THAÍS VASCONCELOS BATISTA

CONTROLE DE VERSÕES NO MODELO HIPERMÍDIA DE CONTEXTOS ANINHADOS

DISSERTAÇÃO DE MESTRADO

Departamento de Informática

Rio de Janeiro, 18 março de 1994

THAÍS VASCONCELOS BATISTA

CONTROLE DE VERSÕES NO MODELO HIPERMÍDIA DE CONTEXTOS ANINHADOS

Dissertação apresentada ao Departamento de Informática da PUC/RJ como parte dos requisitos para obtenção do título de Mestre em Informática : Ciência da Computação.
Orientador : Luiz Fernando Gomes Soares

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro

Rio de Janeiro, 18 de março de 1994

A memória do meu pai
A minha mãe
A Jair
A Larissa

AGRADECIMENTOS

- A Deus, por me guiar, proteger e dar forças.
- A Jair, por sua dedicação e apoio em todos os momentos da minha vida.
- A minha mãe, que nunca mediu esforços para me fazer chegar a esse momento.
- A CAPES pela ajuda financeira.
- Ao Prof. Luiz Fernando Gomes Soares, por sua orientação a este trabalho.
- A Profa. Noemi De La Rocque Rodriguez, por toda a atenção comigo e com o meu trabalho, por sua amizade e orientação. Além disso, todas as manifestações de sua parte sempre foram um estímulo e incentivo a mim e ao meu trabalho.
- A Sílvio Soares Bandeira, meu grande amigo, que muito me ajudou com seus conhecimentos, sua amizade e confiança.
- A minha família e demais amigos que de alguma forma contribuiram para a realização deste trabalho.
- Ao Departamento de Informática da UFRN por ter me dado a oportunidade de utilizar seus equipamentos.

RESUMO

O desenvolvimento de Sistemas Hipermídia requer um modelo conceitual que ofereça estruturas para organização das informações e operações para sua manipulação. Adicionalmente, este modelo deve dispor de um mecanismo para gerenciamento de versões visando abranger aplicações que necessitam manter, de forma consistente, mais de uma versão de seus dados. Além disso, interoperabilidade e portabilidade deve ser possível, para tanto, o modelo deve seguir uma arquitetura aberta. O Modelo de Contextos Aninhados é um modelo conceitual hipermídia projetado seguindo uma arquitetura aberta, e apresentando abstrações para organização e navegação em documentos, gerenciamento de versões e conformidade com o padrão MHEG para intercâmbio de objetos e seus atributos.

Este trabalho consta de três objetivos básicos. Primeiro, apresentar a estrutura proposta pelo Modelo de Contextos Aninhados e o conjunto de mecanismos para controle de versões. Em seguida, propor uma estratégia para notificação de alteração, ativada quando em uma versão ocorre alguma alteração que pode causar inconsistência nas informações. Finalmente, comentar o desenvolvimento de um protótipo mono-usuário com controle de versões que implementa os conceitos introduzidos pelo modelo em uma biblioteca de classes extensíveis. Sobre esta biblioteca foi construída uma interface para autoria e navegação.

ABSTRACT

The development of Hypermedia Systems requires a conceptual model that offers structures to organize informations and operations for their manipulations. Adicionaly, that model must have a mechanism for version management, in order to reach applications which have to keep in a consistent form, more than one version of the data. Besides, interoperability and portability have to be possible. So the model must follow an open architecture. The Nested Context Model is a hypermedia conceptual model projected based on an open architecture, including abstractions for document organization and navigation, versions management and conformance with the MHEG proposal for objetcs interchange.

This research has three basic objectives. First, describe the structure proposed by the Nested Context Model and the mechanism to version control. Following, a notification strategy is proposed to maintain consistency. Finally, discusse the development of a prototype with version control that implements the concepts introduced by the model in a library of classes. Over this library, an interface to authoring and navigation was constructed.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Este capítulo se dedica a expor os motivos que induziram a definição de um novo modelo conceitual hipermídia, bem como os fatores que determinaram a incorporação do tratamento de versões nesse sistema. Como consequência, argumenta-se o porquê de se desenvolver mais um trabalho envolvendo o tema *Controle de Versões no Modelo Hipermídia de Contextos Aninhados*, mostrando o que traz de original e quais as novidades em relação aos anteriores [42, 43].

1.1 Motivação

Os sistemas de informação tradicionais começaram a surgir na época em que a tecnologia apenas permitia a manipulação de dados alfa-numéricos. Por exemplo, os sistemas de banco de dados convencionais foram projetados para atender as aplicações orientadas a registro, onde dados alfa-numéricos são suficientes para representar suas informações.

O surgimento de tecnologias mais avançadas como estações de trabalho gráficas, redes de alta velocidade para acesso rápido aos dados distribuídos, dispositivos com grande capacidade de armazenamento, processadores mais rápidos, possibilitaram o aparecimento de sistemas capazes de atender a outros tipos de aplicações além daquelas orientadas a registro. Essas aplicações usam tipos de dados especiais como texto, gráfico, som, imagem, etc. Tais sistemas são chamados *Sistemas Multimídia* por incluírem tratamento a vários tipos de mídia.

Emprega-se o termo *Sistemas Hipermídia* quando as informações multimídia são armazenadas em um conjunto de nós interligados que podem ser consultados de maneira não sequencial, ou seja, utilizando o paradigma de *Hipertexto*. Em [21] *Hipermídia* é considerada um estilo de construir sistemas para representação e gerenciamento das informações em uma rede de nós multimídia conectados por elos.

Atualmente, o uso desses sistemas tem sido muito difundido, com isso, suas limitações têm se tornado aparente. Uma forte limitação se encontra quando constata-se que a maioria dos sistemas hipermídia são voltados para domínios bem particulares e desenvolvidos como aplicações auto-contidas, sem preocupar-se com a reusabilidade de código entre aplicações e intercâmbio de informações. Porém, como os sistemas hipermídia podem ser utilizados em domínios bastante diversos, então devem satisfazer aos requisitos comuns a diferentes classes de aplicações.

O Modelo de Contextos Aninhados [7] surgiu não para se tornar mais um modelo conceitual disponível na literatura, mas com o intuito de prover um modelo hipermídia genérico, com mecanismos de acesso e armazenamento das informações visando ser usado por diferentes aplicações multimídia. O objetivo é formar um núcleo básico que pode ser refinado por cada aplicação para satisfazer suas particularidades.

Segundo [6] a chave para o sucesso do hipertexto é um eficiente método para gerenciamento dos dados de forma independente da aplicação. O Modelo de Contextos Aninhados procura atingir essa independência usando uma arquitetura aberta baseada em camadas, descrita em [11], que para permitir intercâmbio de informação utiliza a proposta de padrão MHEG [24] da ISO[1] onde é definido o formato dos objetos a serem intercambiados.

Trabalho cooperativo é uma atividade essencial. Isto torna evidente a necessidade dos sistemas hipertexto permitirem várias pessoas trabalhando juntas no mesmo documento,

[1]Multimedia and Hypermedia Information coding Expert Group da International Standards Organization

ao mesmo tempo, e, de preferência usando diferentes máquinas conectadas em rede. Desta forma, as aplicações multimídia exigem que os modelos que as suportam sejam concebidos considerando a necessidade de permitir o trabalho cooperativo, e, consequentemente, de preservar mais de um estado de seus objetos, ou seja, as versões de suas informações. Assim, para afirmar sua proposta de generalidade, e ampliar o seu campo de abrangência, atingindo as aplicações que necessitam manter, de forma consistente, mais de uma versão de seus dados, o Modelo de Contextos Aninhados decidiu integrar tratamento de versões. Uma proposta inicial do Modelo de Versões como extensão ao Modelo básico foi descrito em [31, 32] porém, algumas questões não foram cobertas por completo e certas soluções emitidas não são plenamente satisfatórias para um ambiente de trabalho cooperativo.

Muitas questões emergem quando um sistema se propõe a gerenciar o processo de criação de versões. Um ponto essencial é a preservação da consistência dos objetos. Como, em um sistema hipermídia, os objetos (nós) se referenciam, para que tais referências permaneçam consistentes, é necessário um mecanismo para avisar aos proprietários dos objetos, das mudanças sofridas pelas versões por eles utilizadas. Daí a justificativa para se delinear um mecanismo para Notificação de Mudança.

A motivação deste trabalho reside não apenas em apresentar o modelo conceitual hipermídia de contextos aninhados compreendendo estruturas e operações fundamentais ao tratamento de versões, mas também, em propor uma estratégia para resolver o problema da Notificação de Mudança e, principalmente, em tornar operacional um protótipo do modelo, a fim de avaliar quão satisfatórias são as propostas apresentadas. Este protótipo deve ser construído na forma de uma biblioteca de classes extensíveis em uma linguagem orientada por objetos, para ser utilizada por diferentes aplicações. Sobre esta biblioteca deve ser desenvolvida uma aplicação para autoria e navegação em hiperdocumentos onde esteja disponível uma interface para criação e recuperação de documentos. O caráter de extensibilidade do protótipo permite que outras aplicações possam ser desenvolvidas utilizando sua biblioteca de classes.

Em resumo, embora já existam vários trabalhos que descrevem o MCA com gerenciamento de versões [42, 43], este trabalho ultrapassa a fronteira dos anteriores, oferecendo uma estratégia de Notificação de Mudança e um ambiente para autoria e navegação onde estão implementados os conceitos introduzidos pelo modelo. Assim, a originalidade deste trabalho consiste nestes dois últimos aspectos citados. A pretensão com o design do protótipo é desenvolver uma ferramenta geral para construção de diferentes aplicações hipermídia. Para isto, este protótipo deve capturar a idéia geral do MCA, gerenciar a criação de versões, fornecer operações nas entidades básicas (nós, âncoras e elos) e, sobretudo, ser fácil de estender e adaptar.

1.2 Organização da Dissertação

Antes de abordar o tema principal deste trabalho, que é o modelo de contextos aninhados com suporte a versões, é necessário descrever os conceitos subjacentes aos sistemas hipertextos e ao tratamento de versões. Este é o assunto do segundo capítulo. Ainda neste capítulo, são apresentados um conjunto de requisitos e de problemas relacionados à integração de versões em sistemas hipermídia, bem como diversos trabalhos que buscam soluções para esses problemas. Para concluir o capítulo, faz-se um paralelo entre os trabalhos descritos e o Modelo em questão.

O capítulo 3 introduz os conceitos do modelo básico de contextos aninhados e, em seguida, descreve os aspectos incluídos para gerenciamento de versões e trabalho cooperativo, terminando com uma breve apresentação da arquitetura aberta na qual o modelo foi baseado.

A proposta de Notificação de Alteração é tratada no capítulo 4, após o levantamento dos requisitos exigidos pelas aplicações multimídia, bem como aqueles exigidos pelo Modelo de Contextos Aninhados.

O capítulo 5 disserta sobre o protótipo que implementa o Modelo de Contextos Aninhados com gerenciamento de versões. É apresentado o ambiente de implementação, as decisões de design da biblioteca de classes e da ferramenta construída sobre ela, bem como as limitações dessas porções da implementação. Finalmente, apresenta-se como fazer evoluções no protótipo.

Concluindo esta dissertação, o capítulo 6 destaca as contribuições e apresenta sugestões para trabalhos futuros.

Capítulo 2

Versões e Hipertexto

2.1 Introdução

Este capítulo tem como objetivos apresentar os conceitos associados ao tema dessa dissertação e os termos comuns a cada um deles, analisar a importância da junção de tais conceitos, bem como os problemas decorrentes dessa junção, descrever alguns trabalhos que endereçam soluções para esses problemas e por fim fazer uma breve análise dos pontos abordados no Modelo de Contextos Aninhados relacionando-os com os requisitos e os trabalhos descritos.

O conceito e características de hipertexto são descritos na seção 2.2.

A seção 2.3 apresenta o conceito de versões, aborda os termos normalmente utilizados em modelos de versões e lista uma série de requisitos que os modelos devem satisfazer.

A importância da inclusão de um mecanismo para controle de versões nos sistemas hipertexto é o ponto abordado na seção 2.4.

Na seção 2.5 são relacionados os problemas que comumente surgem em sistemas que adicionam mecanismos para tratamento de versões.

Na seção 2.6 são apresentados conceitos básicos de algumas propostas de modelos de versões encontrados na literatura. Esta lista, apesar de não ser exaustiva, reflete as soluções fornecidas por diferentes modelos para os problemas relacionados na seção 2.5 e também para alguns outros problemas particulares de cada abordagem. Há um considerável volume de trabalhos que endereçam o problema de controle de versões, principalmente para aplicações CAD e Engenharia de Software. Em contraste, existem poucos trabalhos que focam mecanismos de versões em sistemas hipertexto.

Finalmente, na seção 2.7, é feita uma breve apresentação do Modelo de Contextos Aninhados onde são destacados quais os requisitos que o modelo satisfaz, comparando com os aspectos já abordados em alguns dos modelos descritos na seção 2.6.

2.2 O Conceito de Hipertexto

A idéia de hipertexto surgiu há muitos anos quando constatou-se que a mente humana é associativa por natureza e essa associatividade da mente é bastante diferente da maneira linear na qual livros e informações são geralmente organizados. Um livro, como uma estrutura linear, representa um único caminho através de um tópico. Em contraste existem muitos caminhos possíveis através de estruturas associativas. O hipertexto é uma ferramenta para construção e uso de estruturas associativas.

A idéia emergiu também da necessidade de armazenamento e recuperação de textos convencionais não lineares, e de um sistema automatizado que manipule e armazene enormes quantidades de referências cruzadas entre as informações mantidas.

Hipertexto pode ser definido como um sistema de gerenciamento da informação na qual dados são armazenados em redes de nós conectados por elos [40]. Quando o conteúdo dos nós pode ser gráficos, voz, vídeo, além de texto, o sistema é referido como Hipermídia. Os elos representam o meio de relacionamento entre os nós. Para percorrer a rede de informações existe um mecanismo de recuperação chamado navegação, que considera a conexão entre as informações. Segundo [47], a navegação é o processo de seguir um caminho através de um grafo. Em [30], navegar em um hipertexto significa desenhar um percurso em uma rede de informações. A recuperação da informação através de elos associativos oferece a alternativa de acessar os dados de vários pontos do hiperdocumento. Essa estrutura associativa, não encontrada nos métodos convencionais (como processadores de texto, por exemplo), pode introduzir alguns problemas como:

- o usuário pode perder a orientação dentro do documento. As aplicações hipermídia geralmente têm uma estrutura complexa na qual pode-se navegar, mas não visualizar totalmente; esta é a razão porque o usuário pode perder o senso de onde se encontra e para onde pretende ir.
- a sobrecarga cognitiva causada pelo esforço e concentração requerida para manter os vários caminhos de acesso aos dados.

Esses problemas podem ser resolvidos fornecendo ao usuário mapas para exibir a estrutura gráfica do sistema hipermídia, ou marcas para mostrar os nós visitados, ou ainda pontos de referência representando posições familiares [37].

2.3 O Conceito de Versões

O termo versão não é padronizado. Vários termos têm sido encontrados na literatura como sinônimos do termo versão e às vezes com a finalidade de acrescentar algumas propriedades. Por exemplo, em [16] um objeto de projeto tem o ciclo de vida representado por alternativas, para cada alternativa existem revisões e para cada revisão, versões. Em [26], alternativas significam versões sucessoras de uma mesma versão. O termo revisão é usado em [45] como sinônimo de versão.

Segundo [16], uma versão de um objeto é uma representação diferente dos mesmos aspectos do objeto, ou seja, é uma descrição completa do objeto dentro do grau de abstração escolhido pelo proprietário.

Em [3], versões são definidas como sendo variações do mesmo objeto que são relacionadas pela história de sua derivação.

Segundo [17], uma definição válida para todos os domínios de aplicações: uma versão é um estado do objeto que o usuário ou o sistema deve conservar.

Versões são usadas para diferentes propósitos na literatura:

- **Para controle de concorrência:** um pequeno número de versões é mantido por tempo limitado de forma a isolar o usuário de mudanças realizadas por outros usuários [36].
- **Para Recuperação de Falha:** versões são usadas para representar estados consistentes de bancos de dados para o qual o sistema pode retornar depois de alguma falha.
- **Para registrar diferentes alternativas de um objeto:** o uso de versões permite que se possa manter uma série de opções diferentes para um mesmo tópico, possibilitando o usuário explorar simultaneamente várias configurações alternadas.
- **Para aumentar a performance em Sistemas Distribuídos:** Várias cópias do mesmo dado existem em diferentes cidades. Apesar da consistência ser importante, dados obsoletos devem ser tolerados em alguma cidade por um curto período de tempo para evitar frequentes transmissões de dados.
- **Para implementar bancos de dados livres de atualizações:** Atualizações não são feitas substituindo os velhos valores, mas gerando uma nova representação do objeto afetado. Isto produz uma grande sequência de versões que pode ser útil para propósitos de auditoria e consultas do histórico das informações.
- **Para manter o histórico dos documentos:** permitir que sejam preservadas versões cronológicas dos documentos é essencial em um ambiente cooperativo.
- **Para representar a mesma informação em diferentes mídias:** Em [7] uma mesma informação representada em diferentes mídias corresponde a diferentes versões da informação.

Em [28] foram descritos alguns modelos de versões existentes na literatura. Os autores observaram que a maioria das variações entre os modelos consiste apenas em terminologias diferentes para o mesmo conceito e não em diferentes conceitos. Assim, eles produziram um catálogo de características comuns na maior parte dos modelos e propuseram uma terminologia unificada. Ou seja, enumeraram uma série de requisitos que os modelos de versões devem satisfazer. A seguir serão relacionados os comentários mais importantes:

- **Organização do conjunto de versões**: os conceitos chaves introduzidos aqui são histórico de versões, objetos genéricos, relacionamento ancestral/ascendente, derivação principal, ramificação e versão corrente.

  Um objeto pode ser referido de forma global através do conceito de objeto genérico. O objeto genérico representa uma abstração das versões de um objeto; apresenta propriedades comuns a todas as suas versões; mantém as informações sobre o conjunto de versões associado.

  Relações ancestral/descendente são usados para ordenar o histórico de versões. Podem ser implementados usando elos e também são chamados em outros trabalhos de conjunto de versões, grafo de versões e grafo de refinamento.

  Uma única instância de uma versão é identificada como versão corrente e é importante para desintegrar a noção de versão corrente da última versão criada.

  Derivação principal é o caminho da raiz até a versão corrente. Isto é o mesmo que caminho principal e ramificação default.

  O histórico de versões é representado pelos elos que registram as relações ancestral/descendente.

- **Composições Hierárquicas (Configurações)**: Configurações são versões de objetos compostos os quais são constituídos por versões de seus componentes.

- **Ligações (Referências) Dinâmicas e Estáticas**: referências estáticas referenciam versões específicas enquanto que referências dinâmicas referenciam objetos genéricos. Assim, uma referência dinâmica significa que o usuário não sabe qual versão do componente deverá ser usada, tais referências devem ser resolvidas pelo sistema segundo algum critério. O critério mais comum é substituir a referência pela versão default ou versão corrente. Em alguns modelos a versão default é a última versão criada, em outros, o usuário pode indicar qual versão deve ser a default.

- **Configurações Dinâmicas e Estáticas**: A configuração é estática quando todas as referências forem estáticas; caso alguma referência seja dinâmica, a configuração é dita dinâmica.

- **Propagação e Notificação de Mudança**: Quando objetos compostos utilizam como componentes objetos que apresentam versões, modificações nos componentes podem causar alterações nos objetos que os utilizam [19]. As ações a serem realizadas devido a modificações nos componentes podem ser: notificação e propagação de mudança. O problema da notificação de mudança pode ser resolvido por qualquer mecanismo baseado em mensagens ou em sinais. Já para o problema da propagação, a limitação do escopo da propagação é um dos mecanismos mais utilizados.

- **Mecanismos para o compartilhamento de objetos**: os conceitos básicos para o compartilhamento de objetos consiste em separar as áreas de trabalho em públicas, de projeto/grupo e privadas, com objetos sendo movidos entre as áreas através de operações de checkin/checkout.

Demais trabalhos que endereçam modelo de versões, apresentam outros requisitos além dos descritos acima:

- **Suporte a um conjunto de versões:** manter uma trilha de versões para cada entidade individual não é um mecanismo de versões completo. Em geral, usuários irão fazer mudanças coordenadas em um número de entidades na rede em um determinado tempo. O usuário pode desejar colecionar as versões individuais resultantes em um único conjunto de versões para referência futura.

- **Pequeno Esforço cognitivo na criação de versões:** a criação explícita de versões irá resultar em um grande esforço cognitivo. É necessário procurar soluções para gerenciar a criação de versões.

- **Imutabilidade de Versões:** Osterbye [35] afirma que deve ser possível definir a questão da mutabilidade dos aspectos de um nó. Ou seja, deve-se poder determinar os aspectos de um nó que podem ser mutáveis e quais podem ser imutáveis. Tais aspectos compreendem o conteúdo, elos e atributos.

- **Versões de Elos:** alguns trabalhos afirmam que pode ser interessante se ter um mecanismo de versões para elos.
- **Versões da Estrutura:** é desejável ter uma noção de versões da estrutura do hipertexto, pois retornar ao estado prévio do hipertexto inteiro é tão desejável quanto retornar a um simples nó.
- **Suporte ao desenvolvimento exploratório:** o suporte ao desenvolvimento exploratório acarreta o problema de como congelar um estado, pois toda a estrutura onde o autor está trabalhando deve poder ser congelada.

Técnicas para Economizar Memória: para economizar espaço, em vez de armazenar inteiramente cada módulo do grupo de versões, apenas diferenças entre os módulos devem ser armazenadas, isto pode ser feito usando a técnica de deltas. A desvantagem desta técnica é que recuperar uma versão específica requer computação. Outro problema é que os algoritmos de deltas são específicos para determinada mídia. Assim, técnicas para economizar memória são de responsabilidade da camada de armazenamento, não fazendo parte do modelo de dados. Portanto, este aspecto não será considerado requisito para o modelo de versões.

2.4 Importância de Versões para Hipertextos

O potencial dos sistemas hipertexto de ajudar na organização e manipulação da informação tem contribuído para que diferentes tipos de aplicações utilize o hipertexto como sistema de gerenciamento dos seus dados.

Segundo [21] os sistemas hipertexto possuem muitas limitações porque o simples modelo de nós e elos não é completo o suficiente para resolver tarefas requeridas por várias aplicações. A falta de mecanismos para tratamento de versões na maioria dos hipertextos existentes é citada como uma das deficiências.

Em [13] os bancos de dados tradicionais são ditos obsoletos para uso em sistemas CAD e Engenharia de Software por não fornecerem mecanismos para controle de versões e gerenciamento de configurações. Além disso, o hipertexto é ressaltado como uma excelente ferramenta a ser usada por tais sistemas caso suporte e gerencie a criação de versões.

Por exemplo, em Engenharia de Software, o uso de hipertexto como auxílio ao desenvolvimento atende as necessidades de conectar logicamente especificação, implementação e documentação de todo o sistema. Essa organização facilita o entendimento do sistema e permite acesso imediato às informações. Outro aspecto importante é que o hipertexto possibilita manter restrições de integridade entre as partes do sistema. Como o hipertexto interliga tais partes, uma modificação em um objeto dependente de outros pode acarretar a ativação de processos responsáveis pela manutenção da integridade. Porém, para caracterizar a evolução das partes dos sistema, é necessário registrar todos os estágios do processo evolutivo e isso é feito através de versões. Caso não haja gerenciamento de versões, as alterações ficam sendo feitas sobre o próprio objeto, impossibilitando assim a caracterização dos seus estágios intermediários.

Segundo [13] sistemas hipertextos podem fornecer um excelente sistema de armazenamento para aplicações CAD. Na comunidade CAD, há um consenso geral que o controle de versões é uma das funções mais importantes para o design de seus sistemas [9], visto que os projetistas geralmente necessitam gerar e experimentar múltiplas versões de um projeto antes de selecionar o que satisfaz os requisitos solicitados.

Aliado aos fatores anteriores, a cooperatividade constitui outro fator determinante da importância de versões para hipertextos, pois grande parte das aplicações que manipulam dados multimídia são usadas por grupos de pessoas que cooperam na criação de documentos e cada pessoa necessita interagir com a rede de informações e criar versões dessas informações.

2.5 Problemas Associados com Versões

Adicionar um mecanismo para gerenciamento de versões em hipertexto desperta alguns problemas:

- **Semântica de referências entre objetos:** objetos podem ser definidos como compostos de outros objetos [19]. A composição de objetos é expressa pela inclusão de uma referência do objeto composto para o componente. Uma referência a um objeto deve ser feita incluindo no objeto composto uma das seguintes informações sobre o objeto componente:

  - uma versão específica deste objeto (referência estática)
  - mais nova versão do objeto ao longo de um ramo específico do grafo de versões ou uma versão do objeto que satisfaz uma questão particular (referência dinâmica)

  O tipo de referência que é suportado é uma decisão que afeta todo o sistema, principalmente os elos e as estruturas que agrupam o conjunto de nós, visto que, um problema proeminente quando introduzido versões de nós é determinar para qual elemento, no grupo de versões, o elo aponta.

- **Compartilhamento de versões:** nos sistemas hipertexto os nós podem ser compartilhados por diferentes aplicações e grupos de usuários. O compartilhamento de verŝoes pode gerar problemas quando versões inconsistentes estão sendo compartilhadas. Para evitar que isto ocorra é necessário que o modelo de versões adicione algum mecanismo para garantir que apenas versões consistentes possam ser compartilhadas.

- **Propagação de Versões:** os sistemas hipertexto geralmente dispõem de uma estrutura para agrupar um conjunto de nós e elos. Quando ocorre mudanças em algum dos subconjuntos desta estrutura, surge o problema da propagação. Em [28] propagação de mudança é definido como o processo que automaticamente incorpora novas versões em configurações. A propagação é necessária porque mudanças em um dos objetos pode causar efeitos nos demais objetos que o referenciam ou são referenciados por ele [25]. Por exemplo, em um sistema de Engenharia de Software, a atualização de um documento de "design" implica em modificações a serem feitas no código que o implementa.

  Os dois problemas quanto a restrição da propagação são:

  - como limitar o escopo da propagação: raramente o projetista deseja criar uma nova configuração de todos os componentes da hierarquia.

