas com um documento de arquitetura por vez. Seria interessante a possibilidade de se trabalhar com diferentes documentos ao mesmo tempo no ambiente. Assim, o sistema disponibilizaria ao usuário um ambiente de comparações para diferentes arquiteturas através das visões oferecidas pelo sistema.

7
Referências Bibliográficas

ALDER, G. "**Design and Implementation of the JGraph Swing Component**", Technical Report, February 2003. Disponível em http://www.jgraph.com/documentation.html

ALLEN, R. J. "**A Formal Approach to Software Architecture**". Tese de Doutorado, Carnegie Mellon University, EUA, 1997.

AOKI, E. H.; NAKSONE T. L.; SERAPHIM E. Um ambiente de autoria de documentos XML. IX **Simpósio Brasileiro de Sistemas Multimídia e Hipermídia – SBMídia03**, Salvador, Brasil.

ASLST W. M. P. e KUMAR A. **XML Based Schema Definition for Support of Inter-organizational Workflow, Universidade do Colorado,** Relatório Técnico, 2000. Disponível em: http://tmitwww.tm.tue.nl/staff/wvdaalst/workflow/xrl/isr01-5.pdf

BERNERS-LEE, T. J. **The World-Wide Web**. Communications of the ACM, v. 37, n. 8, Agosto de 1994, p. 76-82.

BOUVIN N. O. Augmenting the Web through Open Hypermedia. 150 pages. Ph.D. Thesis, 2000.

BULTEMAN D.; RUTLEDGE L. **"SMIL 2.0: Interactive Multimedia for Web and Mobile Devices"**, Springer, 2004.

BULTERMAN D.C.A.; HARDMEN L.; JASEN J.; MULLENDER K. S.; RUTLEDGE L.; **GRiNS:A GRaphical INterface for creating and playing SMIL documents.** In WWW7 Conference, Computer Networks and ISDN Systems, volume 30(1-7), pages 519-529, Brisbane, Australia, April 1998.

VANNERVAR BUSH. "As We May Think". **The Atlantic Monthly.** Disponível em http://www.w3.org/History/1945/vbush/ Julho de 1945.

CARPANO M. **Automatic Display of Hierarchized Graphs for Computer Aided Decision Analysis.** *IEEE Transactions on Software Engineering*, SE-12(4):538–546, Abril 1980.

CLEMENTS P. A Survey of Architecture Description Languages. In: International Workshop on Software Specification and Design, 8., 1996, Paderborn, Alemanha, 1996. Disponível em ftp://ftp.sei.cmu.edu/pub/sati/Papers_and_Abstracts/Survey_of_ADLs.ps.

(Cormen et al., 2001) Cormen T. H., Leiserson C. E., Rivest R. L. "**Introduction to Algorithms (MIT Electrical Engineering and Computer Science)**", 2001.

(Costa, 1996) Costa R.F., **Um Editor Gráfico para Definição e Exibição do Sincronismo de Documentos Multimídia/Hipermídia**, Dissertação de Mestrado, Departamento de Informática, PUC-Rio, Agosto 1996.

(Cruz, 1998) Cruz T., *Workflow* a tecnologia que vai revolucionar processos. Editora Atlas, 1998.

(Dashofy et al., 2001) Dashofy, E., Hoek, A., Taylor, R. N., "**A Highly-Extensible, XML-Based Architecture Description Language**". In: Working IEEE/IFIP Conference on Software Architectures (WICSA 2001), pp. 103-112, Amsterdam, Netherlands, 2001 .

(DOM, 2004) "Document Object Model (DOM) Level 3", W3C Recommendation, Abril 2004.

(Engelbart, 1968) Engelbart D. C., "**A Research Center for Augmenting Human Intellect**," **Proc. 1968 Fall Joint Computer Conference (AFIPS)**, San Francisco, CA, December, 1968, 1968, pp. 395--410.

(Felix, 2004) Felix M. F, "**Análise formal de modelos de software orientada por abstrações arquiteturais**". Tese de Doutorado, PUC-Rio, Brasil, 2004.

(Furnas, 1986) Furnas G. "**Generalized Fisheye Views**". Proceedings of ACM SIGCHI'86 Conference on Human Factors in Computing Systems, Boston, 1986.

(Furuta et al., 1997) Furuta, R., Marshall, C. C. and Brenner, D., and Hsieh, H-W. Hypertext Paths and the World-Wide Web: Experiences with Walden's Paths. In ***Proceedings of the ACM Hypertext 97 Conference,*** pp. 167-176, Southampton, England, 1997.

