

Modelo de Contextos Aninhados - Um Modelo Conceitual Hipermídia

Luiz Fernando G. Soares

Depto. de Informática, PUC-Rio
R. Marquês de São Vicente 225
22453-900 - Rio de Janeiro, RJ
Brasil

E-mail: lfgs@inf.puc-rio.br

Noemi L. R. Rodriguez

Depto. de Informática, PUC-Rio
R. Marquês de São Vicente 225
22453-900 - Rio de Janeiro, RJ
Brasil

E-mail: noemi@inf.puc-rio.br

Marco Antonio Casanova

Centro Científico Rio, IBM Brasil
Caixa Postal 4624
20701-001 - Rio de Janeiro, RJ
Brasil

E-mail: casanova@vnet.ibm.com

Resumo

Este artigo apresenta um modelo conceitual hipermídia, que entre outras características, suporta grupo de versões, permite a exploração e gerenciamento de configurações alternativas, mantém o histórico do documento, suporta trabalho cooperativo e provê propagação automática de mudanças em versões. O conceito de contexto de versões é usado para agrupar nós que representam a mesma versão de um objeto em algum nível de abstração. O suporte a trabalho cooperativo é baseado nas noções de bases privadas e hiperbase pública. Propagação automática de versões usa o conceito de perspectiva corrente para limitar a proliferação de versões. Todas as facilidades propostas tem como objetivo minimizar o *overhead* cognitivo imposto ao usuário quando da manipulação do documento. Embora os mecanismos de controle de versões discutidos no artigo tomem por base o Modelo de Contextos Aninhados, as idéias se aplicam a qualquer modelo conceitual hipermídia com nós de composição que admitam aninhamento.

Abstract

This paper presents a conceptual model for hypermedia that, among other features, supports versions sets, permits exploring and managing alternate configurations, maintains document histories, supports cooperative work and provides for automatic propagation of version changes. The concept of version context is used to group together nodes that represent versions of the same object at some level of abstraction. Support for cooperative work is based on the idea of public hyperbase and private bases. The automatic propagation of versions uses the concept of current perspective to limit the proliferation of versions. All of the proposed facilities have as a goal the minimization of the cognitive overhead imposed on the user by document manipulation. The version control discussion is phrased in terms of the nested context model, but the major ideas apply to any hypermedia conceptual model that offers nested composite nodes.

Palavras chave: hipermídia, controle de versões, trabalho cooperativo.

1 - Introdução

Serviços multimídia têm sido incorporados em várias áreas, como sistemas de educação e treinamento, sistemas para automação de escritório, sistemas de informação e pontos de vendas, etc. Neste contexto, muitas aplicações multimídia serão construídas para plataformas heterogêneas, e muitas aplicações oferecerão seus serviços através da interconexão de plataformas diversas. Estes serviços usarão uma grande quantidade de objetos estruturados residentes em estações, armazenados em meios digitais, ou recuperados de fontes remotas através de redes. Uma vez que estes objetos representarão um investimento significativo, torna-se imperativo que esta informação permaneça disponível e não seja perdida pela incompatibilidade de estruturas de dados usadas nas diversas aplicações e, desta forma, que os objetos possam ser reusados.

Os sistemas hipermídia existentes foram desenvolvidos como uma aplicação única e auto-contida, fazendo com que sejam especializados e difíceis de serem reutilizados em outras aplicações. Muitos dos esforços empregados na construção de sistemas hipermídia seguiram esta direção, com exceções dignas de menção, como o Neptune [04], HyperBase [15], MultiCard [14], Hyperform [21] e HyperProp [17]. HyperProp provê não apenas um modelo conceitual de dados hipermídia, o Modelo de Contextos Aninhados, como uma arquitetura aberta, cujo modelo de interface separa os componentes de dados e de exibição dos objetos, permitindo a construção de interfaces independentes da plataforma final de exibição. Estes dois componentes combinados vão fornecer uma ferramenta capaz não só de possibilitar a construção de sistemas hipermídia monolíticos, como também de introduzir características hipermídia em uma aplicação qualquer. Eles vão permitir ainda que os mecanismos de armazenamento possam ser adaptados aos requisitos de eficiência e tamanho impostos por uma aplicação particular. Uma descrição detalhada da arquitetura do sistema HyperProp pode ser encontrada em [17].

Um tópico que não foi coberto na descrição original do Modelo de Contextos Aninhados [03] foi o de controle de versões. Embora seja reconhecidamente considerado um tópico de grande relevância, sua complexidade tem aparentemente adiado a apresentação de trabalhos na área. Neste artigo nós descrevemos o Modelo de Contextos Aninhados estendido, que, entre outras características, provê suporte a grupo de versões, permite a exploração e gerenciamento de configurações alternativas, mantém a história do documento, suporta trabalho cooperativo e provê propagação automática de mudanças de versões. Todas as facilidades para manipulação de versões foram projetadas de forma a impor um *overhead* cognitivo mínimo ao usuário. Cabe também aqui salientar que o sistema para tratamento de versões que aqui propomos é adequado a qualquer sistema hipermídia com nós de composição com aninhamento, em particular àqueles baseados na proposta de padrão MHEG [12], caso do Modelo de Contextos Aninhados. A mesma observação é válida com respeito ao suporte a trabalho cooperativo proposto no modelo.