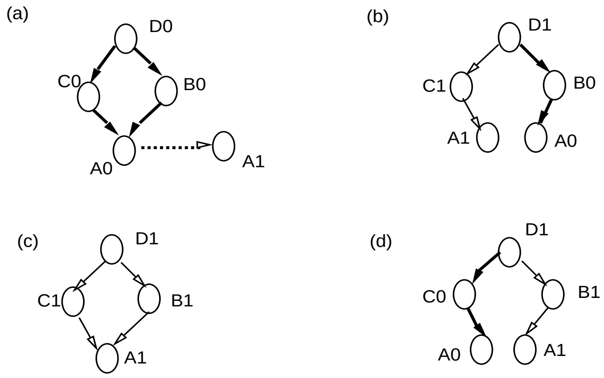

Figura 2.1: Problema da Ambiguidade

- como evitar a ambiguidade: a configuração resultante pode ser totalmente diferente dependendo da sequência na qual a propagação é realizada. O problema da ambiguidade está ilustrado na Figura 2.1. A figura 2.1(a) ilustra a criação da versão A1 descendente de A0. A figura 2.1(b) mostra a mudança sendo propagada ao longo do caminho ACD. Analogamente, a figura 2.1(d) mostra a propagação ao longo do caminho ABD. A figura 2.1(c) ilustra a propagação ao longo dos dois caminhos. Apesar de, geralmente, uma nova versão poder suplantar todas as utilidades da versão que ela substitui (caso da figura 2.1(c)), há situações nas quais é útil ter mais que uma versão do mesmo objeto em uma dada configuração.

- **Notificação de Alteração:** em ambientes de trabalho cooperativo um documento é compartilhado por vários usuários simultaneamente. Caso um dos usuários altere alguma parte do documento os demais devem ser notificados da alteração para evitar inconsistências. Na literatura existem alguns mecanismos para notificação, cada um atendendo a diferentes requisitos. Cabe ao sistema hipertexto escolher o que mais se identifica com as necessidades das suas aplicações, podendo inclusive adaptá-lo conforme seja necessário. Além disso, um outro problema que deve ser decidido quando há notificação de mudanças é o **escopo da notificação**. Como versões fazem referências a outras de forma recursiva, o problema é decidir quais versões devem ser notificadas, se apenas as que fazem referência direta ou se todas, ou seja, as que fazem referência direta ou indireta.

Além dos problemas descritos acima, a integração de um modelo de versões agrava o problema do "Esforço Cognitivo" durante a criação de versões e desorientação durante a navegação.

2.6 Trabalhos Correlatos

2.6.1 O Modelo de Chou

Este trabalho [9] discute os aspectos operacionais e semânticos do tratamento de versões para sistemas CAD com arquitetura distribuída. São abordados aspectos referentes a semântica da criação e manipulação de versões, identificação de versões, notificação e escopo de mudança, além de questões relativas a implementação.

A semântica das operações sobre versões constitui um dos pontos importantes do modelo. A arquitetura do sistema é baseada em uma base de dados pública, várias bases de dados de projeto e várias bases de dados privadas permitindo diferentes níveis de compartilhamento e diferentes *capacidades de versões* (operações possíveis).

Um objeto CAD consiste de duas partes: a interface e a implementação. O modelo define que uma versão de um objeto é outro objeto que compartilha a mesma interface mas possui implementações diferentes.
Para manter o projeto gerenciável, um objeto CAD é representado como uma hierarquia de configurações no qual o componente consiste de componentes mais primitivos. Isto é feito incluindo em um objeto referências a objetos mais primitivos.

É definido um modelo de transações em termos de projetos e cooperação entre projetistas. Tal modelo é composto por três bases de dados: pública, de projeto (compartilhada por projetistas do mesmo projeto) e privada. A cada uma dessas bases de dados está associada um diferente tipo de versão, respectivamente: liberadas, de trabalho e transientes.

Versões liberadas existem apenas na base pública, não podem ser alteradas ou removidas e seus componentes são apenas versões liberadas.

Versões de trabalho residem na base de dados de projeto ou na base de dados privada. São consideradas estáveis, e portanto, não podem ser alteradas. Podem ter como componentes versões liberadas e de trabalho.
Versões transientes existem apenas em uma base de dados privada. E qualquer operação pode ser executada sobre ela. Pode ter como componente qualquer tipo de versão.

As operações sobre versões dependem do tipo de versão. Cada tipo possui um conjunto de operações disponíveis (capacidades de versões). Versões liberadas não podem ser alteradas nem removidas; podem derivar versões transientes; e são criadas através da promoção de versões de trabalho. Versões de Trabalho não podem ser alteradas; só podem ser removidas pelo administrador da base de dados de projeto ou pelo proprietário da base privada; pode derivar versões transientes. Versões Transientes podem ser alteradas e removidas pelo proprietário; podem ser derivadas de versões de trabalho ou liberadas; e podem ser promovidas a versões de trabalho.

O modelo define três formas de criar versões: derivação, promoção e operações de *checkin/ checkout*. A operação de promoção consiste em mudar o estado da versão para um estado de maior estabilidade. A versão pode ser promovida de versão transiente para versão de trabalho ou de versão de trabalho para versão liberada. As operações de Checkin/Checkout copiam uma versão de uma base de dados A para uma base de dados B de nível diferente, sem remover a versão original. Na operação de *Checkout* a base A dever ser de nível maior que B na hierarquia. Na operação de *Checkin*, B deve estar em nível superior a A.

Como versões podem existir em qualquer lugar da hierarquia de base de dados, o nome completo para cada versão de um objeto é uma tripla (nome-do-objeto, nome-da-base-de-dados, número-da-versão). Há duas maneiras de fazer ligação entre versões: referência estática ou referência dinâmica. A referência estática inclui o nome completo do objeto, ou seja, (nome-do-objeto, nome-da-base-de-dados, número-da-versão). Na referência dinâmica é necessário apenas especificar o nome do objeto, as outras partes o sistema seleciona o default, ou seja, se não é especificada o nome da base de dados, o sistema assume que é a mesma base do objeto que está fazendo a referência. Como extensão da idéia de referência dinâmica é definido um Contexto. Um contexto constitui a especificação de versões default para uma configuração particular de um objeto de projeto complexo.

O usuário pode definir vários contextos para experimentar diferentes configurações alternativas do objeto. Por exemplo, suponha um objeto de projeto constituído por outros objetos: A, B e C. O objeto A tem duas versões, Va1 e Va2; B tem a versão Vb1; e C tem duas versões, Vc1 e Vc2. Isto fornece quatro possibilidades de configurações ou contextos: (Va1 Vb1 Vc1), (Va1 Vb1 Vc2), (Va2 Vb1 Vc1) e (Va2 Vb1 Vc2).

Um mecanismo de notificação de mudança é definido para solucionar inconsistências na referência entre versões. A notificação é necessária quando o objeto referenciado é atualizado, removido ou uma nova versão é criada. O modelo define duas formas de notificação:

- Notificação baseada em sinais, onde cada versão de um objeto possui duas marcas de tempo: CN (registra a hora em que a versão é alterada) e CA (registra a hora em que a versão foi aprovada). Esta abordagem constitui uma estratégia Deferida, ou seja, o usuário apenas toma conhecimento da alteração quando consulta as marcas de tempo.
- Notificação baseada em mensagens, onde o sistema envia mensagens para notificar os usuários da versão afetada. Esta aboradagem pode ser Imediata ou Deferida, dependendo se os usuários recebem a mensagem imediatamente ou não.

O fato de que uma versão em geral referencia outras versões de uma forma recursiva representa o problema do Escopo da Notificação. Este modelo adota como estratégia notificar apenas as versões que referenciam diretamente a versão alterada.

O trabalho em questão especifica um conjunto de estruturas de dados que um sistema deve manter para suportar o modelo de versões descrito acima.

Uma das maiores contribuições do modelo consiste na exploração dos aspectos semânticos e operacionais do controle de versões incorporando a complexa configuração de objetos CAD, além disso, um aspecto importante é a exploração do problema de notificação dentro do contexto do modelo de dados apresentado.

2.6.2 RCS - Um Sistema para Controle de Versões

O RCS (Revision Control System) [45] é um sistema de controle de versões para gerenciar revisões (termo empregado como sinônimo de versões) de documentos texto, em particular programas fontes e documentação. Esse Sistema utiliza um conjunto de comandos do UNIX.

A principal função do RCS é gerenciar grupos de revisões. Revisões são organizadas em forma de árvore. A revisão inicial é a raiz da árvore e os ramos indicam de qual revisão uma outra revisão foi derivada. Além de gerenciar grupos de revisões o RCS fornece funções para compor configurações.

Uma revisão é criada quando um arquivo de trabalho é devolvido através de uma operação de devolução do RCS. Caso não exista o grupo de revisões, a revisão 1.1 é criada. Sucessivas operações de devolução acrescentam novas revisões ao grupo.

O RCS define uma configuração como um conjunto de revisões onde cada revisão vem de um grupo diferente. Para montar configurações o RCS usa uma ferramenta chamada *Make*, armazenando a identificação das revisões utilizadas em um arquivo chamado Makefile.

Um aspecto importante no sistema é o armazenamento compactado de versões. Para preservar espaço o RCS armazena revisões na forma de deltas, isto é, diferenças entre revisões sucessivas. A interface mantém esse fato oculto ao usuário. O RCS usa o programa *Diff* para o cálculo dos deltas. O *Diff* primeiro computa a maior cadeia de caracteres comuns as duas revisões e depois produz o delta a partir dessa cadeia. Usar deltas é uma estratégia clássica para redução de espaço, mas aumenta o tempo de acesso. Como a demora excessiva desencoraja o uso do sistema, o mecanismo de controle de versões deve fornecer um meio de reduzir o tempo de acesso. Para isto o RCS utiliza o seguinte mecanismo para formação de deltas: a revisão mais recente é armazenada intacta. Todas as outras revisões são armazenadas como deltas reversos. Um delta reverso descreve como voltar na hierarquia: ele produz a revisão desejada se aplicado a sucessora de tal revisão. Esta implementação tem a vantagem de que a extração da última revisão, que normalmente é a mais usada, é uma operação simples e fácil.

2.6.3 O Modelo de Dittrich

Em [16] os autores descrevem um modelo de versões para sistemas de banco de dados de engenharia onde versões podem ser organizadas logicamente através de um mecanismo de "cluster".

Um objeto de projeto é definido como um conjunto de versões. Uma versão é identificada por um número dentro do conjunto de versões de um dado objeto. O par (identificador-do-objeto, número-da-versão) identifica a versão no banco de dados. O número da versão é gerado automaticamente quando a versão é criada, são sequenciais e nunca reusados. Assim, o modelo acrescenta um aspecto semântico ao número da versão, ou seja, pode-se identificar qual a versão mais recente consultando o mais alto número do conjunto de versões de um objeto. Além disso, pode-se consultar quais versões foram removidas verificando quais os números que faltam no conjunto de versões.

Existem dois tipos de versões: versões congeladas e versões descongeladas. As versões congeladas são versões consistentes, compartilhadas por todos os usuários e não podem ser alteradas nem removidas até que sejam descongeladas. As versões descongeladas são privativas, podendo ser alteradas pelo proprietário.

O modelo define os conceitos de referência genérica e de ambiente. Uma referência genérica é uma referência para o objeto idependentemente da versão a ser usada. Um ambiente é uma relação binária representadas por um conjunto de pares (identificador-do-objeto, número-da-versão).

Um conjunto de operações é fornecido para definir e manipular versões de acordo com as descrições acima:

- criar e remover objetos de projeto
- criar e remover versões
- congelar e descongelar uma versão
- recuperar objetos de projeto e suas versões

- recuperar meta informação útil a respeito dos objetos de projeto e suas versões.

No modelo são apresentados alguns comentários sobre a implementação dos conceitos propostos. A implementação usa um sistema de banco de dados chamado XSQL em um ambiente IBM.

2.6.4 Neptune

O Neptune [13] é um hipertexto projetado como uma arquitetura em camadas, onde o nível principal é a máquina abstrata hipertexto (HAM). A HAM define operações para criação, modificação e acesso a nós e elos; fornece um mecanismo para gerenciamento de versões e acesso rápido a qualquer versão de um hipergrafo. Não há restrições quanto ao conteúdo dos nós. Não há interpretação no nível da HAM, pois nesse nível tem-se apenas dados binários.

Camadas adicionais são construídas no topo da HAM. Tipicamente, uma ou mais camadas de aplicação são construídas no topo da HAM e uma camada de interface com o usuário é construída no topo das camadas de aplicação. As camadas de aplicação consistem de programas que automaticamente manipulam ou transformam dados do hipertexto. Numa aplicação CAD, por exemplo, esta camada pode incluir ferramentas de "design" VLSI, compiladores de linguagem de alto nível ou processadores de texto. A camada de interface com o usuário pode fornecer uma interface de janelas para editar e navegar nos dados e para controlar os programas da camada de aplicação. A interpretação do conteúdo dos nós é de inteira responsabilidade da aplicação.

A HAM identifica nós e elos associando um par atributo/valor a eles. Por exemplo, ao atributo nome-do-nó é dado um valor como "módulo 1" para identificar o conteúdo como o código fonte do módulo 1. Informação é agrupada em configurações usando contextos, os quais são coleções de nós e elos.

O contexto não é útil apenas para agrupar nós e elos, mas também para suportar cooperação, árvores de versões e gerenciamento de configuração. Um contexto permite que se possa agrupar uma configuração de nós e elos. Contextos podem ser usados para definir uma área de trabalho e particionar um projeto em uma área de trabalho do projeto e em áreas de trabalho locais. Uma área de trabalho local permite que seja extraído um subconjunto de nós e elos da área de trabalho do projeto e que seja colocado na área de trabalho onde possam ser feitas modificações e teste.

A HAM fornece dois mecanismos que são importantes para a construção das camadas de aplicação. Primeiro, um número ilimitado de pares atributos/valor pode ser adicionado a um nó ou elo. Segundo, é oferecido um mecanismo de "demon" que invoca a aplicação ou código do usuário quando ocorre um específico evento da HAM como, por exemplo, a atualização em um determinado nó.

Na extremidade de um elo pode ser atribuido um deslocamento dentro do conteúdo de um nó. Se um nó contém texto, por exemplo, o deslocamento pode ser interpretado como uma posição dentro do texto.

O Neptune suporta dois mecanismos para associar elos entre versões de um nó: o elo pode referir-se a uma versão particular de um nó ou a versão corrente de um nó. O primeiro mecanismo é útil para construção de um gerenciador de configurações; o último pode ser utilizado como um mecanismo para atualização automática.

Como controle de versões é o tema central do Neptune, o armazenamento de versões utiliza o conceito de deltas similar ao utilizado no RCS. Novas versões dos nós são criadas a cada comando "SAVE" emitido pelo usuário.

2.6.5 O Modelo de Jarwa

Este trabalho [25] apresenta um modelo de dados orientado a objetos que suporta características de Hipertexto. Os objetivos do modelo são o gerenciamento e a recuperação de documentos produzidos e usados no ciclo de vida de desenvolvimento de software.

Documentos e programas são modelados como objetos compostos, e elos são suportados entre eles. Para localizar fragmentos de documentos relevantes à solicitação do usuário o modelo suporta recuperação de informação por conteúdo, "queries" como em banco de dados e navegação como em hipertexto. São abordados ainda aspectos referentes ao tratamento de versões.

No modelo proposto, as entidades usadas e produzidas durante o ciclo de vida do software são modeladas como objetos que correspondem aos nós de hipertexto. Cada objeto possui as seguintes informações: um valor, suas propriedades, seus descendentes na estrutura hierárquica e os objetos que fazem referência a ele. Objetos são criados de acordo com um esquema de tipos pré-definidos.

O conceito de **Tipo** é usado para especificar uma estrutura e comportamento comum a uma coleção de objetos. Em adição aos tipos convencionais (inteiro, real, string, etc), o modelo considera os **tipos básicos**: texto, gráfico, código-fonte, código-objeto e código-executável. Ainda define o tipo IndexTerms que representa conjuntos de termos indexados relacionados a fragmentos de conteúdo do documento, isto constitui um suporte para introdução da funcionalidade da recuperação de informação em sistemas de banco de dados.
Além dos tipos básicos, o modelo define os **Tipos construídos** que são obtidos aplicando os construtores de tipos (tupla, conjunto e lista) aos tipos básicos.

Os tipos são estruturados como uma hierarquia que define cada tipo como subtipo de outro, herdando suas propriedades. É pré-definido um supertipo global **Objeto** como a raiz da hierarquia, ele representa as características e comportamento compartilhado por todos os objetos.

A definição de um tipo inclui:

- nome do tipo
- nome do seu supertipo
- conjunto de atributos que descrevem os objetos deste tipo e modelam os elos existentes em hipertexto
- conjunto de métodos que podem ser aplicados a esses objetos.

O modelo define três tipos de atributos:

- atributos PROP: são valores dos tipos básicos ou construídos, não se referem a nenhum outro objeto. São as propriedades dos objetos.
- atributos COMP: são usados para declarar um objeto como parte de outro objeto. Isto captura a noção de **elos de composição** entre um objeto pai e seus componentes e permite ligar uma coleção de objetos para constituir um **objeto composto**. O valor do atributo COMP é um identificador de um objeto, e significa que tal objeto é componente do objeto pai. O modelo permite que um objeto seja componente de vários outros objetos. Esta possibilidade oferece suporte a objetos compostos como componentes compartilhados. A existência de um componente é dependente da existência de objetos que se refere a ele como componente, isto significa que um objeto componente só pode ser criado se ao menos um dos seus pais já existe. Além disso, se um objeto composto é removido, todos os seus componentes devem ser removidos, exceto aqueles que são referenciados como componentes de outros objetos.
- atributos REF: usados para fazer uma referência a outro objeto independente. Não possui nenhuma semântica de remoção associada a ele, isto é, a remoção de um objeto não causa a remoção automática dos objetos referenciados. Esses atributos são usados para modelar a referência entre objetos de software.

Desta forma, estes dois últimos atributos servem para modelar os elos de hipertexto, cada um com semântica diferente.

O modelo de versões introduz o conceito de Objeto Genérico para reunir versões de um objeto e modelar suas propriedades invariantes. Como são consideradas várias representações de um objeto genérico, várias árvores de versões devem existir para cada objeto genérico. O objeto genérico indica a versão corrente, que é especificada pelo usuário, ou na falta desta especificação, a mais recente é considerada como corrente. Como um objeto genérico reúne todas as suas versões, vários elos de composição devem existir entre um pai e objetos genéricos descendentes.

O modelo define um mecanismo de propagação de mudança. Tal mecanismo é similar ao mecanismo Evento-Ação. A cada objeto é associada um conjunto de regras que define como reagir a mudanças ocorridas em objetos que ele referencia. Quando uma regra

condição é verificada depois de mudanças em objetos referenciados, a regra ação é executada para manter a consistência.
Assim, regras associadas aos objetos são reunidas em um método de propagação definido no seu tipo. O método de propagação deve executar algumas ações nas versões afetadas pela mudança. Quando as ações não podem ser executadas por algum motivo, o método de propagação deve atualizar o Estado da Versão. O Estado da Versão é descrito usando duas listas: a lista de incompleteza e a lista de inconsistência. A lista de incompleteza indica componentes inexistentes ou deletados, e a lista de inconsistência indica versões referenciadas modificadas que podem causar a inconsistência.

Para resumir, o modelo proposto é um modelo semântico que suporta a combinação de semântica de dados orientada a valor fornecido pelos atributos PROP e semântica de dados orientada a objetos fornecida pelos atributos COMP e REF.

Em relação a implementação, o objetivo do projeto é usar os nós do NoteCards para armazenar os objetos. Isto irá permitir o uso do mecanismo de navegação do NoteCards e uso de outras facilidades para acessar a informação do banco de dados.

2.6.6 O Modelo de Osterbye

Este trabalho [35] discute questões referentes a controle de versões em sistemas hipertexto. Inicialmente são levantados os problemas pertinentes ao controle de versões em hipertexto e em seguida é mostrada a solução fornecida pelo modelo de Osterbye, chamado de HyperPro.

As questões de controle de versões em hipertexto são divididas em duas categorias:

- Questões Estruturais: relacionadas ao modelo de dados
- Questões Cognitivas: relacionadas a interface com o usuário.

As questões estruturais estão associadas aos seguintes aspectos:

- **Imutabilidade de Versões**: em hipertexto deve-se poder definir para as versões de nós os aspectos mutáveis e os aspectos imutáveis.
- **Versões de Elos**: deseja-se ter versões de elos ou apenas versões de nós?
- **Versões de Estrutura**: como deseja-se ter diferentes versões de uma única entidade, é desejável ter a noção de versões da estrutura do hipertexto. Similarmente, ser capaz de retornar ao estado prévio do hipertexto inteiro é tão desejável quanto retornar ao estado prévio de um único nó. Em sistemas com o modelo de dados que suporta composições, estas podem servir como base para versões de estruturas.

As questões cognitivas são as seguintes:

- **Criação de Versões**: em [12] Conklin aponta o problema de nomear nós. Se o usuário é forçado a dar nome a todo nó que é criado, sua atenção é distraída do verdadeiro propósito. Similarmente, se o usuário deve criar explicitamente novas versões dos nós e manter alguma consistência entre diferentes versões, ele terá a atenção desviada do trabalho a ser feito. Segundo Conklin, isto é chamado de *Esforço Cognitivo*. Uma maneira de evitar o esforço cognitivo na criação de versões é criar novas versões implicitamente. No Neptune novas versões são criadas a cada comando "SAVE" do editor. O problema nesta abordagem é que se cria um grande número de versões que não representa um passo consistente na evolução do hipertexto.
- **Seleção de Elemento**: quando se introduz versões de nós surge o problema de determinar para qual elemento, no grupo de versões, o elo aponta. Há as possibilidades de apontar para um elemento específico ou apontar para o elemento corrente, e ainda existe uma terceira proposta que consiste na seleção de elemento usando uma linguagem de consulta. Essa última opção aumenta o esforço cognitivo do usuário visto que ele tem de especificar um critério de seleção sempre que um elo é criado.

No modelo de dados do HyperPro as entidades são organizadas em uma hierarquia conforme mostra a figura 2.2. Os tipos existentes são os seguintes:

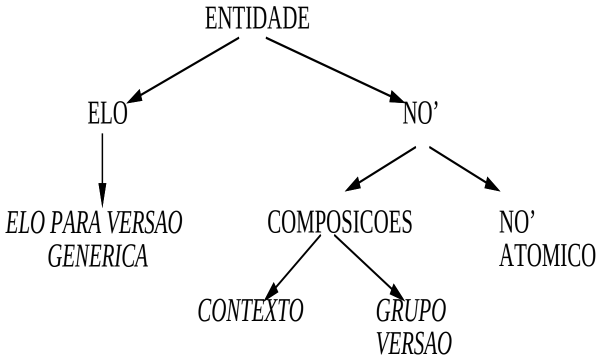

Figura 2.2: Modelo de Dados do HyperPro

Entidade: é a entidade superior, que fornece os atributos dos objetos. Toda a informação é armazenada como atributos. Entidades podem ser membros de **Composições** e devem ser membros de pelo menos um **contexto**. No modelo em questão o usuário pode especializar os tipos embutidos do sistema de acordo com as necessidades das suas aplicações. O tipo Entidade permite a definição da questão de imutabilidade para o tipo. Para uma entidade do tipo X, são especificados quais atributos podem ser mudados depois da entidade ser congelada; para nós são especificados quais tipos de elos podem ser atribuídos a um nó congelado.

Nó: tem os atributos Conteúdo e Nome. Podem derivar versões.

Elo: liga um nó origem a um nó destino. Não há mecanismo para versões de elos por dois motivos: primeiro porque versões de composições substitui versões de elos; segundo porque elos são mais usados para relações e, neste caso, é mais interessante ter elos individuais.

Composições: o conteúdo de um nó de composição é uma sequência de entidades onde repetições são permitidas. Esse tipo de nó serve como uma generalização do **GrupoVersão** e do **Contexto** e também como base para o usuário definir novos tipos. Podem derivar versões.

GrupoVersão: é um tipo especial de composições onde todas as entidades são consideradas versões da mesma entidade. As versões individuais são relacionadas através de elos e organizadas como em árvore (inspirada no RCS) de forma a manter a história do desenvolvimento. Não podem derivar versões.

Contexto: é uma composição cujo propósito é fornecer a noção de estrutura hierárquica. Todas as entidades são membros de no mínimo um contexto. O contexto mantém um critério de seleção a ser usado pelo **Elo para Versão Genérica**. O critério padrão seleciona a mais nova versão do contexto. Como contextos podem derivar versões, é possível ter versões de estruturas.

Elo para Versão Genérica: é uma especialização do elo que computa seu nó destino selecionando um nó do GrupoVersão que ele se refere, ou seja, é um elo entre um nó e um GrupoVersão.

Os tipos em itálico na figura 2.2 são aqueles relacionados com o suporte a versões.

No nível de interface, o modelo endereça os problemas cognitivos e procura fornecer uma interface onde o mínimo esforço seja dispendido para as questões de derivação de versões. Uma tarefa importante para o nível da interface é manter a noção de **contexto corrente**. O Elo para Versão Genérica usará o contexto corrente para o critério de busca. Novos nós e elos são criados como membros do contexto corrente. Quando um elo é criado, o nó destino é examinado e se tal nó for membro de um GrupoVersão, o elo será criado como genérico. O critério de seleção a ser usado pelo elo é o critério associado ao contexto corrente.

O problema de criação de versão é endereçado usando o contexto corrente. Enquanto é possível criar explicitamente novas versões de nós individuais, um comando **Nova Versão** está disponível no nível de contexto. O comando congela todos os membros do contexto e cria um sucessor do contexto corrente (contextos podem derivar versões) que torna-se o novo contexto corrente e que contém os mesmos elementos do contexto velho. A lista de membros do contexto velho é congelada, tornando possível retornar ao estado prévio da estrutura do hipertexto. Todas as mudanças irão acontecer agora no novo contexto.

2.6.7 O Modelo de Haake

Devido a necessidade de suportar versões em um ambiente de editoração hipermídia, esse trabalho desenvolveu um servidor hipermídia de versões chamado CoVer [20].

O ambiente de editoração é baseado em uma arquitetura em camadas:

- a camada de aplicação: consiste em tarefas de aplicações de editoração que compartilham um conjunto de hiperdocumentos contendo todos os documentos envolvidos no processo de editoração.
- a camada de gerenciamento de objetos e versões: que compreende o gerenciador de objetos (servidor hipermídia cooperativo: CHS) e o servidor de versões chamado CoVer, que é implementado no topo do CHS e é utilizado pela camada de aplicação.