(Gamma et al., 2002) Gamma E., Helm R., Johnson R. e Vlissides J., **Padrões de Projeto,** Editora Bookman, Porto Alegre, 2002

(Gansner et al., 2002) Gansner E., Koutsofios E., and North S., **Drawing graphs with dot**, February 4, 2002, Disponível em: http://www.research.att.com/sw/tools/graphviz/dotguide.pdf

(Gansner, 2003) Gansner E., **Drawing graphs with GraphViz,** Abril de 2003, Disponível em http://www.research.att.com/sw/tools/graphviz/libguide.pdf

(Garlan et al., 1997) Garlan D., Monroe R., Wile D. ACME: An Architecture Description Interchange Language, Proceedings of CASCON'97, Novembro 1997.

(Garlan, 1995) Garlan D. An Introduction to the Aesop System, disponível em http://www.cs.cmu.edu/afs/cs/project/able/www/aesop/html/aesop-overview.ps, Julho 1995.

(GraphViz, 2002) Graph Visualization Project Disponível em http://www.graphviz.org/ , 2002

(GXL, 2002) GXL - *Graph eXchange Language*. Disponível em: http://www.gupro.de/GXL/

(Hardman et al., 1993) Hardman, L.; Bulterman, D.C.A.; Van Rossum, G. **The Amsterdam Hypermedia Model: extending hypertext to support real multimedia**. Hypermedia Journal, v. 5, n. 1, 1993, p. 47-69.

(Hollingsworth, 2004) Hollingsworth D., Workflow Management Coalition - The Workflow Reference Model Disponível em http://wfmc.org

(Holt et al, 2001) Holt R. , Schürr A., Sim S. E. e Winter A, "**Graph eXchange Language**" 17 de Abril de 2001, Disponível em http://www.gupro.de/GXL/Introduction/background.html

(HTML, 1999) "HTML 4.01 Specification". **W3C Recommendation**, Dezembro 1999. Disponível em: http://www.w3.org/TR/html401/

(JAXP, 2003) JAXP – Java API for XML Processing. http://java.sun.com/xml/jaxp.

(Jourdan et al., 1998) Jourdan, M., Roisin, C. e Tardif, L. "**Madeus, an Authoring Environment for Interactive Multimedia Documents".** ACM Multimedia Conference 98, Inglaterra, Setembro de 1998, p. 267-272.

(Jourdan et al., 1999) Jourdan, M., Roisin, C. e Tardif, L. "**A Scalable Toolkit for Designing Multimedia Authoring Environment**s". Multimedia Authoring and Presentation: Strategies, Tools and Experiences Multimedia Tools and Applications Journal, Special Number, Kluwer Academic Publishers, 1999.

(Jühne et al., 1998) J. Jühne, A. T. Jensen, and K. Gronback. Ariadne: A Java-based guided tour system for the World Wide Web. **In Proceedings of the 7 th International World Wide Web Conference**, Brisbane, Australia, 1998. W3C.

(Kamada & Kawai, 1989) Kamada T. e Kawai S. "**An algorithm for drawing general undirected graphs**". *Information Processing Letters*, 31(1):7–15, Abril 1989.

(Larman, 2000) Larman C., Utilizando UML e Padrões – Uma Introdução à análise e ao Projeto Orientados a Objetos. Editora Bookman, 2002.

(Magee et al., 1996) Magee J., Kramer J. Dynamic Structure in Software Architectures, Proceedings of ACM SIGSOFT'96: 4th Symposium on the Foundations of Software Engineering (FSE4), pp. 3-14, San Francisco, California, Outubro 1996.

(Monroe, 1999) Monroe R.T. Rapid Development of Custom Software Architecture Design Environments, PhD Thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, agosto 1999.

(Moura, 2001) Moura S.M., **Relações Espaciais em Documentos Hipermídia**, Dissertação de Mestrado, Departamento de Informática, PUC-Rio, Agosto 2001.

(Muchaluat et al., 1997) Muchaluat D.C., Soares L.F.G., Costa F., Souza. **Graphical Structured-Editing of Multimedia Documents with Temporal and Spatial Constraints**, IV International Conference on Multimedia Modeling – MMM'97, Cingapura,pp 279-295, Novembro 1997.

(Muchaluat et al., 1998) Muchaluat D.C., Rodrigues R.F., Soares, L.F.G. "**WWW Fisheye View Graphical Browsers**". V Multimedia Modeling Conference, Lausanne, 1998.

(Muchaluat et al., 2003) Muchaluat-Saade D.C., Silva H.V., Soares L.F.G. "Linguagem NCL versão 2.0 para Autoria Declarativa de Documentos Hipermídia". IX Simpósio Brasileiro em Sistemas Multimídia e Web, Brasil, 2003.