O ambiente de editoração hipermídia consiste de várias aplicações necessitando suporte a versões. Para oferecer suporte a versões idependentemente da aplicação, os conceitos de versões são oferecidos pelo servidor hipermídia de versões, o CoVer. Esse servidor é implementado no topo do servidor cooperativo hipermídia: CHS. As aplicações no ambiente de editoração definem seus tipos de dados específicos - subclasses dos nós, elos e composições do CHS - na interface de aplicação do CHS.

O CHS oferece nós, elos e composições que podem ter atributos adicionados pelas aplicações. Objetos podem ser acessados pelos valores dos seus atributos usando a linguagem de consulta do sistema de banco de dados que está abaixo do CHS. O CHS mantém a história dos objetos. Ele armazena a hora de criação e o autor de cada nó, elo, composições e atributos e, em uma história de atualização, grava a hora e o autor de cada atualização nesses objetos. Porém, o CHS não possui a noção de versões e não preserva os estados anteriores dos objetos atualizados.

Para resolver o problema de versões, o CoVer propôs distinguir objetos de estado único (snobs), representando objetos não versão, de objetos de estado múltiplo (mobs), representando objetos versão. Cada objeto CHS, isto é, nó, elo ou composição, é considerado um snob e pode ser criado pelos comandos do CHS. Tais objetos podem ser transformados em um mob por uma operação explícita. Um objeto que possua versões é dito um objeto versionado.

Um mob é implementado no topo do CHS como uma composição suportando referências para todos os estados do objeto versionado que ele representa. Os estados de um objeto versionado são chamados versões e são representados por nós individuais, elos ou composições.

Referências para objetos versionados são feitas através de um mecanismo de endereçamento em dois passos. Uma referência para um objeto versionado consiste em dois valores, um identificador do objeto, isto é, uma referência para um mob ou snob, e no caso da referência para um mob, um identificador adicional da versão, uma referência para a versão.

Para preservar o estado dos objetos versionados, versões de nós, elos e composições podem ser congeladas. Assim, são distinguidos dois modos internos das versões: atualizáveis e congeladas. A operação de congelamento é uma operação explícita que salva o estado da versão. O estado de um nó atômico compreende seu conteúdo e atributos; o estado de um elo é definido por duas referências para outros objetos do hipertexto e os atributos do elo; e o estado de uma composição depende das referências para os objetos componentes, e dos atributos da composição. Assim, versões referenciadas por elos e composições também pertencem ao estado do elo ou da composição. O CoVer avalia referências e congela versões referenciadas implicitamente de forma a preservar estados de elos e composições.

O modelo implementa a história de derivação usando um elo específico (elo de derivação) referindo a versão ascendente e a versão descendente. Elos de derivação podem ser criados explicitamente pela aplicação. A criação de um elo de derivação automaticamente causa o congelamento do objeto predecessor. Além disso, CoVer oferece duas operações de derivação criando ou uma nova versão ou um novo objeto como cópia de uma versão e instalando o elo de derivação correspondente.

A seleção de versão é baseada na consulta e navegação em versões com respeito a valores dos seus atributos ou relacionamentos com outros objetos. A aplicação pode explorar o desenvolvimento histórico através da data de criação das versões, restringir a visão de versões que foram criadas por um autor específico, ou usar qualquer outro atributo que sugere uma visão sobre o conjunto de versões.

De forma a endereçar os problemas de criação e seleção de versões em hipertexto, dois conceitos adicionais foram introduzidos pelo CoVer: Tarefas e Anotações.

Um conceito chave para a criação de versão é a **Tarefa**. Usuários mudam seu hipertexto para realizar uma tarefa e essas tarefas podem guiar a criação automática de versões. Enquanto mobs mantêm a história de um único objeto versão, as tarefas mantêm as versões de vários objetos usados e criados no contexto de realização de um trabalho. Para criar uma nova versão de um objeto específico, tarefas podem aproveitar todas as versões disponíveis de todos os objetos do sistema. Além disso, qualquer objeto pode ser incluído em uma tarefa através de uma operação fornecida pelo CoVer. Mudanças em objetos incluidos são transformadas em versões derivadas, isto é, o CoVer congela os objetos incluidos, deriva novas versões desses objetos e executa as mudanças nessas versões derivadas. Uma tarefa pode conter várias subtarefas. Uma tarefa é implementada no topo do CHS como uma composição suportando referências aos objetos determinando seu estado corrente. Além de um identificador mantido pelo sistema, uma tarefa é descrita pelo nome adicionado pela aplicação; data e hora de início e finalização - ou seja, se a data de finalização está indefinida, a tarefa está aberta - informação do autor e, opcionalmente, atributos definidos pela aplicação. Uma tarefa de alto nível é mantida pelo CoVer para registrar todas as tarefas criadas pelas aplicações. O CoVer suporta o acesso a tarefas através de consultas nos seus atributos bem como navegação no histórico de tarefas. Esse conceito de Tarefa introduzido no modelo permite o acesso a versões em diferentes níveis de abstração e detalhe.

O modelo apresenta o conceito de **Anotação** que consiste de um comentário no conteúdo e suporta referências para as versões anotadas e referências para as versões que são respostas para a declaração contida na anotação.

2.7 Aspectos Abordados no MCA

Como não existe um consenso sobre um modelo de dados com controle de versões, o modelo de versões que é objeto desse trabalho procura fornecer soluções para os problemas relacionados anteriormente, captando alguns conceitos apresentados em diversos trabalhos analisados e, em alguns pontos, apresentando seu enfoque particular. Muitas características incorporadas ao Modelo de Contextos Aninhados foram baseadas no Modelo de Chou [9] e posteriormente foi constatada uma forte semelhança com os modelos de Osterbye [35] e de Haake [20], cujas novas idéias determinaram o adicionamento de algumas funções.

A estrutura central do MCA é o conceito de nós de contexto, uma extensão ao contexto do Neptune, utilizado para agrupar coleções de nós e elos. Adicionalmente a proposta do Neptune, o MCA introduziu a facilidade de aninhamento dos contextos em vários níveis. Existe uma identificação entre os conceitos de contexto e de configuração, tendo em vista que, da mesma forma que um contexto, uma configuração compartilha seus componentes com outras configurações, mas, as ligações entre esses componentes é particular de cada configuração. O modelo de Osterbye também absorveu do Neptune a noção de contexto e, similarmente ao MCA, adicionou a possibilidade de contextos agruparem contextos. Tal semelhança entre as estruturas nos dois modelos deu origem a outros pontos em comum tanto a nível estrutural quanto funcional. O aninhamento de nós de contextos generaliza as funcionalidades dos *webs* do Intermedia [33].

A inclusão do controle de versões no MCA determinou a necessidade de uma estrutura, semelhante ao Objeto Genérico presente em grande parte dos modelos de versões, com o propósito de agrupar todas as versões de um mesmo objeto. Como os contextos servem para agrupar um conjunto de nós, o modelo estendeu esse conceito, introduzindo o contexto de versões para agrupar um conjunto de nós versões.
Em alguns sistemas o objeto genérico possui existência independente de suas versões, isto significa que o mesmo pode ser criado sem ter ainda uma versão associada, ou até continuar existindo mesmo depois que suas versões associadas forem removidas. No MCA, o contexto de versões tem existência dependente de suas versões, ou seja, se todas as versões associadas ao contexto de versões forem removidas, o contexto de versões também será.

O tipo GrupoVersão do Modelo de Osterbye desempenha a mesma função do contexto de versões e as mobs do Haake.

Objetos compostos podem ser construídos incluindo referências a objetos. Quanto a questão da semântica de referências entre objetos, conforme o modelo de Chou [9], o MCA possibilita que haja referências para uma versão específica do objeto ou uma referência genérica para o contexto de versões. Neste último caso o sistema resolve tal referência dinamicamente, retornando a versão corrente, que não é necessariamente a última versão a ser criada, visto que existe uma operação disponível para que o usuário estabeleça a versão corrente. O sistema garante a integridade referencial, isto é, um objeto não pode ser excluído se outros objetos fazem referência a ele. As noções de referência dinâmica e versão corrente são encontradas também no Modelo de Osterbye onde são chamadas respectivamente de Elo para Versão Genérica e Critério de Seleção.

Assim como o Intermedia e o Neptune, o MCA permite a definição de pontos de ancoragem dentro dos nós origem e destino. Âncoras no MCA são atributos dos nós. A vantagem desta abordagem em relação aquelas onde as âncoras são atibutos dos elos, a HAM por exemplo, é que ocorrendo mudanças nos nós, os elos não precisam ser mudados.

Para solucionar o problema do compartilhamento de versões são adotados os conceitos de Estado do nó e de Bases de Documentos.
O Estado representa uma classificação para as versões de acordo com seu estágio de desenvolvimento. Assim, conforme o estado do nó, são habilitadas ou não as operações de criação de versão, alteração e/ou remoção sobre ele. O conceito de Estado está presente na proposta de Chou onde é denominado Tipo de Versões, no modelo de Dittrich onde existem versões congeladas e descongeladas representando possíveis estados de uma versão e no trabalho de Jarwa onde também existe tal conceito abordado de forma diferente do MCA. No CoVer e no Modelo de Osterbye existem noções semelhantes ao conceito de estado.
O conceito de base de documentos foi absorvido do modelo de Chou. No Neptune, o conceito de contextos para suportar cooperação que faz uma subdivisão em áreas de trabalho com diferentes operações possíveis é uma abordagem semelhante a subdivisão em bases de documentos. Funcionalmente, a noção de Tarefa introduzida no CoVer [20] é similar ao conceito de Base Privada do MCA.

O conceito de Anotações foi adotado pelo MCA baseado no CoVer. No MCA Anotações existem apenas na base privada e não acarretam a criação de novas versões. No CoVer, Anotações estão relacionadas com Tarefas, pois ele registra qual tarefa produziu a Anotação.

No MCA, a criação de versões pode ser realizada de duas formas: ou através da utilização explícita de uma operação de criação de versão de um determinado nó; ou implicitamente através das bases privadas. Como em [20], bases privadas guiam a criação automática de versões. Sempre que um nó Permanente é alterado através de uma base privada, uma versão Temporária deste nó é automaticamente criada na mesma base privada, usando a operação de *Check-in*, a ser descrita no próximo capítulo. Determinados sistemas associam a criação de versões a intervalos regulares de tempo. O MCA considera que a criação de versões desta maneira trata-se de uma abordagem ineficiente tendo em vista que é criado um grande número de versões, o que não representa um passo consistente na evolução do hipertexto [35].

Visando permitir Navegação através de Trilhas, o MCA absorveu do Intermedia o conceito de Trilhas, definindo-a como um tipo especial de nó de composição, o que possibilita o aninhamento de trilhas.

Baseado no modelo de Katz [28], o MCA propõe uma estratégia para resolução do problema da propagação. Diferentes soluções para esse problema também foram apresentadas nos trabalhos de Jarwa e de Banerjee [3].

Com relação a Notificação é apresentada uma solução que baseia-se na proposta de Chou, levando em conta as particularidades das aplicações multimídia de natureza distintas e a estrutura de aninhamento característica do modelo.

Todos esses pontos serão discutidos detalhadamente nos próximos capítulos.

Capítulo 3

Versões no MCA

3.1 Introdução

Neste capítulo será apresentado o Modelo Básico de Contextos Aninhados e as extensões incorporadas ao modelo para dar suporte ao tratamento de versões.

Na próxima seção é descrita a estrutura proposta pelo modelo básico.

Em seguida, na seção 3.3 são levantadas questões referentes ao modelo de apresentação que trata dos detalhes necessários para exibição das informações.

Na seção 3.4, com a apresentação da nova hierarquia de Classes do Modelo de Contextos Aninhados, inicia-se a discussão dos conceitos introduzidos pelo modelo de versões. O Sistema para tratamento de versões proposto para o MCA é adequado a qualquer sistema hipermídia com nós de composição com aninhamento, em particular àqueles baseados na proposta de padrão MHEG, como é o caso do MCA [42]. Na subseção 3.4.1 apresenta-se a estrutura encarregada de agrupar as versões de um nó. A discussão prossegue na subseção 3.4.2 onde serão abordados os aspectos referentes à interface do nó.

A seção 3.5 mostra as soluções oferecidas pelo modelo para permitir trabalho cooperativo garantindo a consistência das informações. Os conceitos de Estado, Bases de Documentos e Anotações são discutidos.

Na seção 3.6 é debatida a questão da Propagação de versões e apresentada a estratégia adotada pelo MCA para solucionar tal problema.

Finalmente, na seção 3.7 é apresentada uma breve descrição da arquitetura aberta baseada em camadas sobre a qual o MCA foi projetado.

3.2 O Modelo de Contextos Aninhados

No modelo de contextos aninhados os documentos são formados por unidades de informação chamadas nós que são conectados por elos.

O modelo define dois tipos básicos de nós: *nó terminal e nó de composição.* A hierarquia de classes proposta pelo modelo está ilustrada na Figura 3.1

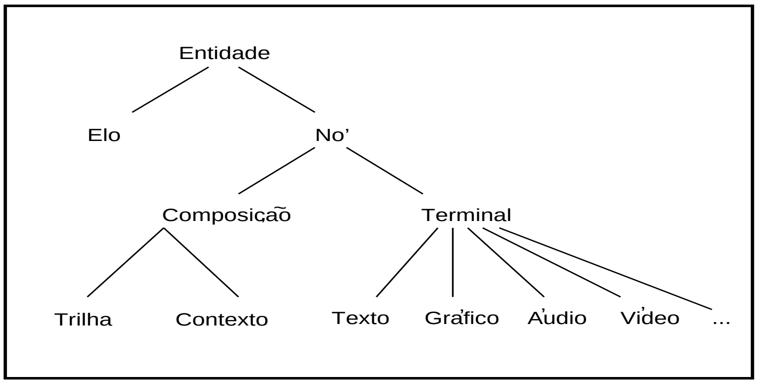

Figura 3.1: Hierarquia de Classes do MCA

Uma *Entidade* permite que pares atributo/valor sejam atribuídos a todos os objetos. Cada entidade tem um identificador único (UID) e uma lista de controle de acesso (ACL). Cada entrada na ACL associa um usuário ou grupo de usuários a esse direito de acesso para cada atributo. Questões como: " Todos os usuários podem criar elos?" "Qualquer usuário pode deletar um elo?" " Qualquer usuário pode editar um nó?" entre outras, podem ser respondidas pelo mecanismo de direitos de acesso. Cada entidade também possui uma *Entidade de Especificação de Apresentação*. Tal entidade contém informação para o modelo de apresentação sobre como a entidade deve ser apresentada ao usuário, portanto, não é usada pelo modelo conceitual.

Nós terminais são aqueles que armazenam dados de qualquer natureza que possa ser representada no computador: texto, gráfico, áudio, etc. Eles podem ser especializados em outras classes de acordo com as necessidades das aplicações.

Um *Nó de Composição* agrupa entidades chamadas *componentes*, podendo, inclusive, incluir nós de composição. Esses componentes não necessariamente formam conjuntos, pois uma entidade pode ser incluida mais que uma vez em um nó de composição. A classe de nós de composição podem ser especializadas em duas outras classes, *nós contexto* e *nós trilhas*.

O *nó contexto* é utilizado para agrupar um conjunto de nós conectados por elos. Cada nó desse conjunto pode ser um nó terminal ou mesmo um nó contexto. Essa recursividade na definição permite que os contextos possam ser aninhados em vários níveis. Além disso, o mesmo nó pode estar contido em mais de um contexto, assim, toda mudança efetuada em um nó será visível por todos os contextos que o contém. Desta forma, o nó contexto permite a implementação das abstrações de agregação, hierarquia e visão, tornando possível que documentos sejam organizados hierarquicamente ou não, e que sejam definidas visões diferentes do mesmo documento. Então, o contexto ajuda a diminuir o problema chamado "Perdido no Hiperespaço" [21]. O conteúdo de um nó contexto é um par (S,L), onde S é um conjunto de nós terminais ou de contexto e L é um conjunto de elos, tais que suas extremidades estão contidas em S. Os elos não são compartilhados pelos contextos, ou seja, cada contexto deve definir seu próprio conjunto de elos. Assim, um elo pertence a um contexto que contém os nós das suas extremidades.

Uma *Trilha* é uma especialização de um nó de composição que contém uma lista ordenada de nós, incluindo trilhas. Um nó pode estar na lista em mais de uma posição. Uma trilha pode representar caminhos dentro de um específico contexto, neste caso, diz-se que a trilha está associada ao contexto.

Todo nó contém dois atributos especiais, CONTEÚDO e MÁSCARA. O valor de CON-

TEÚDO depende da classe do nó (texto, áudio, vídeo, etc.). A MÁSCARA define os "região de ancoragem" dentro do nó, ou seja, as entradas que podem ser utilizadas por elos com extremidade no nó, e essas entradas são chamadas de *Âncoras*. Assim, nenhum nó fará uma referência direta a regiões dentro do conteúdo de um determinado nó, mas fará uma referência indireta através de uma das âncoras. Esta definição de âncora permite que mudanças no conteúdo de um nó seja transparente aos elos que tocam o nó. Considerando, por exemplo, um elo com ponto destino definido no segundo parágrafo de um texto. Se esse ponto final for definido como um deslocamento em bytes, a eliminação do primeiro parágrafo do texto irá causar um erro. Usando âncoras, o ponto destino identifica a informação desejada, sem permitir interferência quando os textos adjacentes forem alterados.

Uma âncora tem associada um *identificador* que serve para identificá-la dentro do conjunto de âncoras de um certo nó e um *valor*. Diferentes classes de âncoras podem ser definidas para diferentes classes de nós. O sistema apenas compreende a representação interna da âncora no caso de nós contexto. Toda classe de âncoras tem um valor especial λ que representa como ponto de ancoragem todo o nó no qual a âncora é definida. No caso do nó contexto **C**, o valor da âncora pode ser:

- λ (isto significa que o ponto de ancoragem é todo o nó C); ou
- um subconjunto de nós contidos em C; ou
- a lista de nós (N_k, ..., N_2, N_1) e uma âncora que deve pertencer a máscara de N_1. Neste caso, o nó N_{i+1} deve ser um nó de composição, N_i deve estar contido em N_{i+1}, para todo i $\in$ [1,k) e N_k deve estar contido em C.

A máscara de um nó contém os identificadores das âncoras deste nó.

Um elo é a entidade usada para conectar nós. No MCA os elos são definidos dentro de um contexto. Eles podem ancorar no nó como um todo ou em alguma região situada no conteúdo do nó. Essa noção de região é descrita no MCA pelo conceito de Âncora, visto acima. Um elo pode possuir múltiplos destinos, o que permite a definição de conexões um-para-muitos, que são importantes para aplicações onde a seleção de um elo deva determinar

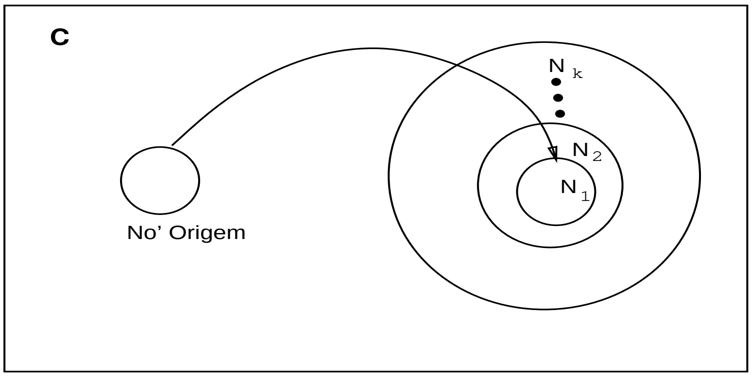

Figura 3.2: Ponto destino de um elo

a apresentação de vários nós. Cada um dos pontos destinos de um elo é definido por uma lista de nós (N_k, ..., N_2, N_1) e uma âncora, que deve pertencer a máscara de N_1. O nó N_{i+1} deve ser um nó de composição e N_i deve estar contido em N_{i+1}, para todo i $\in$ [1,k). O nó N_k é chamado a *base* do elo. Elos são sempre direcionais, apesar de poderem ser seguidos em qualquer direção.

No modelo básico, é definido o ponto final de um elo contido no contexto C, e similarmente, um possível valor de âncora do contexto C, como sendo um par consistindo de uma lista de nós (N_k, ..., N_2, N_1) e uma âncora α, tal que:

- α pertence a máscara de N_1
- para todo i $\in$ [1,k), o nó N_{i+1} é um contexto, N_i deve estar contido em N_{i+1}, e N_k deve estar contido em C.

A figura 3.2 ilustra esta definição do ponto destino de um elo.

Os elos no MCA, assim como no MHEG, possuem **condições** e **ações** associadas a ele. As ações são processadas quando as condições são satisfeitas. Uma **condição disparadora** é associada para a avaliação dos valores dos atributos do ponto de origem do elo. Ações definem relações espaço-temporais entre o ponto origem e os pontos destino de um elo.

Como já foi dito anteriormente, um elo pertence a um contexto que contém os nós das

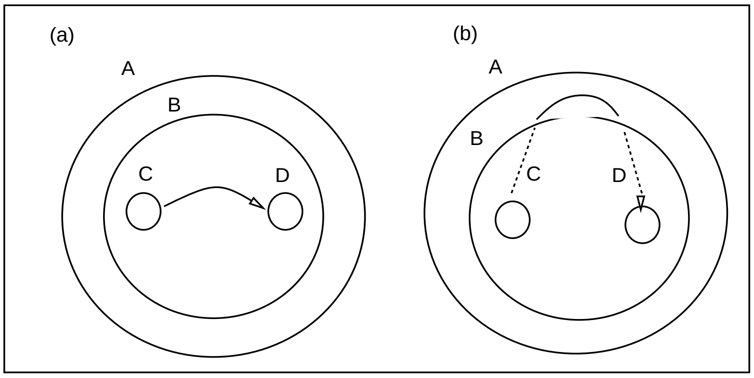

Figura 3.3: Definição de um elo entre dois nós

suas extremidades. Por exemplo, a Figura 3.3 mostra um contexto A que contém outro contexto B que por sua vez contém os nós C e D. Como B contém C e D, um elo entre estes dois nós pode ser definido de duas formas:

- Pode ser definido em B. Como mostra a Figura 3.3(a), o elo seria definido como ((C,i),(D,j)), onde i e j são as âncoras válidas para C e para D, respectivamente. Assim, qualquer contexto no qual B esteja contido, este elo seria visto.
- Outra opção possível, ilustrada na Figura 3.3(b), seria definir o elo para ser visto através do contexto A, ou seja, defini-lo em A como ((B,m),(B,n)), sendo m e n as âncoras especificadas respectivamente como (C,i) e (D,j).

Define-se no modelo a hiperbase como sendo o repositório dos nós. Assim, a hiperbase consiste em um conjunto H de nós, tais que, para qualquer nó $N \in H$, se N é um nó de composição, então todos os nós contidos em N também pertencem a hiperbase H.

3.3 O Modelo de Apresentação

No modelo conceitual, descrito na seção anterior, os dados hipermídia são tratados como estruturas de dados passivas. Porém, um sistema hipermídia deve oferecer a possibilidade do usuário acessar, manipular e navegar através da rede hipermídia. O modelo de apresentação encarrega-se de adicionar às entidades que serão exibidas, informações necessárias para sua apresentação.

Como diferentes contextos podem conter o mesmo nó e contextos podem ser aninhados em qualquer profundidade, é necessária uma forma de identificar através de qual sequência de contextos aninhados um dado nó está sendo observado e quais os elos que atingem o nó neste aninhamento, isto é capturado pelo conceito de *perspectiva* e de *elos visíveis por um nó* através de uma perspectiva.

Uma *perspectiva* para um nó N é a sequência P=(N_1,..,N_m), com m$\geq$ 1 tal que N_1 = N e N_i está contido em N_{i+1}, para i $\in$ [1,m). Como N é implicitamente dado por P, P pode ser chamado de *Perspectiva*. Um nó M está presente em P=(N_1,..,N_m) se e somente se M=N_i, para algum i $\in$ [1,m). Assim, podem haver perspectivas diferentes para um mesmo nó N, se este nó estiver em mais de um contexto. A *perspectiva corrente* é a perspectiva usada para alcançar um nó sobre o qual se está trabalhando em um dado momento.

Elos possuem um conjunto de especificações de apresentação do nó, como no modelo de Dexter [22], que contém informação para o modelo de apresentação indicando como um nó referenciado deve ser apresentado para o usuário.

A apresentação de um componente para o usuário é feita através de sua instanciação que consiste em uma função que retorna um objeto de apresentação dado como entrada o componente e sua especificação de apresentação. No MCA sessões de trabalho são modeladas por *bases privadas*, que será discutido adiante, e uma instanciação é a criação de uma versão na base privada. Especificações de apresentação contém informação para a apresentação de uma entidade. Elas definem métodos para exibição ou edição de entidades. As especificações de apresentação podem ser armazenadas como um atributo das entidades, ou até como um tipo de nó terminal. Ao apresentar um nó, a especificação de apresentação definida explicitamente pelo usuário desvia da especificação definida pelo nó contexto que contém o nó; esta, por sua vez, desvia da especificação de apresentação definida pelo elo usado para alcançar o nó, que desvia da especificação de apresentação definida dentro do nó. Cada tipo de nó tem uma especificação de apresentação padrão, usada se nenhuma outra é definida.

Se N_1 é um nó e $P=(N_1,..,N_m)$ sua perspectiva, um elo e é visível de P por N_1 cuja âncora tem identificador i_1 se e somente se:

- e está em N_2 e (N_1,i_1) é uma das extremidades finais de e; ou
- e é visível de P por N_2 cuja âncora tem identificador i_2, e o valor desta âncora é o par (N_1, i_1).

Um elo e é visível de P por N_1, se e só se:

- N_1 está no caminho de uma das extremidades de e, e e está em N_2, ou N_3, ... ou N_m; ou
- e é visível de P por N_1 com algum identificador de âncora.

O HyperProp oferece suporte a alguns mecanismos de navegação e, além disso, qualquer aplicação hipermídia que necessite de formas de navegação específicas , pode programá-la usando as primitivas de navegação oferecidas pelo HyperProp.

As Formas de navegação oferecidas são:

- Navegação em Profundidade: é a ação de seguir o aninhamento dos nós de composição. Isto permite ao usuário mover-se para frente e para atrás na composição hierárquica.
- Navegação através de Elos: a ação de percorrer os elos.
- Navegação através de Queries: o usuário pode atingir um determinado nó descrevendo as propriedades que este nó satisfaz.
- Browsers para Contextos: mostra uma visão pictorial do hiperdocumento. Navegação em Browsers é outra forma de navegação pré-definida no MCA.
- Navegação através de trilhas: permite usuários seguirem trajetórias estabelecidas em sessões prévias ou definidas como padrão no próprio documento. O sistema

fornece um mecanismo *Trilha de Base Privada* para guardar os caminhos seguidos durante uma sessão. Note que isto pode ser uma maneira de se criar uma trilha.