(Muchaluat-Saade & Soares, 1995) Muchaluat-Saade D.C., Soares L.F.G. **Browsers e Trilhas no Sistema HyperProp.** I Workshop em Sistemas Hipermídia Distribuídos São Carlos, São Paulo - Julho de 1995.

(Muchaluat-Saade et al., 1998) Muchaluat-Saade D.C., Rodrigues R.F., Soares L.F.G. Navegação e Consulta no WWW Através de Browser Gráfico Usando

Visões Olho-de-Peixe. **XXV Seminário Integrado de Software e Hardware - Semish98**, Belo Horizonte, Minas Gerais, Agosto 1998.

(Muchaluat-Saade, 1996) Muchaluat-Saade D.C., **Browser e Trilhas para Documentos Hipermídia Baseados em Modelos com Composições Aninhadas**. Tese (Mestrado em Informática) – PUC-RJ, Rio de Janeiro 1996.

(Muchaluat-Saade, 2003) Muchaluat-Saade D.C., **Relações em linguagens de Autoria Hipermídia: Aumentando Reuso e Expressividade**. 206p. Tese (Doutorado em Informática) – PUC-RJ, Rio de Janeiro 2003. Disponível em ftp://ftp.telemidia.puc-rio.br/pub/docs/theses/2003_03_muchaluat.pdf Acesso em: 26 set 2003.

(Nelson, 1965) Nelson, Theodor Holm. ***A File Structure for the Complex, The Changing and the Indeterminate***, ACM 20th National Conference, pages 84-100, 1965. Disponível em http://portal.acm.org/citation.cfm?id=806036&dl=ACM&coll=GUIDE

(Noik, 1993) Noik E. G., "Layout-independent Fisheye Views of Nested Graphs", IEEE Symp. Visual Languages, 1993.

(North, 2002) North S. C., "**Drawing graphs with NEATO**". Disponível em: http://www.research.att.com/sw/tools/graphviz/neatoguide.pdf , 2002.

(Paula, 1999) Paula V. ZCL: A Formal Framework for Specifying Dynamic Distributed Software Architectures, Tese de Doutorado, Departamento de Informática, UFPE, Pernambuco, Brasil, Agosto 1999.

(Pinto, 2000) Pinto L.A.F. Autoria Gráfica de Estruturas de Documentos Hipermídia no Sistema HyperProp, **Dissertação de Mestrado, Departamento de Informática, PUC-Rio**, Agosto 2000.

(Rodrigues, 2003) Rodrigues R.F., **Formatação e Controle de Apresentações Hipermídia com Mecanismos de Adaptação Temporal.** Tese (Doutorado em Informática) – PUC-RJ, Rio de Janeiro 2003. Disponível em ftp://ftp.telemidia.puc-rio.br/pub/docs/theses/ 2003_03_rodrigues.pdf

(Schema, 2001) XML Schema Part 0: Primer, **W3C Recommendation**, disponível em http://www.w3.org/TR/xmlschema-0/.

(Schwabe & Medeiros, 2001) Schwabe D., Medeiros P.A., Especificação Declarativa de Aplicações Web em OOHDM, **VII Simpósio Brasileiro de Sistemas Multimídia e Hipermídia – SBMídia2001**, Florianópolis, Outubro 2001.

(Shneiderman, 1998) Shneiderman, B. **Designing the user interface: strategies for human-computer interaction**, Reading, Addison-Wesley, 3 ed. 1998.

(Silva et al., 2004) Silva H.V., Rodrigues R.F., Soares L.F.G. "Frameworks para Processamento de Linguagens XML Modulares", Relatório Técnico, Lab TeleMídia, PUC-Rio, Brasil, 2004.

(SMIL, 2001) Synchronized Multimedia Integration Language (SMIL 2.0), **W3C Recommendation Discipline,** disponível em http://www.w3c.org/TR/smil20, Agosto 2001.

(Soares et al., 2000) Soares L.F.G., Rodrigues R.F., Muchaluat-Saade D.C. Modeling Authoring and Formatting Hypermedia Documents in the HyperProp System, **ACM Multimedia Systems Journal,** 8(2):118-134, Março 2000.

(Soares et al., 2003) Soares L.F.G., Rodrigues R.F.,Muchaluat-Saade D.C. Modelo de Contexto Aninhados – versão 3.0. **Relatório Técnico, Laboratório Telemídia PUC-Rio**, Março de 2003.

(Sugiyama et al., 1981) Sugiyama K. , Tagawa S., and Toda M. Methods for Visual Understanding of Hierarchical System Structures. IEEE Transactions on Systems, Man, and Cybernetics, SMC-11(2):109–125, Fevereiro 1981.

(Sun, 1999) Sun Microsystems. **Java Media Framework, v2.0 API Specification**. 1999. Disponível em http://java.sun.com/products/java-media/jmf/

(TCL/TK, 2004) http://www.tcl.tk/software/tcltk/

(VGJ, 1988) VGJ (*Visualizing Graphs with Java*) Manual. Disponível em http://www.eng.auburn.edu/department/cse/research/graph_drawing/manual/vgj_manual.html

(Villard, 2000) Villard A XML based multimedia document processing model for content adaptation. **Digital Documents and Eletronic Publishing (DDEP'00)**, LNCS, Setembro de 2000.

(W3C, 2001) XML Schema Part 0: Primer - W3C Recommendation 02-May-2001. Disponível em: http://www.w3.org/TR/xmlschema-0/

(Warfield, 1977) Warfield J. Crossing Theory and Hierarchy Mapping. IEEE Transactions on Systems, Man, and Cybernetics, SMC-7(7):505–523, Julho 1977.

(Wills, 1997) Wills G.J., **NicheWorks - Interactive Visualization of Very Large Graphs.** In Graph Drawing '97 Conference Proceedings, pages 403-414, 1997. Rome, Italy, Springer-Verlag, or Journal of Computational and Graphical Statistics, vol. 8, no. 2, 190-212.

(Winter et al., 2002) Winter A., Kullbach B e Riediger V. **An Overview of the GXL Graph Exchange Language.** Springer Verlag: S. Diehl (ed.) Software Visualization · International Seminar Dagstuhl Castle, Germany, May 20-25, 2001 Revised Lectures, Disponível em http://www.gupro.de/GXL/Publications/publications.html

(XML, 2000) Extensible Markup Language (XML) 1.0 (Second Edition), **W3C Recommendation**, disponível em http://www.w3.org/TR/REC-xml, Outubro, 2000.

(XML, 2001) XML Schema 1.1, **W3C Recommendation**, disponível em http://www.w3c.org/XML/Schema.

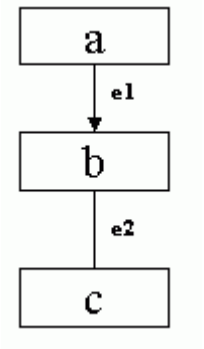
(Zschornack, 2003) Zschornack F., **Evolução de Esquemas de Workflow representados em XML,** Dissertação de Mestrado, Universidade Federal do Rio Grande do Sul, Porto Alegre – abril de 2003.

8
Apêndice A
GXL (Graph eXchange Language)

A linguagem GXL - *Graph Exchange Language* - foi projetada com base nas linguagens TA - *Tuple Attribute Language* (Universidade de *Waterloo*) e GRAX - *GRAph eXchange Format* (Universidade de *Koblenz*) e nos formatos declarativos das principais ferramentas de edição de grafos, tais como: PROGRES (*Graph Rewriting System*), GraphViz, daVinci, GML, XGMMl e GraphXML (Winter et al., 2002).

Um dos objetivos da linguagem GXL é tornar-se o formato padrão de intercâmbio entre as ferramentas de edição de grafos. A linguagem inclui suporte para representação de grafos hierárquicos e hiper-grafos (*hypergraph*), mas pode ser estendida para representar outros tipos de grafos tais como: grafos direcionados, grafos não-direcionados, grafos simples, grafos compostos etc (Holt et al, 2001).

Na Figura 8-1 é apresentado um exemplo de grafo em GXL.

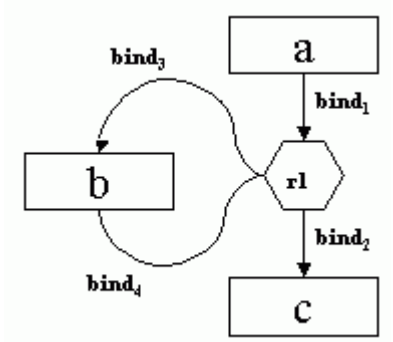



```
<gxl>
 <graph>
   <node id="a"/>
   <node id="b"/>
   <node id="c"/>
    <edge id = "e1" from="a" to="b"/>
    <edge id = "e2" from="c" to="b" orientation="undirected"/>
 </graph>
</gxl>
```

Figura 8-1 Exemplo simples de GXL

Em GXL, os vértices são definidos pelo elemento XML *"node"* e as arestas pelo elemento *"edge"*, tendo esse elemento atributos que identificam o vértice de origem *("from"*) e o vértice de destino (*"to"*). Com relação às arestas, é possível que o autor especifique a mesma como não-direcionada. Para isso, é necessário inserir o atributo *orientation = "undirected"*.