Numa navegação através de trilhas, o usuário tem disponível os comandos:

- próximo: automaticamente vai para o próximo item da lista ordenada de componentes.
- prévio: fornece o componente anterior da lista.
- Início: fornece o primeiro componente da lista.

3.4 Estruturas para Controle de Versões

O modelo de contextos aninhados é suficientemente flexível para dar suporte ao tratamento de versões. A estrutura de contextos, ponto central do modelo, constitui um esquema que sujeito à pequenas adaptações poderá desempenhar eficientemente a função de agregar o grafo de história de versões. Além disso, um modelo com nós de composição aninhados como o MCA, fornece suporte imediato para algumas das finalidades do uso de versões vistas na seção 2.3. Por exemplo, para registrar diferentes alternativas de um objeto basta apenas criar diferentes contextos para corresponder as alternativas distintas. Desta forma, os nós de contexto representam um mecanismo para a definição de visões diferentes de um mesmo documento.

Para incluir controle de versões no MCA, a hierarquia de classes foi estendida conforme mostra a figura 3.4.

Tratamento de versões de elos não é suportada pelo modelo por acreditar-se que esse aspecto adicionaria mais complexidade que funcionalidade ao sistema.

O nó contexto é especializado originando cinco novas classes: *Anotação, Hiperbase Pública, Base Privada, Contexto de Versões e Contexto do Usuário.*

O *Contexto do Usuário* deve ser usado da mesma maneira que nós contexto são usados no modelo básico, como uma facilidade para visões, aninhamentos e hierarquia. A diferença é

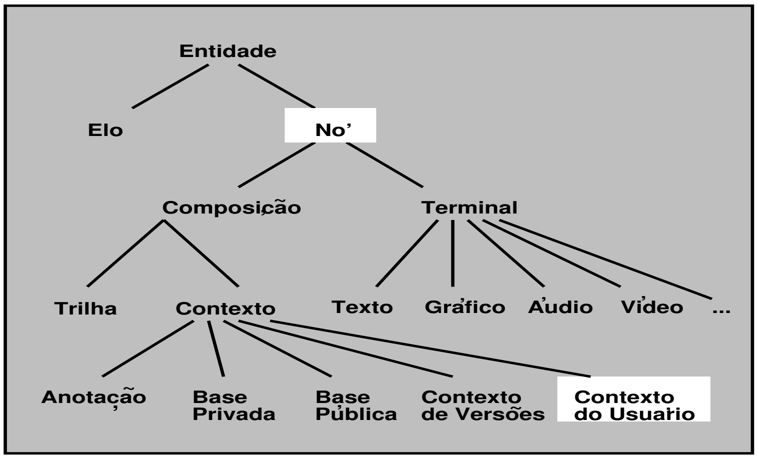

Figura 3.4: Hierarquia de Classes do MCA incluindo Estruturas para Versões

que um contexto do usuário pode conter apenas nós terminais e nós contexto do usuário, enquanto que nós contexto podiam conter qualquer tipo de nó contexto, nó terminal. Agora, o nó contexto passa a ser uma classe abstrata, de forma que, nenhuma instância dele pode ser definida.

Na hierarquia de classes do MCA, figura 3.4, os itens realçados, nós terminais e nós contexto do usuário, são aqueles sujeitos à derivação de versões. Cada atributo dessas duas classes, inclusive o atributo conteúdo, pode ser especificado como *versionável* ou *não-versionável*. No caso do atributo versionável, alterações no seu valor deve causar a criação de uma nova versão do objeto, dependendo do estado do nó. Atributos não-versionáveis podem ter seu valor modificado sem criar uma nova versão do objeto. Essa abordagem satisfaz o requisito da imutabilidade de versões, visto no capítulo anterior. Como o modelo em questão pretende dar suporte ao trabalho cooperativo, é necessário algum tipo de mecanismo de notificação que será particularmente útil no caso de atualização concorrente dos atributos não-versionáveis.

No MCA, o usuário pode especificar a adição de novos atributos sem criar uma nova versão do objeto. Essa facilidade é especialmente importante quando da incorporação de uma nova ferramenta no sistema que, utilizando esse artefato poderá então, armazenar informações nos nós já criados.

3.4.1 Contexto de Versões

O nó contexto de versões contém um conjunto de nós terminais ou nós contexto do usuário, que representam as versões de um nó, denominadas de versões correlatas. As versões correlatas podem não ser da mesma classe, por exemplo, em um contexto de versões pode existir dois nós que descrevem o mesmo texto, sendo que um consiste na versão escrita (classe texto) e o outro na versão falada (classe áudio).

Versões correlatas são ligadas formando um grafo direcionado. O grafo do contexto de versões representa a relação de derivação entre os nós, ou seja, o histórico da criação de versões. Quando ocorre a criação de uma nova versão M de um nó N, é criado um contexto de versões que irá conter os nós M e N, e um elo entre eles. Diz-se que v_2 *foi derivado de* v_1, se há um elo da forma ((v_1,i_1), (v_2,i_2)) no contexto de versões V. As âncoras indicam que parte de v_1 gera que parte de v_2. No contexto de versões, as operações de inserção e remoção de elos geralmente são realizadas pelo sistema, pois a semântica dos elos no contexto de versões consiste em registrar as relações ascentral/descendente entre versões, e essa relação é definida automaticamente quando é criada uma versão de um nó. Assim, no contexto de versões, elos são usados para modelar a estrutura sintática da derivação de versões, expressando sua hierarquia. Assim, para indicar que um nó B foi derivado de um nó A, é adicionado no contexto de versões um elo na forma ((A,i),(B,j)). As âncoras **i** e **j** indicam a parte de A que gera determinada parte de B. Os nós são adicionados automaticamente no contexto de versões quando é criada uma versão através de uma operação explícita de criação de versões. O único caso onde o usuário deve fazer uma inserção explícita de um nó em um contexto de versões é quando desejar criar versões correlatas de classes diferentes. Por exemplo, criar idependentemente os nós A e B, descrevendo a mesma informação, só que A é uma versão escrita e B uma versão falada. O sistema, neste caso, não tem como deduzir que A e B são versões da mesma informação, por isso, o usuário tem de se encarregar desta parte.

O contexto de versões V deve oferecer uma forma de registrar a sua versão corrente. Uma dessas maneiras é reservar uma entrada na máscara de V, isto é, uma âncora para manter a referência para a versão corrente. Outras âncoras podem especificar outras versões seguindo outros critérios de escolha. Assim, é possível definir uma referência de um nó N para um contexto de versões M. Tal referência, denominada *Referência Dinâmica*, permite que a versão recuperada (a versão corrente) seja estabelecida quando é feita a navegação. Para se criar uma referência dinâmica de um nó para um contexto de versões, deve-se incluir o contexto de versões no contexto onde o elo será definido. Esta característica do modelo permite a facilidade de atualização automática de referências.

Outra opção para registrar a versão corrente é possível no caso do modelo incluir entidades virtuais baseadas em uma linguagem de consulta. Neste caso, a referência pode ser feita através de uma consulta. Esta consulta não precisa fazer parte da âncora do contexto de versões pois ela pode ser definida no elo. Uma consulta que define a versão corrente pode retornar várias versões, que serão interpretadas como alternativas e apresentadas em um contexto do usuário.

Portanto, no contexto de versões, o agrupamento tem uma semântica, visto que, o fato de dois nós estarem em um mesmo contexto de versões indica que eles representam a mesma informação, em algum nível de abstração, sem implicar que um seja direta ou indiretamente derivado do outro [42].

Um dos requisitos de um modelo de versões para sistemas hipermídia, citado no capítulo anterior, é permitir a criação de versões da estrutura do documento. Isto é possível no MCA através da criação de versões do nó contexto do usuário.

3.4.2 A Interface do Nó

No modelo de versões, o conceito de âncoras introduzido para fixar cada deslocamento nas extremidades dos elos que atingem os nós, assemelha-se ao conceito de interface descrito em [9]: "versões são objetos que compartilham a mesma interface, mas possuem implementações diferentes". A interface do nó é a máscara que define as possíveis entradas que podem ser utilizadas pelos elos com extremidades nesse nó. A implementação é o conteúdo do nó.

Desta forma, com o intuito de diminuir o risco de inconsistência de versões o modelo impõe como restrição que não pode haver alteração na máscara de um nó versão nem na máscara de um nó que deu origem a versões. A liberdade de alteração das âncoras em

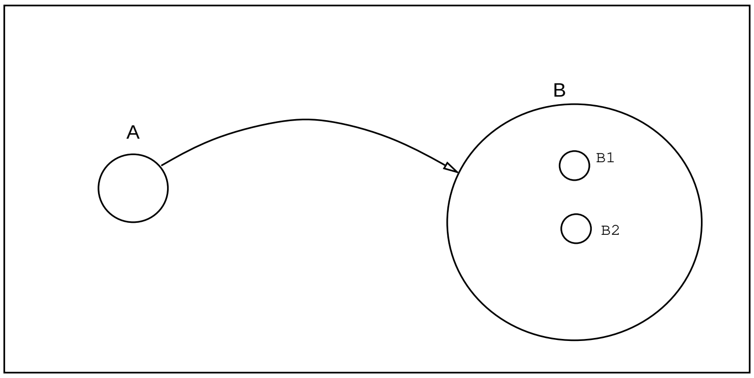

Figura 3.5: Elo entre um nó e o contexto de versões

um nó versão acarretaria problemas em relação aos elos que chegariam em um contexto de versões. Por exemplo, considerando um elo entre um nó A e um contexto de versões B (Referência Dinâmica) que contém B_1 e B_2 (Figura 3.5), ao navegar por esse elo, o sistema deve retornar a versão corrente no contexto B. O destino do elo é o identificador da âncora onde o elo está ancorado. Supondo que B_1 contém a âncora com identificador 1 e 2; e B_2 contém apenas a âncora com identificador 1 e considerando que a âncora destino do elo registre o deslocamento 2, se a versão corrente no contexto B for a versão B_1 não há problemas quando o elo for percorrido, porém, caso a versão corrente no contexto de versões B seja B_2, o elo fica sem um ponto de ancoragem no destino já que B_2 não contém a âncora com identificador 2.

Para evitar tal problema, optou-se por não permitir a alteração da máscara de um nó versão. Contudo, isto não constitui uma restrição muito forte visto que o autor de cada nó pode mudar o conteúdo de suas âncoras, preenchendo-os com os deslocamentos correspondentes em sua versão, inclusive com um deslocamento nulo. Com isso, apesar das entradas em um nó serem as mesmas para todas as versões correlatas, os deslocamentos podem ser diferentes em cada versão, tendo o autor total liberdade para mudá-lo em qualquer instante. É importante ressaltar que o modelo impõe apenas que o número de âncoras seja o mesmo para todas as versões de um nó, não importando se os deslocamentos registrados em cada uma delas seja diferente. Como o sistema não tem como identificar a semântica de um nó, cabe ao usuário manter a consistência quando preencher os deslocamentos em sua versão.

3.5 Suporte a Trabalho Cooperativo

Para fornecer suporte a trabalho cooperativo mantendo a consistência de dados e interconexões entre eles, conceitos adicionais foram introduzidos no MCA.

3.5.1 Estado

O conceito de *Estado do nó* terminal e nó contexto do usuário foi introduzido para controlar quais operações podem ser aplicadas sobre uma versão. O Estado do nó garante que apenas versões consistentes sejam compartilhadas entre usuários. O nó terminal ou contexto do usuário contém um atributo *Estado* que pode conter um dos seguintes valores: *Permanente, Temporário ou Obsoleto.* Quando um nó é criado, ele é marcado Temporário que representa um estado incompleto, e assim permanece até que ocorra explicitamente uma mudança de estado. Uma vez que um nó Temporário representa um estado incompleto, seu conteúdo e atributos podem ser livremente alterados sem que tais alterações dêem origem a uma nova versão.

O conceito de Estado do nó é relevante apenas para os nós terminais e contexto do usuário pois estes possuem atributos sujeitos à derivação de versões. No MCA o usuário pode especificar se a adição de novos atributos a um nó Permanente gera a criação de versões ou não.

Quando um nó torna-se estável, deve ser marcado Permanente.

Quando se deseja eliminar um nó Permanente, deve-se torná-lo Obsoleto. Assim, os nós que o referenciam não se tornam inconsistentes. Esses nós devem ser notificados que estão referenciando algum nó marcado Obsoleto, e então o usuário decide se mantém ou não a referência. O Sistema deve fornecer uma "coleta de lixo" para eliminar nós marcados obsoletos e não referenciados por outros nós. Caso o nó eliminado esteja em algum contexto de versões, correções devem ser feitas nesse contexto. Por exemplo, a Figura 3.6(a) ilustra um contexto de versões C contendo a versão v3 derivada de v2 que por sua vez foi derivada de v1. Eliminando v2, o contexto C deve ser corrigido para exibir

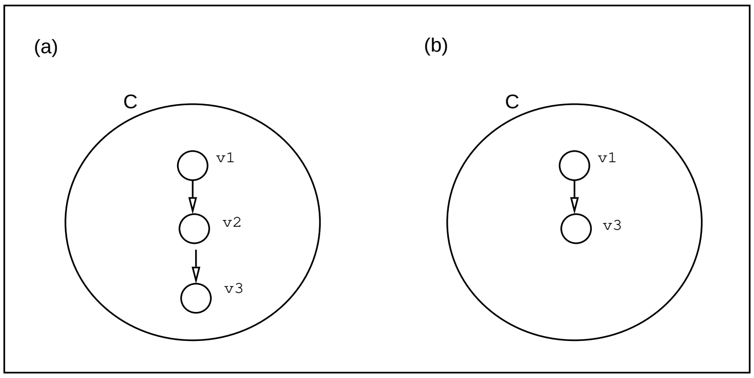

Figura 3.6: Eliminação de versões do Contexto de Versões

v3 sendo derivada de v1, Figura 3.6(b).

Um nó no estado Permanente possui as seguintes características:

- os atributos sujeitos ao versionamento não podem ser modificados;
- contém apenas nós Permanentes ou Obsoletos, caso seja um Contexto do usuário;
- não pode ser diretamente destruído;
- pode ser marcada Obsoleto mas não Temporário;
- pode derivar versões;

Um nó no estado Temporário possuiu as seguintes características:

- todos os seus atributos podem ser modificados;
- pode conter nós em qualquer estado, caso seja um Contexto do usuário;
- não pode derivar versões;
- pode ser destruído;
- pode ser marcado Permanente, mas não Obsoleto;

Um nó no estado Obsoleto possui as seguintes características:

- todos os seus atributos não podem ser modificados;
- não pode mudar de estado;
- pode conter apenas nós Permanentes ou Obsoletos, caso seja um Contexto do usuário;
- não pode derivar versões;
- é automaticamente destruído pelo sistema, quando não é referenciado por nenhum outro nó.

De acordo com as definições acima, se um nó V é derivado de W (diretamente ou transitivamente), então W ou é Permanente ou Obsoleto. Essa restrição garante que um nó contexto do usuário Permanente ou Obsoleto contém apenas nós Permanentes ou Obsoletos, que por sua vez implica em: (i) ele apenas contém elos cujos nós destino são Permanentes ou Obsoletos; (ii) a consulta que define seu conteúdo retorna um conjunto de nós Permanentes ou Obsoletos, se ele é um nó de contexto virtual; e (iii) as consultas nos seus elos ou âncoras sempre resultam em um conjunto de nós Permanentes ou Obsoletos.

3.5.2 Bases de Documentos

Em um ambiente de trabalho cooperativo é necessário oferecer aos usuários tanto a possibilidade de compartilhar informações quanto a de possuir informações privadas. Por isso, o MCA propõe a estratégia de subdivisão do espaço de nós, sendo cada subconjunto deste espaço denominado de Base de Nós. Uma base desse conjunto é chamada Base Pública ou Hiperbase, onde estão os documentos compartilhados. Todas as demais bases de nós são chamadas Bases Privadas, e equivalente a sessão de trabalho no Modelo de Dexter, usadas para modelar a interação do usuário com o hiperdocumento.

A Base pública é um tipo especial de nó de contexto que agrupa conjuntos de nós terminais e de nós contexto do usuário. A Base pública está relacionada com o Estado do nó, pois nela existem apenas nós no estado Permanente ou Obsoleto.

Uma base privada pode conter outras bases privadas, permitindo a organização da sessão de trabalho em várias subsessões aninhadas. Uma base privada é definida como um tipo especial de nó de contexto que agrupa qualquer entidade, exceto a base pública e nós contexto de versões, tal que:

- uma base privada pode pertencer a mais de uma base privada;
- se um nó de composição N está contido numa base privada BP, seus componentes ou estão contidos em BP ou na base pública ou em qualquer base privada do aninhamento de bases privadas contidas em BP; e
- se um elo está contido na base privada, seu ponto de origem deve ser um nó de Anotação.

Assim, uma base privada coleciona todas as entidades usadas pelo usuário durante uma sessão de trabalho.

3.5.2.1 Primitivas de Versões em Bases de Documentos

- *Check-Out*: esta primitiva é usada quando se deseja mover nós de qualquer base privada para a hiperbase pública. O nó a ser movido deve estar no estado Permanente. Se um nó contexto do usuário C é movido para a hiperbase, então todos os nós terminais e contexto do usuário contidos em C também devem ser movidos para a hiperbase. Não se pode mover um nó da hiperbase pública para alguma base privada, o que pode ser feito é criar uma versão de tal nó na base privada.

  Se depois da criação de uma versão V de um nó N da base pública em uma base privada esta nova versão não sofrer modificação, ao se passar a versão criada para a base pública, ela é destruída, pois não é necessário replicar os nós. Além disso, os contextos das bases privadas que contém V são atualizados para conterem N, ao invés de V. Da mesma forma, se V é uma versão de N que não foi modificada, todas as versões criadas de V devem ser transformadas em versões de N no nó contexto de versões.

- *Shift:* Esta primitiva serve para transferir todos os nós terminais e contexto do usuário de uma base privada para a hiperbase pública. Quando esta operação é aplicada, todos nós terminais e contexto do usuário da base privada são marcados Permanentes e movidos para a hiperbase pública, e todas suas bases privadas são recursivamente transferidas para a hiperbase pública. No fim desse processo a base privada que foi transferida irá conter apenas trilhas, anotações (e elos associados) e bases privadas, que, por sua vez, irá conter apenas trilhas, anotações e bases privadas, recursivamente.

- Duas primitivas *Open* e *Check-in* são usadas para criar novas versões Temporárias do nó N na base privada BP. Elas diferem quando N é um nó contexto do usuário. Neste caso:

  - *Open* cria uma versão temporária N_1 de N na BP, bem como de cada um dos componentes de N, e assim recursivamente. N_1 irá conter as novas versões dos componentes de N, e seus elos serão criados de maneira a refletir os elos em N. Se um componente permanente pertence a mais de um contexto, apenas uma versão temporária será criada para este nó. Quando se cria uma nova versão de um contexto do usuário da base pública em uma base privada, só são criadas versões dos nós presentes neste contexto quando estes nós são visitados. Por exemplo, seja um documento na base pública que contém um contexto do usuário C e por sua vez, este contexto do usuário contém os nós I, H e M. Se, em uma sessão de trabalho sobre este documento, em uma base privada qualquer, o contexto C é selecionado, automaticamente será criada uma versão temporária C_1 deste contexto do usuário na base privada, mas os nós presentes nele serão, a princípio, os mesmos I, H e M, e não novas versões destes nós. Se o nó H for selecionado, então será criada uma nova versão H_1, na mesma base privada, que substituirá H em C_1, conforme explicação acima. Se o nó M não for manipulado durante esta sessão de trabalho, a versão deste nó presente em C_1 permanecerá a original.
  - *Check-in* cria uma versão temporária N_1 de N, em BP, que contém os nós

originais contidos em N.

Como consequência disso, se N_1 é criado através da operação de *Check-in*, modificações feitas em nós contidos em N_1 irão criar novas versões desses nós. Assim, se o mesmo nó, M, aparece em dois nós contexto C_1 e C_2, e se ele é modificado nas duas perspectivas, duas diferentes versões, M_1 e M_2 serão criadas. Se, por outro lado, N_1 é criado com *Open*, uma única versão temporária M_1 será criada e irá sofrer modificações em ambas perspectivas.

- *Delete:* esta primitiva está disponível para o usuário remover um nó N da base privada. Se N é um nó trilha ou um nó de anotação, ele simplesmente será removido da base privada e destruído. Se N é um nó terminal ou contexto do usuário, o resultado depende do estado de N. Caso ele seja temporário, será efetivamente destruído e removido do seu contexto de versões; caso seja permanente, será marcado obsoleto.

  Quando um nó Permanente é marcado Obsoleto, ele é transferido da base privada que o contém para a base pública. Se o nó marcado Obsoleto for um contexto do usuário, todos os seus componentes serão também transferidos.

  Uma base privada pode também ser destruída. Neste caso, todos os seus nós, incluindo bases privadas, serão também destruídos.

3.5.3 Anotações

Facilidades para usuários comentar trabalhos são importantes para trabalho cooperativo e desenvolvimento exploratório. Uma anotação consiste em um comentário, em qualquer mídia, que suporta referências para as versões que ele comenta, e referências para as versões que são consideradas respostas à declaração contida na anotação.

Um nó *Anotação* é uma especialização do nó contexto que agrupa conjuntos de nós terminais, nós contexto do usuário e trilhas. Um ou mais de um nó nesse contexto contém os comentários feitos pelo usuário. Elos desses nós para diferentes pontos na base privada indicam os nós sendo comentados. Anotações podem apenas ser incluídas em bases privadas.

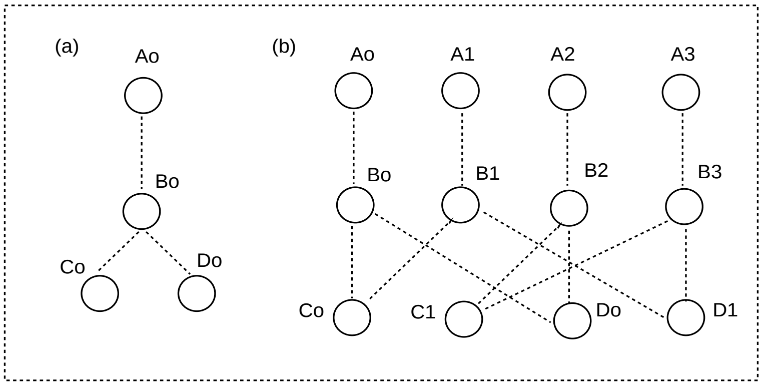

Figura 3.7: Proliferação de versões causadas pela propagação exaustiva

Anotações permitem a introdução de novos comentários referindo-se a nós permanentes, sem a criação de novas versões

3.6 Propagação de Versões

Propagação de versões é o processo de construir automaticamente uma nova configuração para incorporar uma versão resultante da derivação em um dos componentes da configuração [28]. Ou seja, é o mecanismo que gera uma nova versão de um objeto sempre que um dos seus sub-objetos derivar uma nova versão [18]. No modelo de Contextos Aninhados, quando uma versão é criada, todos os contextos do usuário que contém uma versão correlata são afetados. Assim, como uma versão pode estar contida em vários contextos do usuário, a propagação pode gerar a criação de um grande número de versões.

As questões chaves no mecanismo de propagação de versões são [28]:

- como limitar o escopo da propagação, pois dificilmente se deseja criar versões de todos os contextos.
- como evitar ambiguidades na propagação, pois, dependendo da perspectiva em que a propagação é efetuada, o resultado final pode ser diferente.

A Figura 3.7 ilustra a propagação exaustiva. A hiperbase inicial 3.7(a) contém um nó contexto a_0 que contém b_0 e que, por sua vez contém c_0 e d_0. Após criar as versões c_1 e d_1, correlatas às versões c_0 e d_0, respectivamente, a propagação ilimitada resulta na criação das versões a_1,a_2 e a_3, descendentes de a_0. As versões descendentes de b_0 foram criadas para conter as combinações das versões correlatas à c_0 e d_0. Então, b_1 contém c_0 e d_1, b_2 contém c_1 e d_0, e b_3 contém c_1 e d_1. Analogamente, são criadas versões descendentes de a_0 para conter as versões correlatas à b_0. A Figura 3.7(b) mostra o resultado obtido após a propagação de c_1 e d_1. Desta forma, quanto maior o número de contextos aninhados, maior é o número de combinações das versões criadas pela propagação exaustiva.

A Figura 3.8 mostra o problema da ambiguidade. Esse problema ocorre quando contextos diferentes possuem uma mesma versão, assim, tal versão pode ser identificada por diferentes perspectivas. Na hiperbase inicial, Figura 3.8(a), existem duas perspectivas para d_0: (d_0,a_0) e (d_0,b_0). Criando a versão d_1 correlata à d_0, de acordo com a perspectiva, existem três possibilidades de propagação. Pode-se propagar d_1 pela perspectiva (d_0,a_0), como na Figura 3.8(b). A segunda opção, ilustrada na Figura 3.8(c), consiste em propagar d_1 pela perspectiva (d_0,b_0). E finalmente, pode-se propagar d_1 pelas duas perspectivas, como mostra a Figura 3.8(d). Desta forma, dependendo da perspectiva escolhida para a propagação de d_1, o resultado final do documento é alterado.

Várias sugestões foram emitidas em [28] para evitar a ambiguidade:

- Proibir qualquer propagação de versões. Obviamente, essa opção é bastante restritiva.
- Propagar as versões criando todas as configurações possíveis. Mas, geralmente, essa proliferação é indesejável.
- Permitir a propagação onde não ocorre ambiguidade.
- Definir mecanismos operacionais através dos quais a aplicação pode tornar suas intenções não ambiguas.

No modelo em questão optou-se por deixar a cargo do usuário a escolha de propagar ou não. Caso o usuário ative o mecanismo de propagação, tal propagação ocorrerá apenas

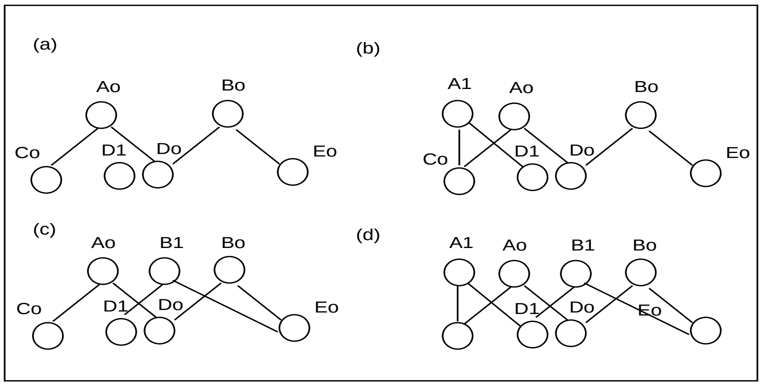

Figura 3.8: Ambiguidade causada pela propagação ao longo de perspectivas

na perspectiva corrente. E, como existe apenas um caminho definido pela navegação [7] para se chegar a uma versão, então, é possível limitar a propagação a esse caminho, de forma não ambigua.

Por exemplo, suponha que, depois de sucessivas navegações, a sequência de nós marcadas na Figura 3.9 corresponda à perspectiva corrente. Para criar uma versão correlata a d_0, o usuário deve navegar para a versão b_0 e solicitar a criação da versão d_1. A nova perspectiva corrente é a sequência mostrada na Figura 3.9(b). Aplicando a propagação, a perspectiva torna-se a ilustrada na Figura 3.9(c).

Criando a versão e_1, correlata a e_0, conforme a Figura 3.9(d), ao propagar e_1, o efeito é mostrado na Figura 3.9(e). Observe que a versão b_2 criada de b_1, contém d_1 e não d_0, isto porque a perspectiva corrente quando e_1 foi criada, continha b_1 que, por sua vez, continha d_1.

A estratégia de propagação adotada considera o estado das versões presentes na perspectiva corrente. Como já foi visto anteriormente, se uma versão estiver no estado Temporário, não poderá derivar novas versões. Por exemplo, no momento da propagação caso a versão b_1 seja Temporária, ao invés de criar uma nova versão b_2, a propagação altera o conteúdo de b_1 para conter e_1 ao invés de e_0, Figura 3.9(f). Desta forma, apenas as versões Permanentes da perspectiva corrente acarretam a criação de versões durante a propagação.

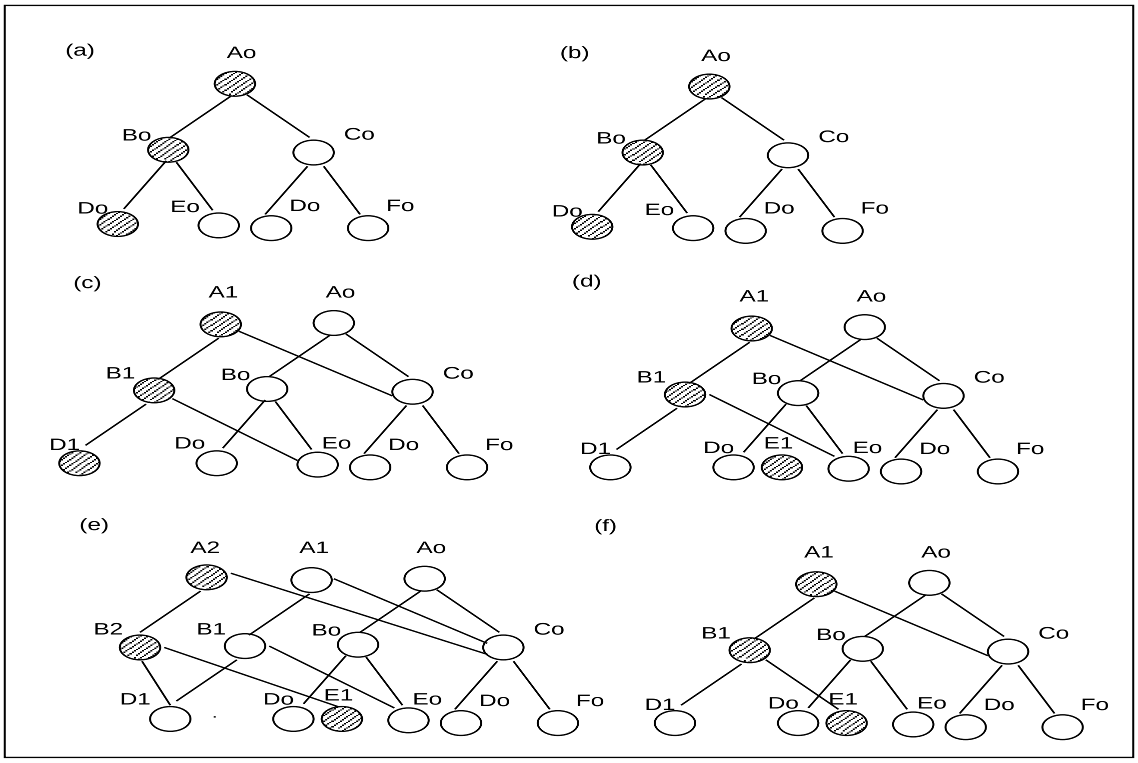

Figura 3.9: Propagação de versões na perspectiva corrente

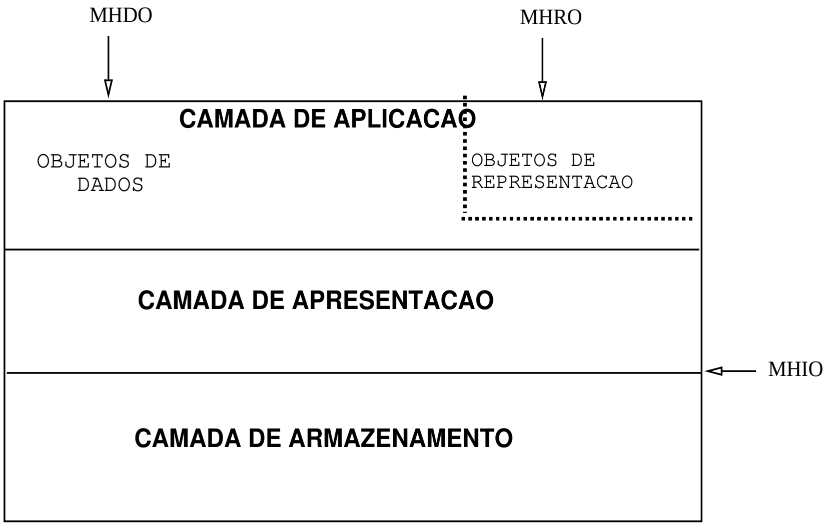

Figura 3.10: Arquitetura Hipermídia em Camadas

3.7 Versões na Arquitetura Hipermídia em Camadas

O objetivo da definição da arquitetura hipermídia em camadas é principalmente fornecer flexibilidade e interoperabilidade, permitindo a integração de várias aplicações multimídia. Flexibilidade é conseguida oferecendo interfaces nos vários níveis através das quais as aplicações podem interagir e também isolando os mecanismos de armazenamento que podem ser ajustados nas aplicações. Interoperabilidade é possível devido a interface de intercâmbio de objetos MHEG oferecida pela arquitetura.

Em [11] é apresentada detalhes da arquitetura em questão e são estabelecidas as relações entre os objetos intercambiáveis definidos pelo padrão MHEG e as abstrações do MCA, visando permitir o intercâmbio dos objetos multimídia.

3.7.1 Camada de Armazenamento

É responsável pelo armazenamento e recuperação dos objetos. Para tal, implementa objetos de armazenamento persistentes, que possui um identificador único, um tipo específico e atributos. Esses objetos correspondem no Modelo de Contextos Aninhados a hiperbase pública e ao contexto de versões. A interface MHIO (Interface de Objetos Intercambiáveis Hipermídia Multimídia) é oferecida por esta camada para permitir compatibilidade entre aplicações e equipamentos, possibilitando intercâmbio de dados hipermídia que corresponde a codificação dos objetos multimídia de acordo com o formato estabelecido pelo padrão MHEG.

3.7.2 Camada de Apresentação

É responsável pelo mapeamento e conversão do formato de armazenamento dos objetos de dados usados pelas aplicações para o formato de intercâmbio definido pela interface MHIO, e vice-versa. Desempenha portanto o papel de Formatador/Analizador MHEG.

3.7.3 Camada de Aplicação

Define os conceitos de **Objetos de Dados** e **Objetos de Representação**. Trata das funções referentes ao gerenciamento da base privada dos sistemas.

- **Objeto de Dados:** corresponde a um objeto novo ou a uma versão do objeto de armazenamento juntamente com os atributos não persistentes definidos pela aplicação. Sua função é gerenciar e fornecer persistência aos objetos da base privada em uma sessão do sistema hipermídia. Para isto, oferece métodos para manipular os atributos não persistentes e métodos para manipular informação pertencentes ao objeto de armazenamento. Assim, o objeto de dados inclue métodos para criação e remoção de nós, exportação de nós para a hiperbase e importação da hiperbase e também métodos para manipulação dos atributos dos objetos.

- **Objetos de Representação:** consiste em uma especialização da classe de objetos de dados adicionando métodos para manipulação do conteúdo dos nós e para navegação em uma sessão do sistema hipermídia. O objeto de representação é uma nova versão de um objeto de armazenamento, derivado de um objeto de dados.

Duas interfaces são oferecidas pela Camada de Aplicação:

- Interface MHDO (Interface para Objetos de Dados Hipermídia Multimídia)
- Interface MHRO (Interface para Objetos de Representação Hipermídia Multimídia)

Estas interfaces contém, respectivamente, os métodos associados com os objetos de dados e objetos de representação.

Capítulo 4

Mecanismo de Notificação

4.1 Introdução

A finalidade deste capítulo consiste em discutir o problema da notificação de mudança. Segundo [28] o problema da notificação é simples e qualquer mecanismo baseado em sinais ou em mensagens parece ser suficiente para as aplicações, todavia, os trabalhos que discutem tal problema demonstram que a notificação de mudança requer que sejam estudados os requisitos das aplicações e dependendo das particularidades de tais aplicações, é necessário um mecanismo mais elaborado.

A seguir, na seção 4.2, são descritos trabalhos que endereçam mecanismos para resolução do problema da notificação.

Na seção 4.3 são discutidos requisitos de notificação para o modelo de contextos aninhados.

Procurando atender os requisitos descritos na seção anterior, uma técnica de notificação para o modelo de contextos aninhados é proposta na seção 4.4

A seção 4.5 expõe o escopo de notificação para contextos aninhados.

Finalizando o presente capítulo encontra-se uma descrição de uma possível implementação da técnica apresentada na seção 4.4.

4.2 Trabalhos Correlatos

4.2.1 Batory e Kim

Batory e Kim [5] consideram que um problema importante em bancos de dados CAD é a notificação de mudança e endereçam uma solução para o problema em seu modelo de dados. Mudanças feitas em versões podem afetar a validade de versões de outros objetos relacionados. Tais mudanças devem ser monitoradas, e objetos afetados devem ser notificados. Os relacionamentos que são importantes na notificação de mudança são aqueles envolvendo versões instanciadas. Por exemplo, se a implementação da versão A instancia a versão B e a versão B é alterada, então A deve ser notificada da mudança de B. Como esses relacionamentos são explicitamente declarados em tal modelo, o sistema pode monitorar mudanças em versões e fazer automaticamente as notificações apropriadas.

O mecanismo fornecido por esse modelo trabalha da seguinte maneira. Toda versão tem duas marcas de tempo distintas. Uma marca de tempo, chamada de marca de tempo de notificação de mudança (CN), indica a última vez que uma versão foi mudada. A outra, chamada marca de tempo de aprovação de mudança (CA), indica a última vez em que o proprietário da versão aprovou as mudanças realizadas.

Seja V.CA e V.CN, respectivamente, as marcas de tempo de aprovação e notificação de mudança da versão V. V é *consistente em implementação* se V.CA $\geq$ V.CN. Em outras palavras, se seu proprietário aprovou as mudanças realizadas. Então, uma versão é inconsistente em implementação quando atualizações forem feitas e não forem aprovadas.

Seja I o conjunto de versões que são instanciadas na implementação da versão V. Se nenhuma versão em I tem a marca de notificação de mudança que excede a marca de aprovação de mudança de V (isto é, se para todo X $\in$ I, X.CN $\leq$ V.CA), então V é *consistente em referências*. V é inconsistente em referências se há uma ou mais versões em I que tenham sido atualizadas, mas os efeitos dessas atualizações em V não tenham sido determinadas. Para tornar V consistente em referências, os efeitos das atualizações das versões em I devem ser conhecidos. Isto pode ser feito de duas formas: ou as atualizações em I não têm efeito em V, no caso V.CA é setado para a hora corrente, ou a implementação de V necessita ser modificada, no caso V.CN (e possivelmente V.CA se as mudanças forem aprovadas) necessitam ser setados para a hora corrente. Enquanto tais ações não sejam feitas, V permanece inconsistente em referências.

O mecanismo de marcas de tempo descrito acima pode resultar em uma versão que é vista como consistente, enquanto que uma versão relacionada (isto é, uma que a instancia) é inconsistente. Por exemplo, suponha que a versão V seja criada instanciando uma versão A que é consistente em referência e implementação. Agora suponha que A torne-se inconsistente em referências devido a modificação de uma versão que ela instancia. A consistência em implementação e referência de V é afetada apenas se as modificações indiretas a A cause a modificação direta de A. Até que essa determinação seja feita, V permanece consistente. Claramente há um problema de sincronização. O modelo determina que uma versão é *Totalmente Consistente* se ela e suas versões instanciadas são consistentes em implementação e em referências.

4.2.2 Chou e Kim

Chou & Kim [9] estenderam o trabalho de Batory & Kim e forneceram um esquema para o problema da notificação de mudança incluindo aspectos relativos à implementação.

- Requisitos para Notificação de Mudança:

  No modelo em questão uma versão de um objeto pode referenciar qualquer número de versões de outros objetos. No contexto da hierarquia de sua base de dados isto representa uma das seguintes opções:

  - Uma versão transiente ou de trabalho na base de dados privada pode referenciar outras versões transientes ou de trabalho na mesma base de dados privada, versões de trabalho numa base de dados de projeto do usuário da base de dados privada, ou versões liberadas.

- Uma versão de trabalho na base de dados de projeto pode referenciar outras versões de trabalho na mesma base de dados de projeto ou versões liberadas.
- Uma versão liberada pode referenciar apenas outras versões liberadas.

De acordo com essas referências que o sistema pode suportar, as seguintes situações podem exigir notificação de mudança:

- Uma versão transiente ou de trabalho na base de dados privada referencia uma versão transiente na mesma base de dados privada e a versão transiente é atualizada, destruída ou deriva uma nova versão.
- Uma versão transiente ou de trabalho na base de dados privada referencia uma versão de trabalho na mesma base de dados privada ou numa base de dados de projeto, e a versão referenciada é destruída ou deriva uma nova versão.
- Uma versão de trabalho na base de dados de projeto referencia outra versão de trabalho na base de dados de projeto, e a versão referenciada é deletada ou deriva uma nova versão.

• Tipos de Técnicas de Notificação:

Num ambiente CAD distruibuído, dois tipos de técnicas de notificação podem ser suportadas: baseada em mensagens ou baseada em sinais.

Na abordagem baseada em mensagem, o sistema envia mensagens para notificar os usuários das versões afetadas. O mecanismo baseado em mensagens é distinguido como imediato ou deferido dependendo se os usuários afetados são notificados imediatamente depois das mudanças em uma versão ou em algum momento mais tarde.

Na abordagem baseada em sinais, o sistema simplesmente atualiza as estruturas de dados mantidas, de tal forma que os usuários afetados tomarão conhecimento das mudanças apenas quando fizerem um acesso explícito a versão. Esta abordagem é necessariamente uma estratégia de notificação deferida.

Assim, um objeto tem opções de notificação de mudanças à sua disposição:

- mensagens x sinais
- imediata x deferida (no caso de notificação baseada em mensagens)

  - tipos de mudanças que determinam a notificação (atualização, remoção, criação de uma nova versão).

  Quando a aplicação define um objeto, deve especificar essas opções com respeito aos objetos versionados que ela referencia. Contudo, é impraticável solicitar ao usuário que especifique um conjunto diferente de opções para cada uma das referências no objeto. Então, é mais razoável que se tenha apenas um único conjunto de opções especificada para um objeto, e aplicá-las a todos os objetos que ele referencia.

- Técnica de Notificação adotada pelo modelo

  O modelo em questão adota a técnica de notificação baseada em sinais.

  Cada versão do objeto possui duas marcas de tempo distintas. Uma marca de tempo, chamada marca de tempo de notificação (CN), indica a hora em que a versão foi criada ou a última vez que foi modificada. A outra, chamada marca de tempo de aprovação (CA), indica a última vez que o proprietário da versão aprovou as mudanças.

  Para cada objeto é necessário manter o número da versão de cada versão que ele referencia. Isto é necessário para suportar referência dinâmica. Suponha que uma versão Va referencia uma versão transiente Vb3, que foi derivada de uma versão de trabalho Vb2. Vb3 é atualizada, e consequentemente Va aprova as mudanças em Vb3. Então Vb3 é destruída. Com referência dinâmica, Va referenciará Vb2. A marca de tempo de aprovação de mudança não captura o fato de que a aprovação foi para as mudanças feitas em Vb3, e não em Vb2.

  Uma diferente técnica de notificação, chamada Percolação de Versão resolve esse problema. Nessa técnica, quando uma nova versão é derivada de uma velha versão de um objeto, o sistema automaticamente gera novas versões dos objetos que diretamente ou indiretamente referenciam a versão velha. Essa técnica tem algumas falhas. Uma é que ela gera um grande número de versões não usadas. No exemplo mostrado na Figura 4.1, suponha V1 de um objeto A referencie V1 de um objeto B e V1 de um objeto C. Se o proprietário deriva novas versões V2 dos objetos B e C, mesmo se a intenção do proprietário foi apenas criar uma nova versão V2 do objeto A, o sistema irá gerar três novas versões de A. V2 de A irá referenciar V1 de B e V2 de C; V3 de A irá referenciar V2 de B e V1 de C; e V4 de A irá referenciar V2 de B e V2 de C.

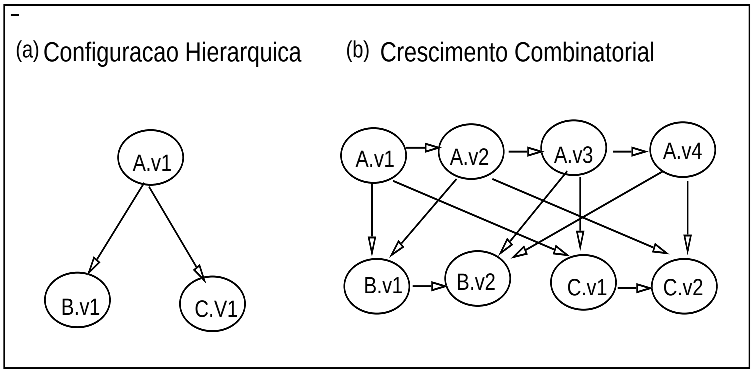

Figura 4.1: Percolação de Versão

Outra falha da técnica de percolação é que tal técnica não é útil quando versões de um objeto componente são deletadas. No exemplo acima, o proprietário de V1 do objeto A não será notificado quando V1 do objeto C ou V1 do objeto B for destruída.

- Escopo de Notificação

  O fato de que uma versão em geral referencia outras versões que, sucessivamente, referenciam outras versões, representa o problema com o escopo da notificação de mudança. Supondo, por exemplo, que uma versão Vi referencie Vj e Vj referencie Vk. Se a versão Vk é modificada, tanto Vj quanto Vi devem ser notificadas, ou apenas Vj deve ser notificada? Em geral, as possibilidades são:

  - notificar apenas as versões que diretamente referenciem a versão modificada
  - notificar todas as versões que diretamente ou indiretamente referenciem a versão modificada

  A filosofia atrás da primeira abordagem, que é a adotada no modelo em questão, é que o proprietário de Vj, a versão que diretamente referencia Vk, a versão modificada, deve reagir à modificação. O proprietário pode determinar que nenhuma ação corretiva deve ser realizada em Vj, assim, não há necessidade de notificar Vi, a versão que referencia Vj. Apenas se o usuário atualizar Vj, em resposta às mudanças em Vk, as modificações em Vj irão causar notificação ao proprietário de Vi.

numero de componentes	metodo de notificacao	tipo de evento
descritor do componente 1		
....		
descritor do componente n		

Figura 4.2: Formato da Tabela do Componente

O fato de notificar apenas Vj é especialmente forte no modelo em questão. Como um objeto CAD tem a parte de interface e a parte de implementação e a parte de interface não é modificável, apenas atualizações na parte de implementação podem resultar novas versões. Assim, há uma pequena probabilidade de que Vj necessite ser atualizada em resposta à atualizações na parte de implementação de Vk, e então, a probabilidade é menor ainda em relação a necessidade de atualização em Vi.

- Implementação

  Para a técnica de notificação baseada em sinais, uma tabela como a da Figura 4.2 é associada com cada versão e contém a seguinte informação sobre as versões que ela diretamente referencia como componente.

  - o número de componentes da versão
  - o método de notificação de mudança (ou baseada em sinais ou baseada em mensagens)
  - o tipo de evento que requer notificação (criação, atualização ou remoção)
  - um conjunto de descritores dos componentes

  Cada descritor do componente contém a identidade da versão referenciada e o tipo de referência que está sendo usada. Para referência estática, o nome completo da versão referenciada é armazenado. Para referência dinâmica, o nome da base de dados e/ou o número da versão cujas mudanças foram aprovadas mais recentemente

nome da versao referenciada	nome da versao	metodo de notificacao	tipo de evento	...	nome da versao	...

Figura 4.3: Formato da Lista de Referência Invertida

pela versão pai, são armazenadas para um nome de base de dados não especificado e/ou o número da versão.

Para suportar notificação baseada em mensagens, é necessário manter para cada versão V, uma lista de referência invertida das versões que referenciam V e que solicitam notificação caso ocorra mudanças em V. Quando uma nova referência a V é criada, o nome da versão que referencia V é adicionado à lista de referência invertida de V. Como mostrado na Figura 4.3, para cada referência, o tipo de evento (uma combinação de atualização/remoção/criação de versões) e o tipo de notificação (imediata ou deferida) são também armazenadas. Quando uma versão, com um lista de referência invertida não vazia é modificada, a lista é percorrida para a base de dados que contém atualmente essas versões e mensagens são enviadas aos seus proprietários.

4.3 Requisitos de Notificação no MCA

Os ambientes CAD distribuídos possuem muitas semelhanças com o ambiente de edição cooperativa de documentos multimídia. Ambos visam auxiliar o trabalho em equipe, representam objetos como estruturas hierárquicas, demandam suporte a versões e mecanismos de notificação de mudança. Porém, os requisitos de notificação apresentam necessidades diferentes nos dois ambientes.

Em ambientes CAD geralmente as transações são de longa duração, fazendo com que o projetista detenha o objeto de trabalho por longos períodos de tempo. Assim, não há necessidade de notificação imediata, e a abordagem de notificação por sinais é suficiente na maioria das vezes.

Os ambientes multimídia, em geral, possuem a particularidade de fornecerem aplicações em tempo real, como, por exemplo, teleconferência, o que demanda, para o problema da notificação, uma abordagem diferente da adotada em ambiente CAD, visto que neste tipo de aplicação as interações com os usuários acontecem de forma instantânea. Por outro lado, para outras aplicações, o mecanismo baseado em sinais é bastante adequado, uma vez que, não há necessidade de uma notificação imediata.

No MCA diz-se que um nó $\mathbf{N}_1$ referencia um nó $\mathbf{N}_2$ quando $\mathbf{N}_1$ é a origem de um elo cujo destino é $\mathbf{N}_2$.

No contexto da hierarquia das bases de nós do MCA, nós na base pública não necessitam do mecanismo de notificação visto que são nós no estado Permanente e como tal não podem sofrer alterações. Além disso, por estarem na base pública não referenciam nós não permanentes.

Assim, o mecanismo de notificação no MCA será operacional apenas para as bases privadas e deverá notificar nós nos seguintes casos:

- Nós terminais ou contexto do usuário no estado permanente ou temporário que referenciem algum nó no estado temporário e tal nó foi atualizado.
- Nós terminais ou contexto do usuário no estado permanente ou temporário que referencie algum nó que se tornou obsoleto, assim, o mecanismo de notificação deve informar a mudança de estado do nó referenciado.
- Um nó contexto do usuário no estado permanente ou temporário que contenha algum nó que se tornou obsoleto, assim, o mecanismo de notificação deve informar a mudança de estado do nó contido neste contexto do usuário.
- Nós terminais ou contexto do usuário no estado permanente ou temporário que referencie algum nó permanente que derivou uma nova versão, então, o mecanismo de notificação deve informar que foi criada uma nova versão do nó referenciado.

- Um nó contexto do usuário no estado permanente ou temporário que contenha algum nó permanente que derivou uma nova versão, então, o mecanismo de notificação deve informar que foi criada uma nova versão do nó contido neste contexto do usuário.

Vale salientar que, não haverá notificação no caso de um nó temporário **conter** outro nó temporário que foi atualizado. Neste caso o proprietário tem conhecimento de que qualquer um dos nós presentes no contexto poderão ser atualizados, então não há necessidade de notificação. O mesmo argumento poderia servir para o caso de um nó temporário **referenciar** outro nó temporário que foi atualizado, mas isto não acontece porque no caso de referência, existe o problema da âncora onde o elo chegará. Pode ser que a atualização da âncora não seja conveniente para o nó que a referencia.

4.4 Proposta de Notificação Adotada no MCA

De acordo com os requisitos de notificação discutidos na seção anterior, o mecanismo de notificação para contextos aninhados deve agir nos seguintes casos:

- Atualização do nó referenciado
- Mudança de Estado (quando o nó referenciado ou contido no contexto do usuário é marcado Obsoleto)
- Derivação de nova versão (o nó referenciado ou contido no contexto do usuário deriva uma nova versão)

Portanto, esses são os três tipos de alteração que determinam a notificação. Devido a natureza das aplicações multimídia, o mecanismo de notificação deve combinar estratégias imediatas e deferidas. A notificação deverá ser feita de forma imediata quando a alteração atingir a representação que está sendo manipulada pelo usuário, ou seja, a representação corrente. Se o nó atingido pela alteração não se encontra na representação corrente, a notificação deferida é suficiente.

O mecanismo de notificação para contextos aninhados funciona da seguinte maneira. Todo nó terminal ou contexto do usuário na base privada deve possuir as seguintes informações associadas:

- um campo *Notificação* que quando ativado sinaliza que há notificação para o nó
- *Tipo de Alteração* que indica o tipo de evento que provocou a notificação(atualização, mudança de estado ou derivação de nova versão), ou seja, se o nó está sendo notificado de uma atualização, uma mudança de Estado ou uma derivação de nova versão.
- *Nó Alterado* que indica em qual nó referenciado ou contido no contexto do usuário ocorreu a alteração

Além disso, cada nó **N** deve possuir uma lista de referências invertida e uma lista de pertinência.