Além das entidades básicas *"node"* e *"edge"* pertencentes à linguagem, existe um elemento *"rel"* (relacionamento) responsável pela organização dos relacionamentos entre vários nós. A Figura 8-2 exemplifica um grafo GXL com o elemento *"rel"* e seu respectivo código.

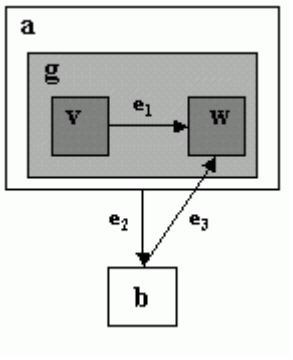



```
<gxl>
 <graph>
   <node id="a"/>
   <node id="b"/>
   <node id="c"/>
   <rel id ="r1">
       <link ref ="a" role="bind1" direction="in" />
       <link ref ="c" role="bind2" direction="out" />
       <link ref ="b" role="bind3" direction="out" />
       <link ref ="b" role="bind4" direction="none" />
    </rel>
 </graph>
</gxl>
```

Figura 8-2 Grafo GXL com elemento *rel*

O elemento *"rel"* é formado por um conjunto de elementos *"links"*, cada qual possuindo um atributo *"ref"* que faz referência a um nó definido no documento, um atributo *"role"* que é usado como rótulo do *"link"* no desenho e um atributo *"direction"* que determina qual o sentido da aresta. Na Figura 8-2, o elemento *"rel"* está representado como um hexágono (*"r1"*).

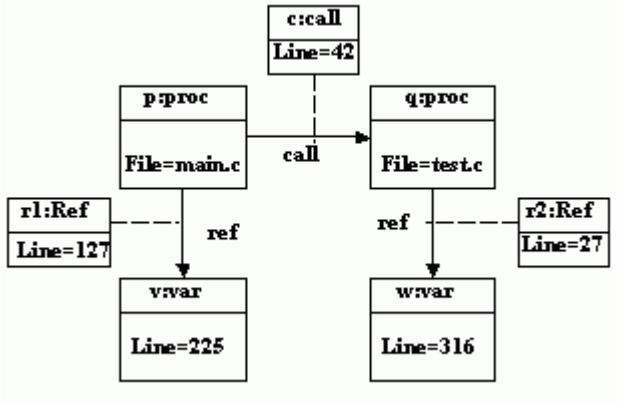
A representação de grafos compostos em GXL é feita através da definição do elemento *"graph"* como filho de um elemento *"node"*. No entanto, o conceito de composicionalidade não é tratado, pois uma aresta pode interligar diretamente dois vértices que estejam contidos em vértices compostos distintos, sem a necessidade de mapeamentos. A Figura 8-3, ilustra um grafo composto e seu respectivo código em GXL. Observe que o elemento *"graph"* com *"id"* igual a *"g"* está definido como filho do elemento *"node"* com *"id"* igual a *"a"*, e que a aresta *"edge"* com *"id"* igual a *"e3"* relaciona diretamente dois vértices de composições (sub-grafos) diferentes.




```
<gxl>
 <graph>
   <node id="a"/>
      <graph id="g">
          <node id= "v"/>
          <node id= "w"/>
          <edge id = "e1" from="v" to="w"/>
       </graph>
    </node>
   <node id="b"/>
   <edge id = "e2" from="a" to="b"/>
   <edge id = "e3" from="b" to="w"/>
</gxl>
```

Figura 8-3 Grafos compostos em GXL

Uma das várias aplicações da linguagem GXL é a representação de uma especificação em UML (*Unified Modelling Language*) (Larman, 2000). A Figura 8-4, apresenta uma configuração de UML e seu código em GXL.




```
<gxl>
  <graph>
    <node  id="p">
      <attr name="File">
          <string>main.c</string> </attr>
    </node>
    <node  id="q">
      <attr name="File">
          <string>teste.c</string></attr>
    </node>
    <node  id="v">
      <attr name="Line">
          <int> 225 </int></attr>
    </node>
    <node  id="w">
      <attr name="Line">
          <int>316 </int></attr>
    </node>
    <edge id ="r1" from="p" to="v">
        <attr name="Line">
          <int>127 </int></attr>
    </edge>
    <edge id ="r2" from="q" to="w">
        <attr name="Line">
          <int>27 </int></attr>
    </edge>
    <edge id ="c" from="p" to="q">
        <attr name="Line">
          <int>42 </int></attr>
    </edge>
  </graph>
</gxl>
```

Figura 8-4 UML representada em GXL