A lista de referências invertida contém todos os nós que referenciam **N** e serão notificados nos casos de alteração de N. Quando uma nova referência de $\mathbf{N}_1$ para **N** é criada, $\mathbf{N}_1$ é adicionado na lista de referências invertida de **N**.

A lista de pertinência contém todos os contextos do usuário onde o nó **N** está contido e serão notificados no caso de **N** mudar de estado ou derivar uma nova versão. Assim, quando o nó **N** é inserido em um contexto do usuário **C**, **C** é incluído no lista de pertinência de **N**.

A necessidade de se definir duas listas para cada nó se explica devido a diferença entre os eventos que determinam a notificação no caso de um nó terminal ou contexto do usuário **referenciar** outro ou no caso de um nó contexto do usuário **conter** outro. Nesta última opção, apenas a mudança de estado ou a derivação de nova versão provocam a Notificação. Assim, é necessário registrar em listas diferentes os dois tipos de relacionamento entre os nós para poder verificar os eventos que determinar a notificação em cada um desses casos, e ativar corretamente o mecanismo de notificação.

Consideramos que associar essas informações a cada nó é importante para evitar o esforço cognitivo do proprietário do nó notificado, uma vez que, sem tais informações, o proprietário teria que percorrer a lista de referências invertida e a lista de pertinência de cada nó para descobrir qual nó referenciado ou contido no contexto do usuário foi alterado e que tipo de alteração foi efetuada.

Para a estratégia imediata, pode-se empregar o mecanismo de notificação baseado em mensagens. Quando um nó que contém uma lista de referências invertida e/ou lista de pertinência não vazias é modificado, os proprietários de cada nó destas listas que exigir notificação imediata são notificados através de mensagens emitidas pelo sistema. Recebendo a mensagem, o proprietário consulta os campos *Tipo de Alteração* e *Nó Alterado* e desativa o campo *Notificação* para indicar que está ciente da alteração. Quando o campo *Notificação* é desativado, o sistema libera os campos *Tipo de Alteração* e *Nó Alterado.*

Para a estratégia deferida, as informações associadas a cada nó são suficientes. Então, quando um nó que contém uma lista de referências invertida e/ou lista de pertinência não vazia é modificado, o sistema apenas atualiza os campos *Notificação, Tipo de Alteração* e *Nó Alterado.* Quando o proprietário realizar o próximo acesso ao nó, verificará se o campo *Notificação* encontra-se ativado. Da mesma forma que na notificação baseada em mensagens, estando ciente da alteração, o proprietário deve desativar o campo *Notificação.*

Esse mecanismo de notificação permite ainda que um determinado nó receba mais de uma notificação ao mesmo tempo pois os campos *Tipo de Alteração* e *Nó Alterado* poderão ser instanciados, respectivamente, com uma lista de tipos de alteração e com uma lista de nós.

4.5 Escopo de Notificação no MCA

A alternativa adotada pelo modelo de contextos aninhados em relação ao escopo de notificação consiste em notificar apenas as versões que referenciem diretamente a versão modificada. Essa estratégia evita a proliferação de notificações desnecessárias uma vez que, na maioria das vezes, a modificação em uma versão só afeta outras versões que a

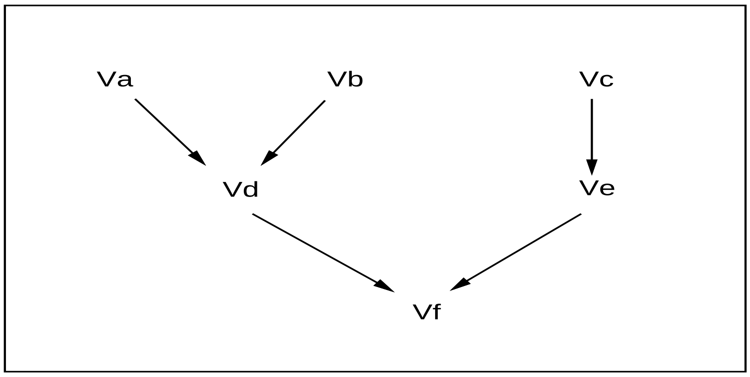

Figura 4.4: Escopo de Notificação

referencie diretamente. Além disso, como o modelo de contextos aninhados divide o nó em interface e implementação e impede que a interface seja atualizada, e ainda restringe as referências à interface do nó, então, as atualizações na implementação possue efeito reduzido sobre outros objetos que referenciam a sua interface. Assim, uma notificação restrita a referências diretas é suficiente [10].

Considerando, por exemplo, a situação da figura 4.4 onde as versões Va e Vb referenciem uma versão Vd que, por sua vez, referencie uma versão Vf. Considerando também que a versão Vc referencie Ve que referencia Vf. Se a versão Vf é modificada, na estratégia adotada pelo MCA, apenas Vd e Ve serão notificadas. Se a alteração em Vf ocasionar uma alteração em Ve então Vc será também notificada. Da mesma forma, apenas se Vd sofrer alterações é que as versões Va e Vb serão notificadas. Assim, reduzindo-se o escopo de notificação, evita-se a proliferação desnecessária de mensagens.

4.6 Aspectos Relativos a Implementação

A implementação do mecanismo de notificação descrito acima deverá associar a cada nó **N** as seguintes estruturas de dados:

- *Notificação*: Flag

- *Nós Alterados*: poderá ser instanciado com uma lista de nós
- *Tipo de Alteração*: consiste de uma lista com número de elementos igual ao número de elementos da lista de nós do campo *Nós Alterados*. Cada elemento desta lista poderá ser instanciado com os caracters *A, ME ou DV* indicando respectivamente Atualização, Mudança de Estado e Derivação de Versão. Os elementos desta lista são correspondentes aos elementos da lista de nós que compõe o campo *Nós Alterados*, de forma a indicar que tipo de alteração ocorreu no nó alterado correspondente.
- *Lista de Referências Invertida*: instanciado com a lista de nós que referencia **N**
- *Lista de Pertinência*: instanciado com a lista de nós contexto do usuário que contém **N**.

Quando o nó **N** sofre alguma alteração que determine a notificação dos nós que o referenciam e/ou que o contém, a lista de referência invertida e/ou a lista de pertinência são percorridas e para cada nó da lista, o sistema ativa o campo *Notificação*, instancia o *Tipo de Alteração* e envia o nome do nó para o campo *Nós Alterados*. Caso o nó notificado esteja na representação corrente, o sistema envia uma mensagem informando ao proprietário que ocorreu alguma alteração em um dos nós referenciados. Assim, o proprietário deverá verificar as estruturas de dados associadas ao seu nó para identificar a alteração ocorrida

Capítulo 5

Implementação

5.1 Introdução

A implementação do protótipo de um sistema mono-usuário com controle de versões é o tema em discussão no presente capítulo.

Essa implementação tem dois objetivos principais. O primeiro é oferecer uma biblioteca de classes - a **HipClass** - que reflete o Modelo de Contextos Aninhados e pode ser usada para se construir sistemas hipermídia. O outro objetivo consiste em proporcionar um ambiente para autoria e navegação em hiperdocumentos - o **Sistema HIP** - que usa a biblioteca de classes. Assim, pretende-se oferecer uma sessão de trabalho onde o usuário possa interagir com o sistema para criar seus hiperdocumentos. Uma sessão de trabalho é modelada como sendo uma base privada, desta forma, é necessário implementar a parte referente a base privada do modelo. Durante este capítulo, será usado o termo Hiperbase não como sinônimo da Hiperbase Pública, mas no sentido de representar a base privada.

O protótipo deve ser extensível de forma que não seja restrito a uma aplicação específica e deve ser modular para que o processo de estendê-lo seja uma atividade elementar. Para modularizar o sistema e facilitar sua extensão, a implementação foi dividida em módulos, conforme mostra a figura 5.1. Os módulos implementados nesse trabalho encontram-se

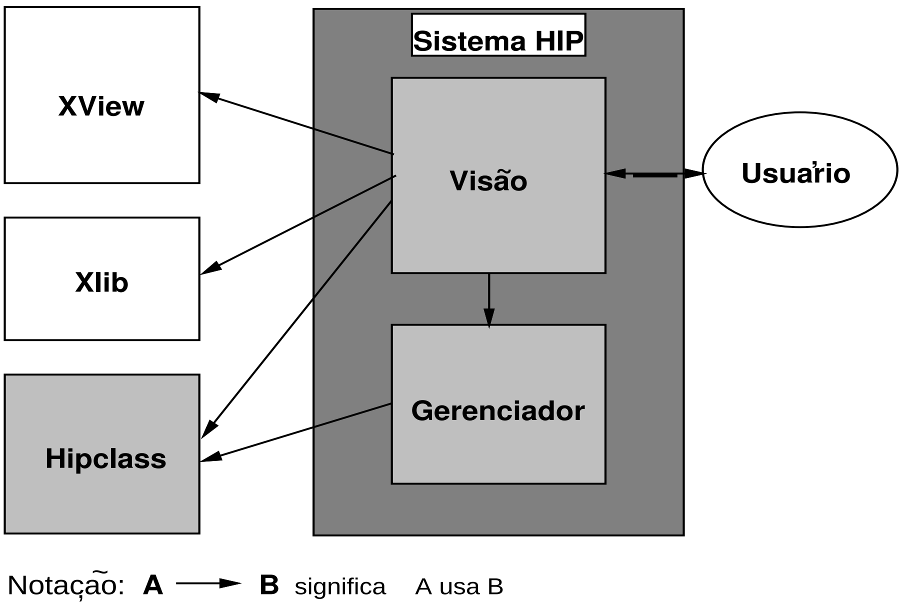

Figura 5.1: Visão Geral

destacados em tonalidades mais escuras. Os outros módulos são bibliotecas usadas para construção da interface e estão apresentadas adiante.

A função da HipClass é permitir que sejam criados objetos de um sistema hipermídia que seguem o Modelo de Contextos Aninhados, tais como Nós texto, Nós de Contextos, Nós Contexto de Versão, Âncoras e Elos. Esses objetos serão chamados de **Objetos do MCA**.

No HIP, o módulo **HipVisão** proporciona ao usuário a visualização dos objetos do MCA e coordena a interação com o usuário através de uma interface gráfica no estilo *"WIMP - Windows, Icons, Menus e Pointers"*. O HipVisão apenas representa os objetos do MCA, enquanto que as alterações dos estados desses objetos é gerenciada pelo **HipGer**.

De acordo com a arquitetura proposta na seção 3.7, como a camada de aplicação trata do gerenciamento da base privada dos sistemas, este trabalho situa-se em tal camada, correspondendo a implementação de parte dos objetos de dados e de objetos de representação. Na arquitetura, a porção dos objetos de dados inclue métodos para gerenciar e fornecer persistência aos objetos da base privada. Assim, este trabalho abrange apenas parte dos objetos de dados, uma vez que não oferece a persistência dos objetos. O objeto de representação é uma especialização do objeto de dados adicionando métodos para manipulação do conteúdo dos nós e para navegação. Como este trabalho oferece métodos para criação e manipulação dos nós texto e contexto, além de métodos para navegação, então os objetos de representação são também implementados.

O ambiente de implementação sobre o qual o protótipo foi desenvolvido é comentado na seção 5.2.

A seção 5.3 apresenta a biblioteca de classes que implementa os objetos do modelo de contextos aninhados e os métodos para sua manipulação. Portanto, constitui a parte central deste protótipo, sobre a qual diferentes aplicações podem ser construídas.

O HIP - uma ferramenta para autoria e navegação em sistemas hipermídia - é o assunto em discussão na seção 5.4. A seção apresenta o que a ferramenta oferece para o usuário, a sua arquitetura básica e a relação entre objetos do MCA e os objetos visuais da interface.

Para concluir o presente capítulo, a seção 5.5 mostra como o protótipo suporta modificações e extensões. É feita uma avaliação das qualidades do protótipo como modificabilidade, extensibilidade, portabilidade e reusabilidade. Em especial, será ressaltada a facilidade de extensões da biblioteca de classes para tratar outras mídias além da mídia texto. Será discutido ainda a possibilidade de implementação de outros ambientes utilizando diferentes ferramentas para construção de interfaces gráficas, e suporte a vários usuários. Os pontos necessários para tornar o sistema multi-usuário são também apresentados.

5.2 Ambiente de Implementação

A plataforma de hardware utilizada para a implementação foram as estações de trabalho SPARCstation2 da Sun Microsystems. Ela foi escolhida por sua performance satisfatória em aplicações que necessitam de interfaces gráficas, e pela disponibilidade do compilador para a linguagem utilizada e de bibliotecas para a construção da interface gráfica com o usuário. Alem disso, essa plataforma é hoje bastante comum no meio acadêmico, permitindo uma maior divulgação do trabalho entre pesquisadores da área.

O código fonte da biblioteca de classes foi implementado utilizando a linguagem C++, compatível com a versão 2.1 da AT&T para o sistema operacional SunOs 4.1 das estações Sun. O XView Toolkit em conjunto com o Xlib foram utilizados na construção da interface - módulo HipVisão. Os módulos do HIP foram implementados em linguagem C para maior compatibilidade com as bibliotecas do Xview e do Xlib. Como C++ é uma extensão da linguagem C, utilizou-se o mesmo compilador no desenvolvimento de todo o protótipo.

A linguagem C++ foi escolhida por estar disponível em diferentes plataformas e possibilitar a implementação dos conceitos da abordagem orientada por objetos, como: classe, herança e polimorfismo, atendendo, portanto, aos requisitos de generalidade, portabilidade e extensibilidade solicitados pelo modelo.

Utilizar uma linguagem de programação orientada por objetos oferece um número de vantagens importantes para as aplicações. Uma dessas vantagens é a modelagem de todas as entidades conceituais em um simples conceito: **objetos**. O comportamento de um objeto é capturado através de mensagens as quais um objeto responde. Outra vantagem da programação orientada por objetos é a noção de **hierarquia de classes** e **herança** de propriedades ao longo da hierarquia de classes. A hierarquia de classes captura o relacionamento IS-A entre uma classe e suas subclasses. Todas as subclasses de uma classe herdam todas as propriedades definidas para a classe e podem ter propriedades adicionais locais a elas. Como o MCA é composto por uma hierarquia de objetos conforme visto no capítulo 3, e para cada objeto são definidos operações e um conjunto de atributos, nada mais natural do que implementá-las usando uma linguagem orientada por objetos. Assim, foram definidas classes com seus atributos e métodos, que correspondem aos objetos definidos pelo modelo. Além dessas classes, outros tipos abstratos de dados auxiliares, como listas, são de enorme importância para a construção dos objetos do MCA. Listas de Nós, de Elos e de Âncoras são implementadas facilmente usando o conceito de herança proporcionado pela linguagem.

O XView Toolkit [23] foi escolhido como ferramenta para implementação da interface pela boa qualidade da interface que se pode construir usando as facilidades que ele oferece para construção de objetos da interface como janelas, botões e menus, seguindo o padrão OPEN LOOK. Assim, o sistema pode ser executado no ambiente de trabalho *Open Window* da Sun.

O XView é construído sobre o Xlib, que é uma biblioteca que segue o padrão X Window. Assim, como está descrito adiante na seção 5.5, não haverá grandes dificuldades em se utilizar um outro toolkit baseado no X Window/Xlib.

Rotinas do Xlib [34] são utilizadas para os desenhos dos ícones que representam os objetos do MCA, ou seja, dos objetos visuais.

Vale ressaltar que, devido a independência entre a implementação do modelo e sua interface gráfica, a biblioteca de classes pode ser executada em qualquer plataforma que possua um compilador compatível com o C++ da AT&T.

5.3 A Biblioteca de Classes

A biblioteca de classes implementa as abstrações definidas pelo modelo conceitual de contextos aninhados. Esse é o seu requisito básico. Para isto, foi feito um mapeamento entre os objetos do modelo e as classes a serem implementadas. A figura 5.2 mostra o modelo de classes que apresenta a hierarquia de classes bem como os tipos de relacionamento entre elas. Este modelo segue as convenções notacionais definidas na metodologia OMT (Técnica para Modelagem de Objetos), descrita em [38].

Nesta biblioteca encontram-se diponíveis classes para criação dos objetos da base privada do MCA e operações para manipulação e gerenciamento destes objetos. Assim, a biblioteca de classes consiste na máquina hipermídia propriamente dita, que trata dos eventos gerados pelas aplicações e oferece serviços para se manter e recuperar documentos estruturados.

Não é função da HipClass oferecer uma interface para autores ou usuários que desejem navegar em hiperdocumentos. Ela oferece meios para engenheiros de software que queiram

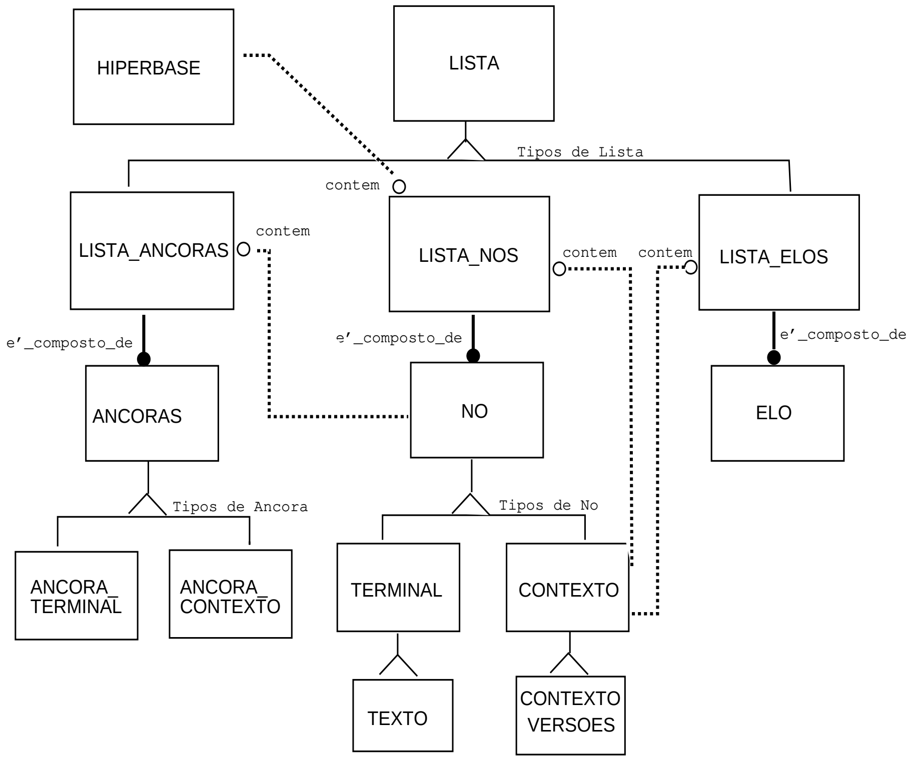

Figura 5.2: Modelo de Classes

construir sistemas para desenvolvimento de aplicações hipermídia. Tal como nas implementações orientadas por objetos, as classes definem as interfaces dos serviços por elas oferecidos. Os clientes, i.e., os sistemas usam esses serviços enviando mensagens. São mensagens solicitando informações ou realizando alterações nos estados dos objetos do MCA.

5.3.1 Os serviços oferecidos pela HipClass

Os serviços oferecidos por esta biblioteca de classes são aqueles determinados no Modelo de Contextos Aninhados. De acordo com o modelo, os documentos são formados por nós que são conectados por elos. Assim, a biblioteca de classes deve oferecer operações para a criação de nós e elos na base privada. No escopo deste trabalho é tratada apenas a mídia texto nos nós terminais. A biblioteca fornece operações para **criação de nós contexto ou texto**. O **identificador** é a identificação interna do nó e é gerado automaticamente quando um nó é criado. Encontra-se também disponível as operações de **criação e eliminação de âncoras** em um nó. Tais operações dependem da classe do nó, visto que o ponto de ancoragem tem representação diferente para cada tipo de nó. Ainda sobre âncoras, pode-se **mudar o valor** de uma âncora, o que é especialmente necessário quando se cria versões de um nó. Neste caso, a máscara da versão não pode ser alterada. Pode-se apenas remanejar as âncoras já existentes.

A operação de **criação de versão** constitui um ponto importante dessa biblioteca de classes. Para maximizar os benefícios do controle de versões, o sistema encarrega-se de automaticamente realizar as tarefas de gerenciamento, evitando o máximo possível a interação com usuário, de forma a abreviar o esforço cognitivo. Assim, sendo solicitada a criação de versão, o sistema analisa se o estado do nó permite a criação de versões do mesmo. Sendo possível, encarrega-se da criação ou atualização do contexto de versões e do ajuste dos atributos. O atributo **versão corrente** é setado assim que é criada a primeira versão de um nó. Este atributo contém inicialmente, o identificador do primeiro nó inserido no contexto de versões. Posteriormente, pode-se utilizar a operação de **mudança da versão corrente** para indicar qual a versão deve ser considerada a versão corrente.

Para os nós contexto estão disponíveis operações para **inserir nós em um contexto** e

retirar nós de um contexto.

Em relação aos **Elos**, a operação de **criação de elos** entre nós de um contexto solicita os nós origem e destino do elo e suas respectivas âncoras. Para **retirar elos** basta indicar o elo a ser retirado.

Para garantir a integridade, há o controle de acesso às operações através da manutenção do atributo **Estado do Nó**. Existem operações para **Mudança de Estado** do Nó.

Para maiores detalhes sobre os serviços oferecidos pela biblioteca de classes uma consulta a [4] é recomendada.

5.3.2 Limitações

Com a frequente evolução do Modelo de Contextos Aninhados, o desenvolvimento do protótipo não acompanhou a última etapa de alteração do modelo conceitual, pois esta se deu recentemente quando o escopo desse trabalho já se encontrava delimitado e o protótipo concluído. Assim, esse último passo da evolução não está incluído no protótipo. Porém, isso não constitui um grande problema, uma vez que essa recente extensão apenas refinou os conceitos anteriores, o que não comprometeu a coerência dos aspectos implementados no protótipo. Este, então, pode ser facilmente adaptado à nova realidade através do uso da biblioteca de classes disponível. Entre os aspectos não incluídos na implementação ressalta-se os nós trilhas e Anotação. Outro ponto importante é que no caso do nó contexto, a implementação das âncoras considera apenas que o ponto de ancoragem seja ou o contexto como um todo, ou um dos nós dentro do contexto. Na definição atual do modelo, o ponto de ancoragem pode ser definido também como sendo uma lista de nós ou um subconjunto de nós contidos no contexto. Um problema análogo acontece com os elos. O modelo permite que um elo possua múltiplos destinos, porém, a implementação considera que um elo possui apenas um ponto destino.

Não houve preocupação com a persistência dos objetos criados durante uma sessão de trabalho, visto que, em paralelo com o protótipo em questão, foi implementado um outro, cujo enfoque é a persistência dos objetos da base privada seguindo as normas fixadas pela proposta de padronização MHEG para intercâmbio de objetos multimídia [11].

Uma outra limitação é que a biblioteca de classes foi desenvolvida em paralelo com o protótipo que implementa a persistência dos objetos da base privada. Com isso, esta biblioteca não pôde incluir os aspectos implementados em tal protótipo. Acredita-se que não haja muito problema de compatibilidade entre essas duas implementações, uma vez que ambas seguem as abstrações definidas no modelo de contextos aninhados, foram especificadas seguindo os conceitos de orientação por objetos e obedecendo a hierarquia proposta no modelo, e implementadas utilizando a mesma linguagem de programação.

Como suporte a Trabalho Cooperativo, a HipClass implementa os conceitos de Estado e o gerenciamento de versões. Assim, ela oferece um suporte limitado, necessitando ser expandida de forma a incluir a Base Pública, Propagação de Versões, Mecanismo de Notificação e demais aspectos relativos a cooperatividade.

A seguir será visto um exemplo de aplicação que usa os serviços da HipClass.

5.4 HIP - uma ferramenta para autoria e navegação

O sistema HIP é um sistema para desenvolvimento de aplicações hipermídia. Com ele um usuário pode realizar as tarefas de autoria e navegação em hiperdocumentos através de uma interface gráfica. Essa interface proporciona um ambiente confortável para o usuário realizar operações com a máquina hipermídia implementada pela biblioteca de classes, apresentada na seção 5.3.

A interface, conforme dito anteoriormente, foi construida usando a ferramenta XView [23], que proporciona um estilo padrão de interface, o OPENLOOK, comum ao ambiente OpenWindows.

Esta interface permite o redimensionamento, movimentação e iconificação de janelas. Operações através de cliques em botões ou escolhas em menus. Informações e atributos podem ser obtidos, fornecidos ou modificados usando-se caixa de diálogos. Permite a visualização de gráficos e ícones, entre vários outros recursos. Tudo isso torna o sistema muito fácil de usar, especialmente para aqueles familiarizados com o padrão.

5.4.1 O modelo de Objetos

O HIP é, então, um sistema que oferece uma sessão de trabalho para o usuário visualizar os objetos do MCA e realizar operações sobre eles. É possível visualizar-se a hiperbase, os nós texto, contexto e contexto de versões, os elos e as âncoras.

Tem-se então dois níveis de objetos. Os objetos do MCA que são construídos pelas classes do HipClass e os objetos visuais[1] que representam os primeiros, mostrando o seu estado. A criação ou modificações nos estados de objetos do MCA deve ser refletida imediatamente na sua visualização. Por outro lado, as operações realizadas pelo usuário sobre os objetos visuais devem resultar em modificação nos estados dos objetos do MCA.

A figura 5.3 mostra essa correspondência. As setas indicam que é a partir dos objetos visuais que os do MCA são inspecionados ou modificados. Isso é bastante interessante em termo de projeto, pois facilita a construção e evolução do sistema.

5.4.2 A arquitetura básica do HIP

O sistema HIP é uma arquitetura em camadas. Na figura 5.4 podemos identificar os três níveis. As setas na figura se referem ao uso de recursos entre os módulos. O módulo HipGer, por exemplo, é cliente do módulo HipClass.

O primeiro, de baixo para cima, é o nível do modelo que corresponde ao módulo HipClass, descrito na seção 5.3.

O segundo nível, o gerenciador, usa serviços oferecidos pelo nível inferior. Essa camada é responsável pelo gerenciamento da criação, destruição e modificações dos objetos no

[1]Embora janelas, menus, botões, etc. sejam objetos visuais, aqui se considera por esta denominação apenas os que representam os objetos do MCA.

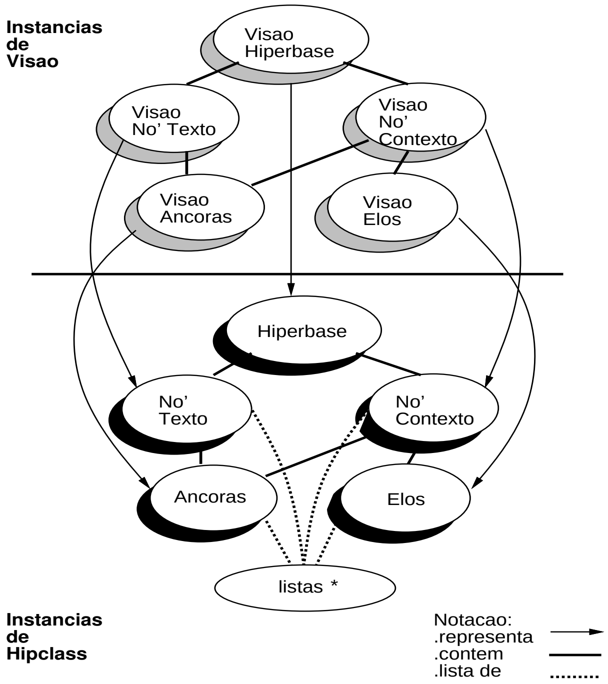

Figura 5.3: Modelo de Objetos

MCA provenientes da camada superior, que por sua vez é manipulada pelo usuário. Ele funciona como um intermediário entre a visão e o modelo.

O nível gerenciador contém o módulo HipGer. As rotinas deste módulo são ativadas pelos módulos que implementam a interface com o usuário quando este quer realizar tarefas que alteram o estado dos objetos do MCA. Essas rotinas se encarregam de verificar se determinadas condições estão satisfeitas ou não para que a operação possa ser realizada. Caso isto não seja possível, mensagens são enviadas ao usuário. Esse gerenciamento se faz necessário para que a biblioteca de classes tenha uma arquitetura mais enxuta, evitando que seus métodos tenham a preocupação de verificar estado de outros objetos e devolver mensagens de erro usando recursos da interface. O gerenciador foi introduzido na arquitetura visando o controle da execução de tarefas. Esse controle deve obedecer regras definidas pelo MCA, como, por exemplo, permitir a criação de versões apenas de nós permanentes. Assim, o gerenciador reflete também o MCA . A única ligação do HipGer com a interface é que ele chama a rotina da interface que mostra mensagens de erro ao usuário.

Se o módulo HipGer não existisse, o gerenciamento ficaria a cargo da interface ou da biblioteca de classes. Ficando a cargo da interface, esta passa a ter uma maior dependência do modelo, sendo encarregada não só da sua visualização mas também do controle de operações. Se o gerenciamento fosse de responsabilidade da biblioteca de classes, seus métodos ficariam com algoritmos mais complexos e com mais interdependência entre si. Além disso, para as operações que não satisfizerem as condições necessárias, mensagens deveriam ser enviadas ao usuário. Isso aumentaria a interação da biblioteca de classes com a interface. A existência do gerenciador torna mais viável a expansão para um sistema multi-usuário com trabalho cooperativo, como será visto na seção 5.5.

O terceiro nível, denominado visão, corresponde ao que foi denominado de HipVisão. Ele é composto por vários submódulos.

O módulo principal, o HipMain, controla a criação dos objetos visuais e oferece alguns serviços básicos. Os outros métodos possuem rotinas para permitir que o usuário possa visualizar e interagir com os objetos do MCA. Cada módulo implementa uma visão correspondente a um tipo de objeto. As principais visões estam descritas na seção 5.4.3.

A única função desse nível que é usada por um nível mais inferior é a que mostra as mensagens de erro provenientes do gerenciador.

5.4.3 Autoria e navegação

Nessa subseção, é descrita a interface que possibilita ao usuário realizar as tarefas de autoria e navegação em hiperdocumentos, operando sobre a máquina hipermídia.

Ao ser ativado, o sistema mostra uma janela que espelha a base privada que é referida aqui como hiperbase, figura 5.5. A Hiperbase irá conter todos os nós referentes aos hiperdocumentos criados naquela seção de trabalho (vale lembrar que não existe persistência). O usuário pode interagir com os nós através de operações que podem ser ativadas por botões, menus ou mesmo por manipulação direta do tipo *"drag-and-drop" e "tdrag-and-load"*. Com essas operações o usuário pode criar e eliminar nós, inseri-los dentro de um contexto e consultar ou modificar seus atributos.

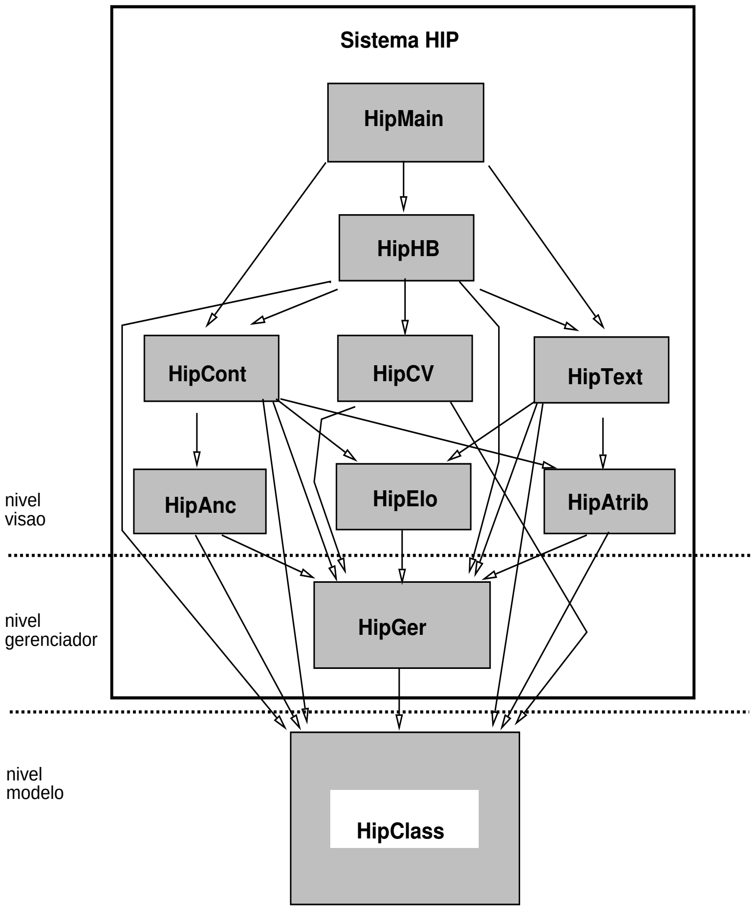

Figura 5.4: Arquitetura Básica do HIP

Os nós são representados por ícones indicando os diferentes tipos de nós.

Na janela exibida na parte inferior da hiperbase pode-se visualizar o nós contexto de versões. Esses nós estão separados, por serem criados automaticamente pela máquina hipermídia e por não pertencerem a hiperbase. A operação **abrir** é a única disponível para os nós contexto de versões.

A visualização de um nó contexto pode ser vista na janela mostrada na figura 5.6. As operações oferecidas ao usuário são bastante parecidas com as da hiperbase. Na visão do contexto pode-se tambem visualizar, criar, eliminar e mudar âncoras e elos. Na parte inferior da figura são mostradas as âncoras do contexto que está sendo exibido.

Um nó contexto de versões, tal como os nós contexto, contêm nós. A janela de um contexto de versões é mostrada na figura 5.7. Apenas a operação de mudar a versão corrente e a visualização da mesma estão disponíveis para este objeto.

Um nó texto é visualizado através de uma janela gráfica, como mostra a figura 5.8. Nela são oferecidas para o usuário as operações de criar, eliminar e mudar âncoras e elos. É possível também mudar atributos através de uma caixa de diálogo. As âncoras são visualizadas através de destaques que são dados na palavra (ou palavras) do texto as quais elas estão associadas. O destaque consiste em apresentar as letras da(s) palavras(s) onde está a âncora, em negrito. Uma âncora que possui um elo também é destacada visualmente.

A operação de edição requer que uma janela para edição de texto seja ativada. Essa janela utiliza os recursos de edição oferecidos pela XView, o que evitou que um editor de texto fosse implementado.

Um fato importante a ser ressaltado é que mesmo que um usuário modifique um texto que possui âncoras, a posição de cada uma delas é preservada.

O mecanismo de navegação implementado no HIP possibilita ao autor de hiperdocumentos navegar na estrutura que ele está criando. Relembrando, o Hip mostra visualizações dos objetos do MCA.

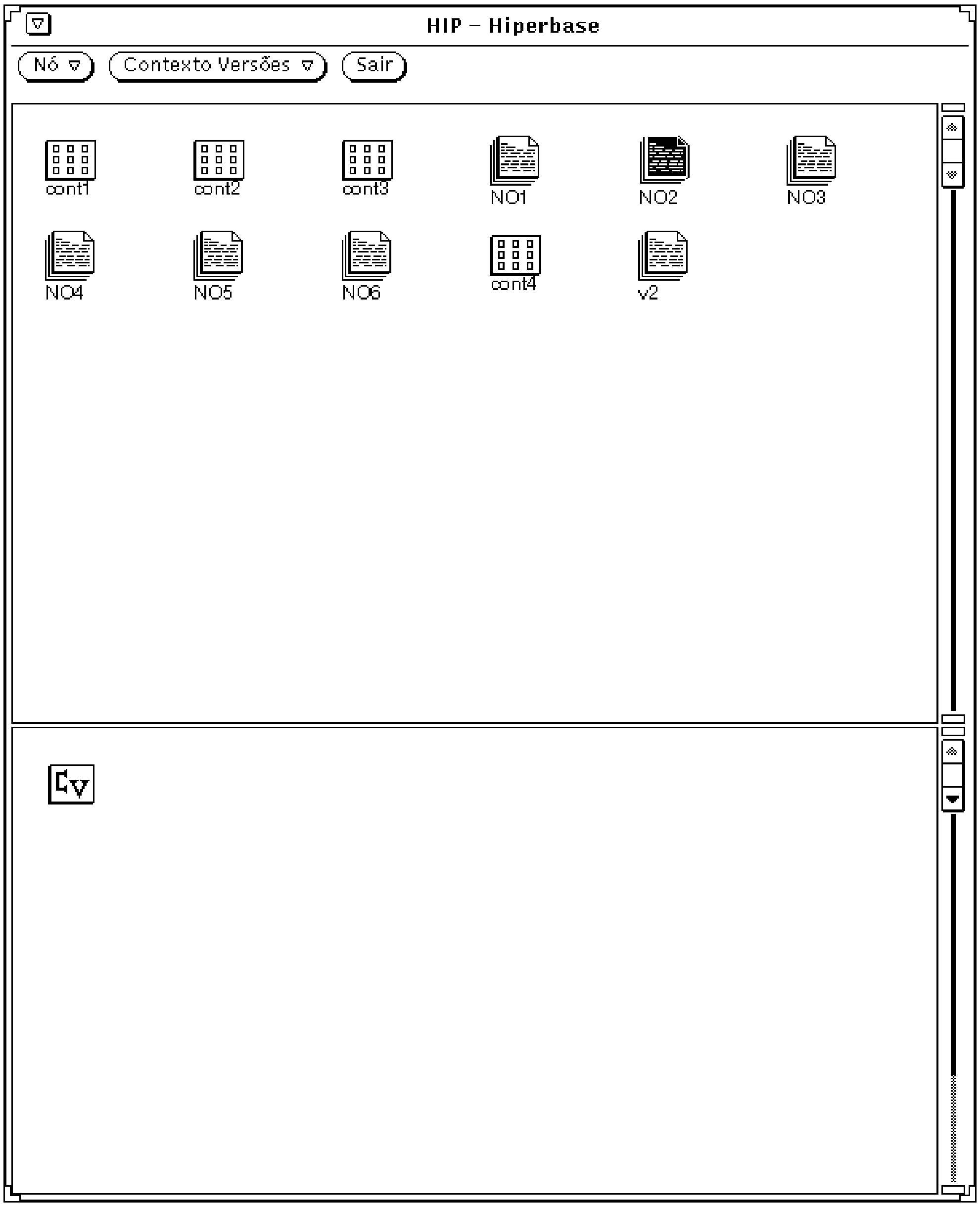

Figura 5.5: A hiperbase a visualização dos contextos de versões (parte inferior)

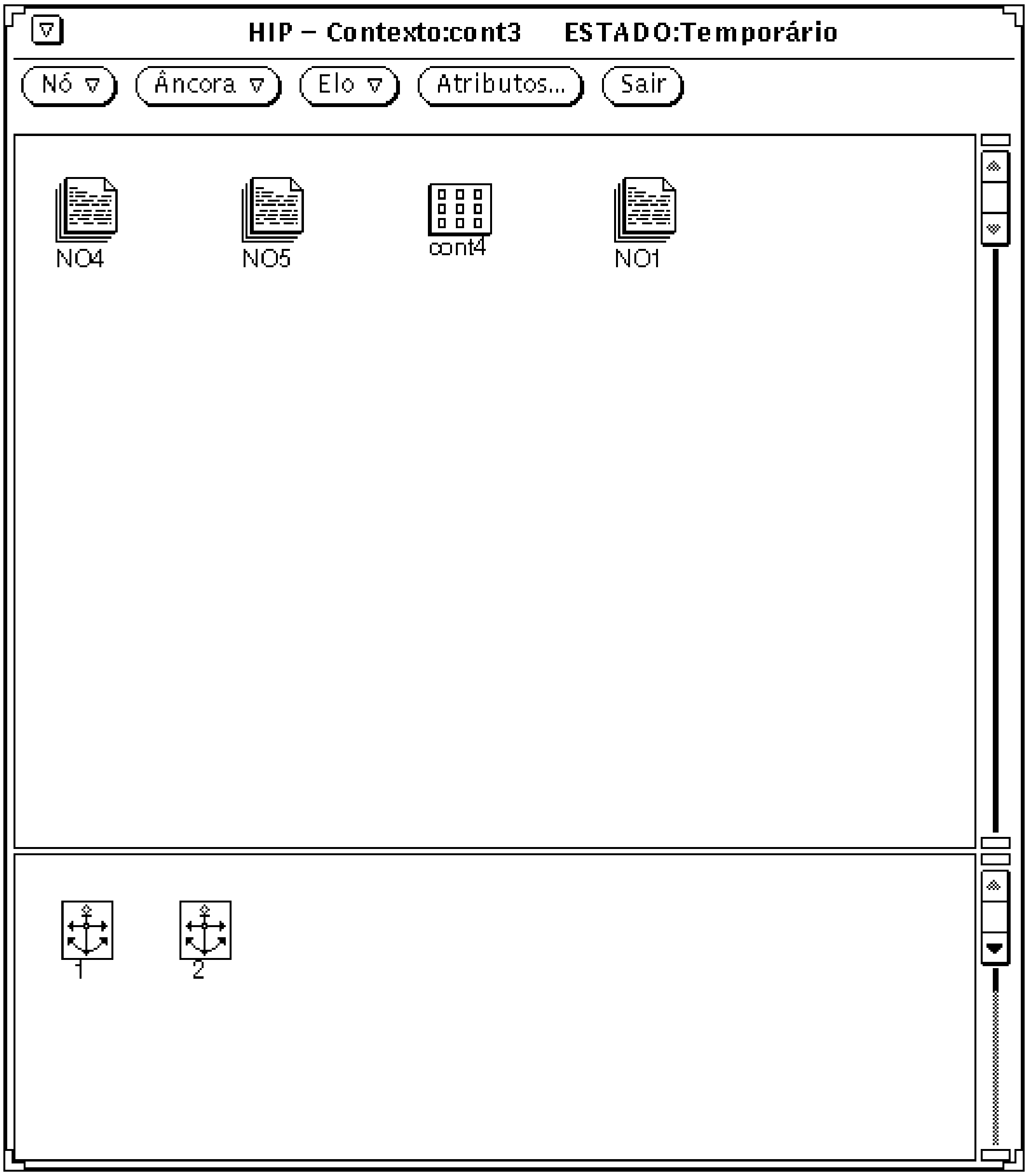

Figura 5.6: Um nó contexto e suas âncoras (na parte inferior)

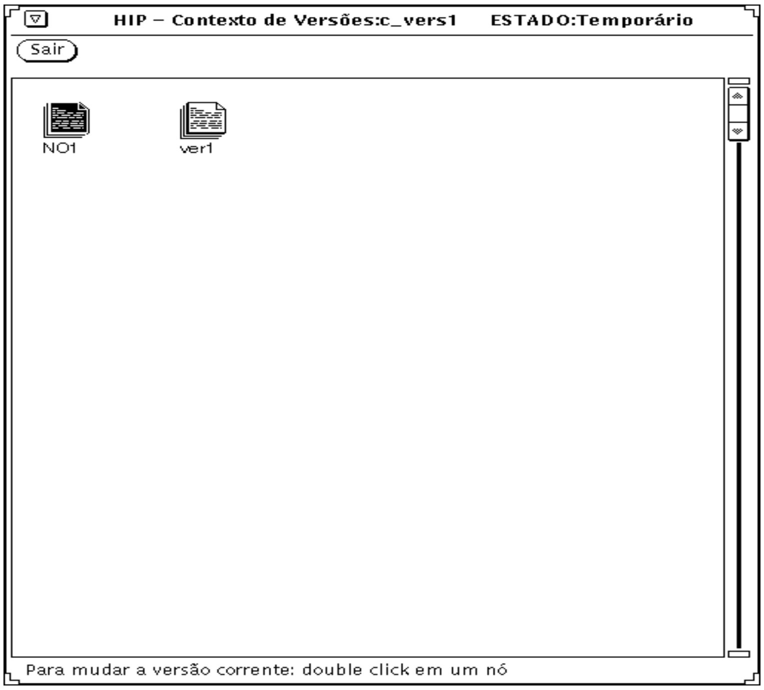

Figura 5.7: Um nó contexto de versões com a versão corrente em preto

No contexto desse sistema, a navegação consiste em percorrer os nós organizados hierarquicamente pela estrutura de contextos aninhados e em atravessar os elos existentes nos níveis desta hierarquia.

No nó texto, ao se visualizar uma âncora, caso ela esteja associada à origem de um elo, pode-se navegar clicando-se duas vezes com o mouse. O elo é percorrido e o usuário navega para um outro ponto do hiperdocumento.

Conforme foi dito, esse trabalho não se preocupou em projetar ou implementar o modelo de apresentação (ver capítulo 3), que deve ser definido para cada aplicação. O mecanismo de navegação aqui implementado, é importante ressaltar, não é aquele que deve ser oferecido a um usuário que deseje obter informações ou realizar um *'browsing'*. Nesse tipo de interface objetos como nós contextos, elos, versões e contexto de versões não são visualizados pelo usuário. Apenas o nós terminais, em particular a visualização de seu conteúdo e as representações das âncoras são apresentados ao usuário.

Em resumo, o HIP oferece uma interface para um autor, onde ele pode elaborar hiper-

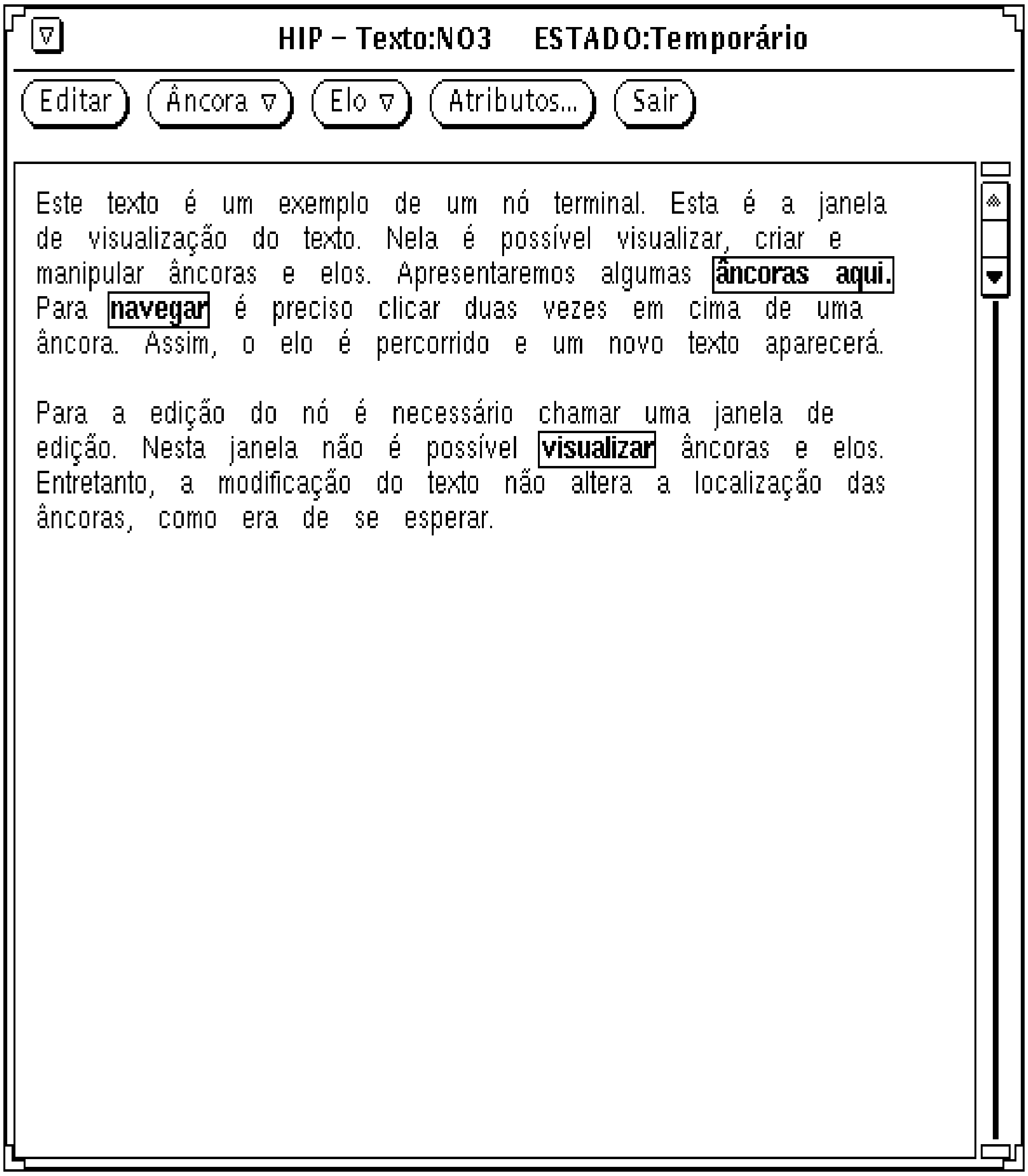

Figura 5.8: Nó texto e suas âncoras

documentos e navegar através de seus componentes. Não foi definido um modelo de apresentação para a exibição dos hiperdocumentos construídos.

5.4.4 Limitações

Ainda não foi implementado um mecanismo de coleta de lixo automática dos nós obsoletos. O usuário precisa fazê-la manualmente através da operação **eliminar nó** oferecida pela interface.

Também não possível ter visualização e manipulação direta dos elos ligados aos nós. Apenas a ligação com as âncoras indica que em um determinado nó um elo tem origem ou destino.

5.5 Evolução do protótipo

Nessa seção são discutidos aspectos de qualidades do sistema implementado no que diz respeito a facilidades de modificações, extensões, portabilidade e reuso.

5.5.1 Extensões à biblioteca de classes

Dois fatores foram primordiais no que diz respeito às qualidades de software da biblioteca de classe. A implementção segue o modelo de contextos aninhados e foi projetada e programada seguindo os conceitos de orientação por objetos. A seguir são discutidos, como estes fatores proporcionaram as qualidades.

A incorporação de novas classes à biblioteca acontecerá quando houver alterações no MCA. Mas, tendo em vista que o sistema é ainda um protótipo, uma das coisas que mais foi levada em consideração durante o projeto e implementação foi a sua extensibilidade.

O HipClass é fechado em si mesmo. Nenhuma alteração nas outras partes do sistema -

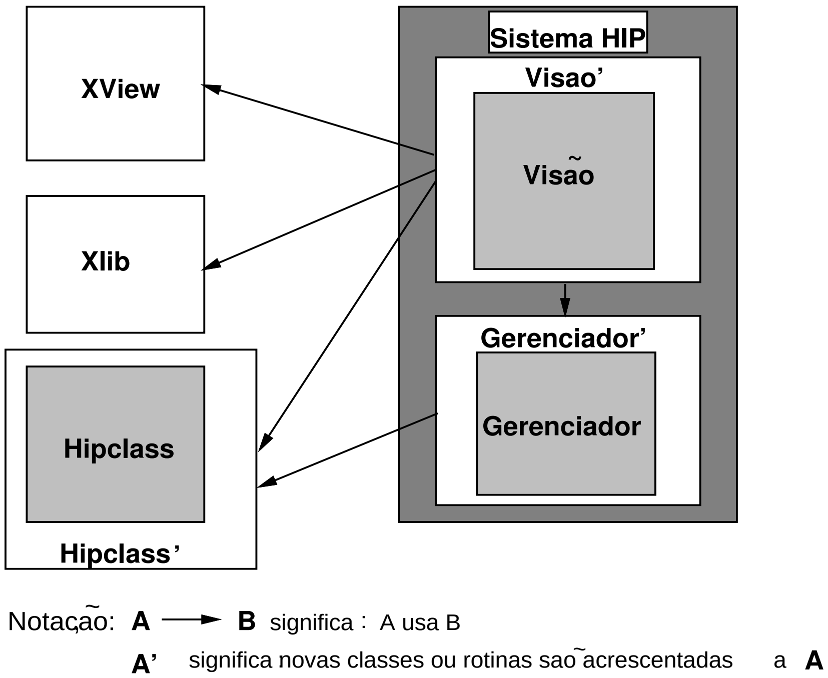

Figura 5.9: Modificações necessárias em extensões à HipClass

HipVisão, Gerenciador ou nas bibliotecas XView e Xlib - causam efeito no HipClass. Por outro lado, se extensões ou modificações feitas neste módulo, os módulos do Sistema HIP precisam ser alterados. Isso é esperado, pois o Hip é responsável pela visualização dos objetos construídos pela biblioteca. Obviamente, as bibliotecas Xview e Xlib não sofrem qualquer modificação. Esse aspecto está ilustrado na figura 5.9.

Como o módulo gerenciador também incorpora alguns conceitos do MCA no que diz respeito a verificação de condições e deteção de erros, ao se tentar reusar a biblioteca de classes para construir uma outra interface para Autoria e Navegação, é necessário que exista também um módulo gerenciador. Pode-se aproveitar o mesmo módulo HipGer, ou, pode-se também construir um outro gerenciador.

Caso se queira estender a biblioteca para que ela permita outras mídias além da texto, novas classes precisam ser incorporadas. Essa classes serão extensões às classes existentes na biblioteca e deverão pertencer a mesma hierarquia, tal como no modelo. Pouquíssimas ou nenhuma alterações serào necessárias aos componentes já existentes.

A classe **vídeo**, por exemplo, deve ser subclasse da classe **nó terminal**. Como as âncoras estão associadas e são dependentes do tipo do nó terminal, uma nova classe **âncora vídeo** precisa também ser implementada.

No HIP, novas rotinas precisam ser implementadas para permitir ao usuário manipular e visualizar as extensões que são incorporadas à biblioteca de classes. Isso será discutido na seção 5.5.2.

A manutenibilidade do HipClass foi favorecida, pois seguindo o MCA, a classes possuem a hierarquia dos objetos do modelo com as operações bem definidas. Isso não só facilitou a implementção como torna as modificações e manutenções atividades simples. Com base na estrutura e hierarquia do modelo, as classes implementadas são componentes bastante encapsulados que fornecem serviços a outras classes e outros módulos através de uma interface bem definida. A arquitetura, embora possua uma certa complexidade, está bem estruturada e a implementação dos métodos bastante enxuta.

Modificações em rotinas sem alterações na sua interface, normalmente não causam impacto nos outros módulos. Houve uma grande preocupação em projetar bem as interfaces para que as mesmas não sofressem modificações sempre que os métodos fossem alterados.

A reusabilidade de componentes de software é considerada um recurso poderoso em grandes sistemas de software. O reuso de classes ou rotinas de bibliotecas facilita bastante a vida dos programadores. O sistema HIP fornece exemplos de reuso de componentes, uma vez que sua construção é baseada no uso das bibliotecas HipClass, Xview e Xlib.

A biblioteca de classes também mostra outros exemplos de reuso. Devido a características da linguagem orientada por objetos utilizada, a implementação de alguns métodos de certas classes reusam métodos de suas superclasses. Esse aspecto é bastante interessante para a extensibilidade da bibliotecas. Complementando o que foi dito anteriormente, se, por exemplo, quiséssemos estendê-la para tratar a mídia áudio, seriam necessárias novas classes com classe **nó áudio** e uma **lista de nós áudio**. Essas classe reusariam componentes da classe **nó texto** e da classe **lista**.

Devido a independência entre a implementação do modelo e sua interface gráfica, a biblioteca de classes pode ser executada em qualquer plataforma que possua um compilador compatível com o C++ da AT&T. Foi feito um teste de compilação do módulo HipClass usando um computador PC-compatível e o compilador Turbo C++ da Borland, que mostrou que a biblioteca pode ser tranportada para este ambiente. Entretanto, para que o usuário (não programador) possa dispor de seus recursos, uma nova interface precisa ser construída.

A biblioteca de classes está bem documentada. Em [4] estão descritos a especificação de requisitos, os modelos das classes e objetos e o código fonte bastante comentado. Algumas poucas alterações foram feitas no HipClass e não estão na documentação citada acima.

5.5.2 Mudanças na Interface

A portabilidade do sistema HIP é um pouco menor que a da biblioteca de classes. O Hip é construído usando rotinas das bibliotecas XView e Xlib. Assim, o sistema apenas pode ser transportado para ambientes que ofereçam tais bibliotecas.

Entretanto, pode-se migrar para outros ambientes e construir uma nova interface usando ferramentas que esse novo ambiente ofereça. É importante enfatizar, que a troca da ferramenta apenas implica em alteração do módulo HipVisão. O gerenciador - HipGer - não usa serviços da XView ou Xlib e, portanto, têm maior portabilidade. Esses aspectos estão mostrados na 5.10.

Conforme visto na seção 5.4 o HIP é composto por vários submódulos. Esses submódulos estão associados aos objetos do MCA e, consequentemente, às classes do HipClass. Assim, existe independência entre a funcionalidade de cada módulo. Esse fato é importante para a extensibilidade e modificabilidade do sistema.

Extensões ao HIP que se tornem necessárias devido a extensões à biblioteca de classes podem ser implementadas facilmente sem grandes modificações nas rotinas já existentes.

Caso sejam feitas extensões no HipClass para incorporar a mídia gráfico, será necessário construir um novo submódulo para permitir a visualização dos nós gráficos. Essas rotinas

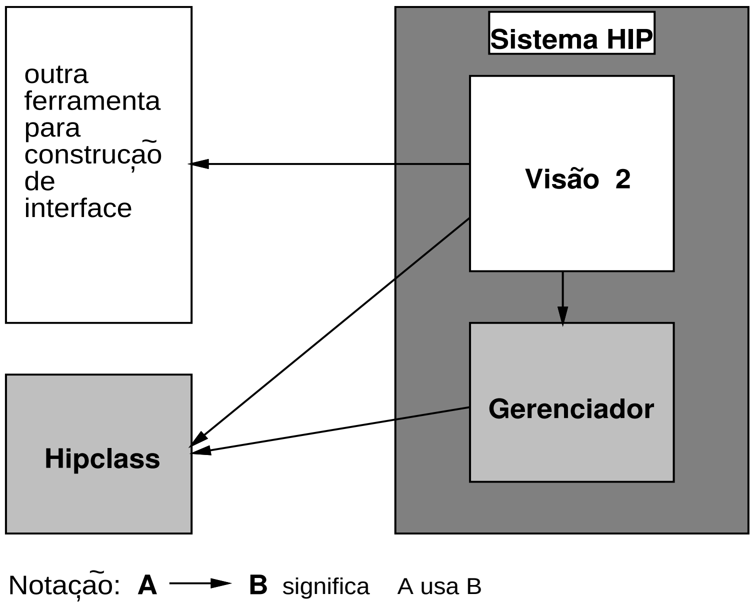

Figura 5.10: Usando outra ferramenta para construção de interfaces

podem, muitas vezes, reusar rotinas de outros submódulos do HipVisão. A parte básica do novo módulo, i.e., as janelas, menus, botões, ícones, etc., são construídas de maneira análoga às outras rotinas.

As modificações em rotinas do HIP, embora não possua as facilidades dos sistemas orientados por objetos, estão modularizados segundo critérios que o tornam bastante independentes. Cada submódulo está associado a um tipo de objeto visual, e dentro de cada módulo, as funções foram projetadas visando o máximo de independência funcional e encapsulamento.

O sistema HIP está parcialmente documentado. O código fonte possui bastante comentários e está modularizado de maneira a facilitar a sua compreensão.

5.5.3 Tornando o HIP Multi-Usuário

Futuramente, o sistema deverá tornar-se multi-usuário, oferecendo maior suporte a trabalho cooperativo. O projeto do sistema já antecipou uma arquitetura básica que possibili-

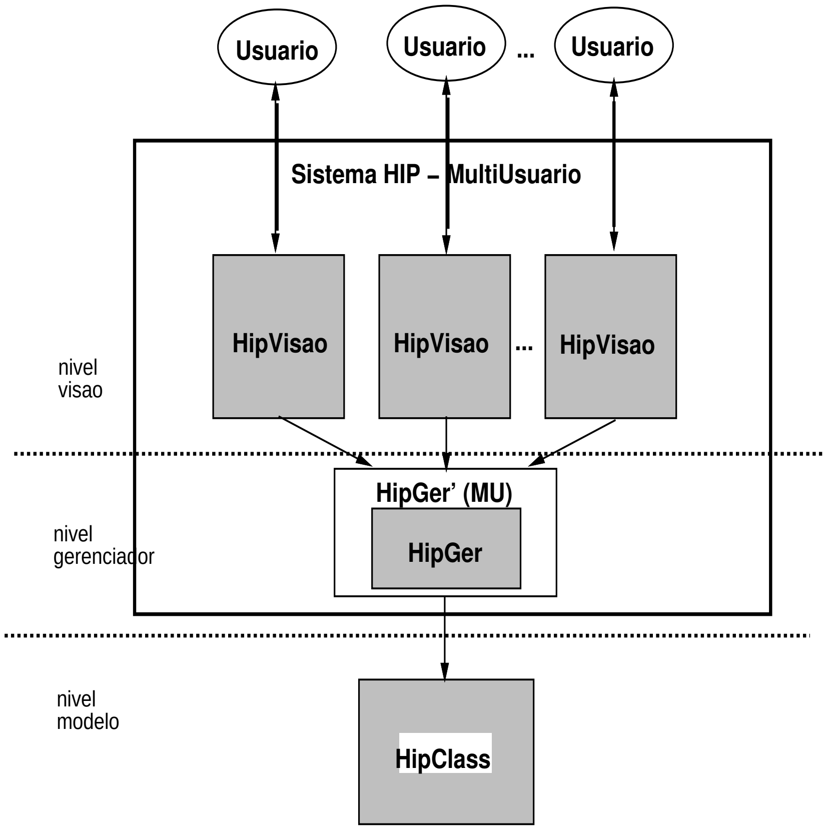

Figura 5.11: Tornando o Sistema HIP multi-usuário

tasse tais extensões. A arquitetura em camadas do HIP, apresentada na figura 5.4, possui um nível que gerencia as solicitações provenientes da interface com usuário e repassa para a biblioteca de classes.

Num sistema multi-usuário, como mostra a figura 5.11 cada usuário estaria interagindo com uma instância do módulo HipVisão da mesma maneira que no sistema mono-usuário. Cada módulo HipVisão repassaria suas solicitações para o nível gerenciador. É nesse nível que estão as principais mudanças.

O módulo gerenciador - HipGer - deverá ser incrementado para dar suporte às solicitações de cada usuário. Ele precisaria implementar um escalonador de tarefas, controlando cada pedido dos usuários, de maneira a evitar inconsistências nos objetos do MCA.

O HipGer também será o responsável em determinar como cada interface é notificada quando houver alterações realizadas pelos usuários. No sistema atual, o HipVisão sempre sabe quando um usuário está modicando um objeto do MCA. Entretanto, com vários usuários, um determinado módulo HipVisão precisa ser avisado pelo HipGer que um outro módulo HipVisão solicitou alterações ao HipClass, para que ele possa atualizar a sua representação dos objetos do MCA.

Capítulo 6

Conclusões

6.1 Introdução

Neste trabalho foi apresentada uma extensão do modelo de contextos aninhados, que oferece um modelo de versões para documentos multimídia. Pode-se comprovar mais uma vez a funcionalidade e versatilidade da estrutura de contextos, visto que foi simples modelar o grafo de história de versões usando o conceito de contextos introduzido no modelo básico [31]. Foram descritos mecanismos para suportar a criação cooperativa de documentos e a propagação de versões. Finalmente, foi proposta uma estratégia de Notificação de Mudança que leva em conta os requisitos das aplicações multimídia.

Um protótipo do modelo de contextos aninhados incluindo controle de versões baseado nos conceitos propostos foi implementado na forma de uma biblioteca de classes extensível, utilizando a linguagem C++ e o sistema operacional SunOS das estações Sun. Sob esta biblioteca de classes foi desenvolvida uma ferramenta para autoria e navegação em documentos hipermídia, utilizando o XView Toolkit [23] na construção da interface.

A escolha da linguagem e da plataforma de implementação do protótipo foi realizada tendo em vista possibilitar futuras extensões do modelo e o desenvolvimento de aplicações sobre ele.

A idéia do projeto do Modelo de Contextos Aninhados é tornar operacional um protótipo do modelo incluindo gerenciamento de versões, persistência dos objetos e conformidade com o padrão MHEG. Como a implementação de um protótipo com tais características é uma tarefa muito extensa, evolvendo a definição de muitas questões, o trabalho foi dividido em duas partes complementares. Uma consistindo da implementação da porção relativa à base privada do sistema com gerenciamento de versões. A outra porção consiste na implementação da base privada, compreendendo o mecanismo de persistência dos seus objetos, obedecendo as normas fixadas pelo padrão MHEG.

O presente trabalho encarregou-se de implementar a parte que trata do gerenciamento de versões, não houve preocupação com a persistência dos objetos.

Paralelamente, foi implementado um protótipo de um sistema mono-usuário e sem controle de versões oferecendo persistência aos objetos e obedecendo as normas fixadas pelo padrão MHEG para intercâmbio de objetos multimídia. A descrição deste protótipo se encontra em [11].

Na seção 6.2 serão analisadas as contribuições que esse trabalho veio a proporcionar.

A seguir, seção 6.3, traz sugestões de possíveis continuidades a serem dadas ao presente trabalho. Este tópico compreende questões referentes à extensão do modelo de contextos aninhados e do protótipo que o implementa.

6.2 Contribuições da Dissertação

Dentre as contribuições relevantes do presente trabalho, destaca-se a Proposta de Notificação de Mudança que constitui um aspecto essencial para preservar consistência de versões durante o trabalho cooperativo.

Outro ponto importante é a implementação do protótipo do MCA com controle de versões, que vem possibilitar extensões do modelo e consequentemente permitir uma avaliação das suas propostas. Ou seja, espera-se um *feedback* das aplicações, avaliando se as propostas do modelo atendem as suas necessidades.

Durante a especificação do protótipo houve uma constante preocupação com a sua extensibilidade. Com isso, conseguiu-se definir uma arquitetura com módulos bem independentes funcionalmente para facilitar a extensão em várias direções. Assim, o protótipo permite tanto a extensão da biblioteca de classes de forma a tratar outras mídias além da mídia texto, como a construção de várias aplicações sobre ele e ainda sua ampliação de forma a torná-lo multi-usuário.

O projeto do Modelo de Contextos Aninhados envolve um grupo de pessoas trabalhando em seu desenvolvimento. Com isso, existe uma grande preocupação em divulgar as novas possibilidades que surgem no final de cada etapa percorrida. No capítulo 5 foi dado ênfase a aspectos de evolução futura do protótipo implementado. Apresentou-se as tarefas a serem realizadas para adicionar ao protótipo os aspectos abordados acima. Essa descrição é primordial quando trabalha-se com trabalho cooperativo, pois é necessário fornecer subsídios aos demais componentes do projeto, de forma a evitar o desperdício de tempo na compreensão do sistema quando for necessário a sua extensão.

Ainda sobre as contribuições do protótipo, pode-se ressaltar a validação dos conceitos introduzidos pelo modelo.

6.3 Trabalhos Futuros

Como esta dissertação compreende a descrição do modelo conceitual de contextos aninhados com suporte a versões e a implementação de um protótipo deste modelo conceitual, em termos de trabalhos futuros, pode-se dividir os pontos em aberto em duas classes:

- A nível do modelo conceitual
- A nível de implementação

6.3.1 A nível do Modelo Conceitual

1. Introduzir *Estruturas Virtuais* (nós, elos e âncoras) que são estruturas que resultam da avaliação de uma expressão, sendo criadas apenas no momento da navegação. Nós virtuais têm seu conteúdo e âncoras computadas quando são selecionados durante a navegação. A âncora virtual será computada quando um elo que aponta para ela for atravessado. Um elo virtual terá seus pontos destino computados quando selecionado.

2. Definir uma *Linguagem de Consulta* para acessar os objetos usando os valores dos seus atributos como no CHS [20].

3. Definir uma *Linguagem para Navegação Automática*

6.3.2 A nível de Implementação

1. O primeiro problema a ser resolvido é a inclusão a persistência dos objetos. Esse passo corresponde a junção com o protótipo implementado em [11]. Este protótipo foi desenvolvido sobre o AIX versão 3.2 em estações IBM RISC System 6000, usando a linguagem C++. A interface foi construída usando o OSF/Motif 1.1. Apesar dessas duas porções terem sido desenvolvidas em plataformas diferentes, não há o comprometimento da compatibilidade entre as mesmas, uma vez que foram especificados em conformidade com o MCA, seguindo o paradigma de orientação por objetos, desenvolvidos usando a mesma linguagem de programação e preocupando-se bastante com a questão de portabilidade. As interfaces foram construídas usando ferramentas diferentes, mas para incluir persistência não se faz necessário absorver a interface, é preciso apenas usar a biblioteca de classes onde estão implementados os aspectos referentes à persistência dos objetos.

2. Adicionar na biblioteca de classes do Modelo de Contextos Aninhados os aspectos referentes trabalho cooperativo, como a Hiperbase Pública, os mecanismos de Notificação de Alteração e de Propagação de Versões. Como estes pontos já estão definidos e, muitas decisões de implementação foram descritas, não deverá ser dispendido muito esforço na implementação. No caso particular da Hiperbase, é necessário implementar as primitivas de versões descritas na subseção 3.5.2.1.

3. Incluir os novos conceitos introduzidos no último aperfeiçoamento do modelo de contextos aninhados. Com a crescente evolução do modelo, o desenvolvimento do protótipo não acompanhou a última etapa de alteração do modelo conceitual, pois esta se deu recentemente quando o escopo desse trabalho já se encontrava delimitado e o protótipo concluído. Este fato não representa um grande problema, uma vez que essa recente extensão apenas refinou os conceitos anteriores, não comprometendo a coerência dos aspectos implementados no protótipo. Assim, este protótipo pode ser facilmente adaptado à nova realidade através do reuso da biblioteca de classes disponível.

Bibliografia

[1] Ahuja, S.; Ensor, J. & Horn, D. "The Rapport Multimedia Conferencing System" ACM SIGOIS, vol. 9, Julho 1988.

[2] Bancilhon, F.; Kim, W. & Korth, H. "A Model of CAD Transactions", Proceedings of International Conference on Very Large Data Bases, Agosto 85, 25-33.

[3] Banerjee, J. et alli. "Data Model Issues for Object-Oriented Applications". ACM Transactions on Office Information Systems, Vol. 5, No. 1, January 1987, pages 3-26.

[4] Batista, T. V. "Sistema de Autoria em Hiperdocumentos". Projeto Final de Programação do Departamento de Informática da PUC-Rio, Dezembro de 1992.

[5] Batory, D. & Kim, W. "Modeling Concepts for VLSI CAD Objects. in ACM Transactions on Database Systems, Vol. 10, No. 3, September 1985.

[6] Bigelow, J. "Hypertext and Case" in IEEE Software, Março 88.

[7] Casanova, M.A.; Lima, M.J.D.; Rangel, J.L.; Rodriguez, N.; Soares, L.F.G.; Tucherman, L. "The Nested Context Model for Hyperdocuments", Proceedings of Third Conference on Hypertext, Texas, Dezembro 1991.

[8] Campbell, B. & Goodman J. "HAM: A General Purpose Hypertext Abstract Machine", In Communications of the ACM, Vol. 31, No. 7, Julho 88, 856-861.

[9] Chou, H. & Kim, W. "A Unifying Framework for Version Control in a CAD Environment", Proceedings of the Twelfth International Conference on Very Large Data

Bases, Agosto 1986, 336-344.

[10] Colcher, S. & Ridolfi, L. "Notificação de Alteração para Contextos Aninhados", Comunicação Pessoal, março 1992.

[11] Colcher, S. "Uma Arquitetura Aberta para Sistemas Hipermídia.", Dissertação de Mestrado do Departamento de Informática da PUC-RJ, Agosto 1993.

[12] Conklin, J. "Hypertext: An Introduction and Survey". IEEE Computer, Setembro 1987.

[13] Delisle, N. & Schwartz, M. "Neptune: a Hypertext System for CAD Applications", ACM SIGMOD, Vol. 15, No. 2, Junho 1986, 132-142.

[14] Delisle, N. & Schwartz, M. "Contexts - A Partitioning Concept for Hypertext" ACM Transactions on Office Information Systems, Vol. 5, No. 2, Abril 1987, 168-186.

[15] Dias, E. & Magalhães, G. "MVC: Um Modelo para Controle de Versões e Configurações", Anais do V Simpósio Brasileiro de Engenharia de Software, Ouro Preto, Outubro 91, 93-106.

[16] Dittrich, K. & Lorie, R. "Version Support for Engineering Database Systems" IEEE Transactions on Software Engineering, Vol. 14, No. 4, Abril 1988.

[17] Ducreux, F. X. "Temps et Versions dans une approche Orientée Objet", Note Technique - NOT006, Junho 1990.

[18] Fauvet, M. C. "Modelling and Managing Versions and Histories in an Object Oriented Environment". Aristote Report - RAP015, Julho 1991.

[19] Golendziner, L. G. "Um estudo sobre versões em Bancos de Dados Orientados a Objetos". Relatório de Pesquisa No. 213 do Instituto de Informática da UFRGS, março 93.

[20] Haake, A. "Cover: A Contextual Version Server for Hypertext Applications." ACM ECHT Conference. Dezembro 92.

[21] Halasz, F. "Reflections on Notecards: Seven Issues for the Next Generation of Hypermedia Systems", Communications of the ACM, Vol. 31, Number 7, Julho 88.

[22] Halasz, F. G. & Schwartz, M. "The Dexter Hypertext Reference Model", in Proceedings of the NIST Hypertext Standardization Workshop, Fevereiro 1990, 95-133.

[23] Heller, D. The Definitive Guide to the X Window System, volume 7 - XView Programming Manual. O'Reilly Associates, Inc., 1991.

[24] ISO/IED JTC1/SC29/WG12 Multimedia and Hypermedia Information coding Expert Group (MHEG). ISO/IEC CD 13522-1 - Coded Representation of Multimedia and Hypermedia Information Objects (MHEG), Part I: Base Notation (ASN.1), Junho, 1993.

[25] Jarwa, S. & Bruandet, M. F. "An Object-Oriented Model for Hypertext Database: Application to Document Management in Software Engineering", Rapport Aristote - RAP004, Janvier 1990.

[26] Katz, R.; Chang, E. & Batheja, R. "Version Modelling Concepts for Computer Aided Databases" ACM SIGMOD, Maio 86, 28-30.

[27] Katz, R.; Bhateja, R.; Chang, E.; "Design Version Management", IEEE Design & Test, Fevereiro 1987, 12-21.

[28] Katz, R. "Toward a Unified Frameword for Version Modeling in engineering Databases", ACM Computing Surveys, Vol. 22, No. 4, Dezembro 1990, 375-408.

[29] Kim, W.; Baloou, N.; Garza, J. & Woelk, D. "Distributed Object-Oriented Database System Supporting Shared and Private Databases", ACM Transactions on Information Systems, Vol.9, No. 1, Janeiro 1991, 31-51.

[30] Lévy, P. As Tecnologias da Inteligência. Editora 34, 1a. Edição, 1993.

[31] Lima, M.J.D. "O Modelo de Contextos Aninhados para Documentos Multimídia: Definição e Implementação", Dissertação de Mestrado do Departamento de Informática da PUC-RJ, 1992.

[32] Lima, M.J.D & Cavalcanti, M.R., "Tratamento de Versões de Documentos Multimídia", a ser publicado no VII Simpósio Brasileiro de Banco de Dados, Porto Alegre, 1992.

[33] Meyrowitz, N. "Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework" OOPSLA Proceedings, Setembro, 1986, 186-201.

[34] Nye, A. The Definitive Guide to the X Window System, volume 1 - Xlib Programming Manual. O'Reilly Associates, Inc., 1990.

[35] Osterbye, K. "Structural and Cognitive Problems in Providing Version Control for Hypertext". ACM ECHT Conference, December 1992, pg 33-42.

[36] Papadimitriou, C. & Kanellakus, P. "On Concurrency Control by Multiple Versions", ACM Transactions on Database Systems, vol. 9, Março 1984, 89-99.

[37] Parsaye, K. et alli. "Intelligent Databases: Object-Oriented, Deductive and Hypermedia Technologies"; John Wiley & Sons, Inc. 1989.

[38] Rumbaugh, J. & alii. Object-Oriented Modeling and Design. Ed. Prentice Hall, New York, 1991.

[39] Sciore, E. "Using Annotations to Suport Multiple Kinds of Versioning in an Object-Oriented Database System" ACM Transactions on Database Systems, Vol. 16, No. 3, Setembro 1991, 417-438.

[40] Smith, J. & Weiss, S. "Hypertext", ACM Communications, Vol.31, N0. 7, Julho 88.

[41] Soares, L. F. & Casanova, M. A. "Uma Arquitetura Distribuída para o Modelo de Contextos Aninhados com Intercâmbio de Objetos MHEG", Comunicação Pessoal,

Dezembro 92.

[42] Soares, L. F.; Casanova, M. A. & Rodriguez, N. " Tratamento de versões em um Modelo Conceitual Hipermídia com Nós de Composição", Comunicação Pessoal, Março 93.

[43] Soares, L. F.; Rodriguez, N. & Casanova, M. A. "An Open Hypermedia System with Nested Composite Nodes and Version Control", Comunicação Pessoal, Dezembro 93.

[44] Tepedino, J.; Albuquerque, E. & Meira, S. "Gerenciamento de Objetos em Sistemas de Hipertexto: Uma Proposta", Anais do V Simpósio Brasileiro de Engenharia de Software, Ouro Preto, Outubro 91.

[45] Tichy, W. "RCS - A System for Version Control", Software Practice and Experience, Vol. 15, Julho 1985, 637-654.

[46] Utting, K. & Yankelovich, N. "Context and Orientation in Hypermedia Networks", ACM Trans. on Information Systems, Vol. 7, 1990, 58-84.

[47] Wiil, U. K. et alii. "Design and Implementation of a HyperBase". Technical Report of University of Aalborg, September 1990, Denmark.

[48] Woelk, D. & Kim, W. "Multimedia Information Management In an Object-Oriented Database System", Proceedings of the 13th VLDB Conference, 1987, 319-329.

CONTROLE DE VERSÕES NO MODELO HIPERMÍDIA DE CONTEXTOS ANINHADOS

Dissertação de Mestrado apresentada por **Thaís Vasconcelos Batista**, em 18 de março de 1994 ao Departamento de Informática da PUC-RJ, e aprovada pela Comissão Julgadora formada pelos seguintes professores:

____

Luiz Fernando Gomes Soares (Orientador)
Departamento de Informática - PUC/RJ

____

Marco Antônio Casanova
Centro Científico Rio - IBM Brasil

____

Noemi De La Rocque Rodriguez
Departamento de Informática - PUC/RJ

____

José Lucas Mourão Rangel Neto
Departamento de Informática - PUC/RJ

Visto e permitida a impressão
Rio de Janeiro, 18 de março de 1994

____

Prof. LUIZ FERNANDO GOMES SOARES
Coordenador de Programas de Pós-Graduação do
CENTRO TÉCNICO CIENTÍFICO