O presente artigo se concentra na apresentação do modelo conceitual de dados do sistema HyperProp, isto é, o Modelo de Contextos Aninhados (MCA). A próxima seção descreve o modelo básico, deixando para a seção 3 a descrição do modelo estendido com a incorporação dos mecanismos de controle de versões. Trabalhos relacionados ao modelo descrito são discutidos na seção 4. A seção 5 é reservada às conclusões.

2 - O Modelo de Contextos Aninhados

O objetivo da construção do sistema HyperProp é fornecer um ambiente para a construção de aplicações hipermídia através de uma biblioteca de classes que reflitam seu modelo conceitual. A descrição que se segue é, assim, uma descrição destas classes e sua funcionalidade, sem a facilidade de controle de versões, que reservamos para a seção 3.

2.1 - O Modelo Básico

A possibilidade de representar referências arbitrárias entre partes quaisquer de um documento representa, para muitos, a característica marcante de um sistema hipertexto. No entanto é necessário combinar esta flexibilidade com mecanismos de organização do documento, a fim de evitar o problema conhecido como "lost in hyperspace" [08], mecanismos para definição de diferentes visões de um mesmo documento e mecanismos para organização hierárquica (linear ou não) de documentos. Os modelos com nós de composição provêm suporte a tais mecanismos, em especial modelos que permitem o aninhamento destes nós.

A definição de documentos hipermídia no MCA [03] é baseada nos conceitos usuais de nós e elos. *Nós* são fragmentos de informação e *elos* são usados para a interconexão de nós que mantêm alguma relação.

O modelo distingue duas classes básicas de nós, chamados de nós terminais e nós de composição, sendo estes últimos o ponto central do modelo. A figura 1 ilustra a hierarquia de classes proposta.

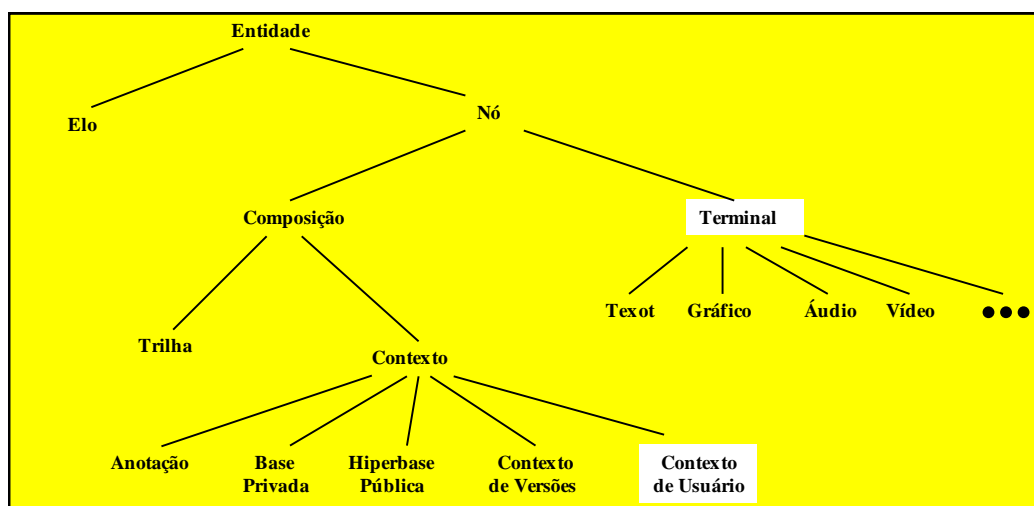


Figure 1 - Hierarquia de Classe do MCA

Uma *entidade* permite que pares atributo/valor sejam associados a qualquer objeto. Cada entidade possui um identificador único (UID) e uma lista de controle de acesso (ACL). Cada entrada nesta lista associa um usuário, ou grupo de usuários, aos seus direitos de acesso a cada atributo do objeto. Cada entidade possui também uma *especificação de apresentação*. Embora esta especificação não seja usada pelo modelo conceitual estrutural, como no modelo Dexter [10] ela contém informações para o modelo de apresentação sobre como a entidade deve ser exibida ao usuário.

Intuitivamente, um *nó terminal* contém dados cuja estrutura interna é dependente da aplicação e não é parte do modelo. A classe de nós terminais pode ser especializada em outras classes (texto, gráfico, áudio, vídeo, etc.) conforme requerido pelas aplicações.

Um nó de *composição* agrupa entidades, chamadas *componentes*, incluindo outros nós de composição. Os componentes podem ser ordenados, o que será bastante útil em alguns mecanismos de navegação que veremos a posteriori. Os componentes não formam necessariamente um conjunto, podendo uma entidade ser incluída mais de uma vez em um nó de composição. A classe de nós de composição pode ser especializada em outras classes, incluindo as classes dos nós de contexto e trilhas.

Um *nó de contexto* é usado para agrupar um conjunto de elos, trilhas, nós terminais e de contexto, recursivamente. Os componentes de um nó de contexto C formam necessariamente um conjunto, isto é, um componente A é incluído apenas uma vez no nó de contexto. Diz-se que A está contido em C . Note que isto não impede que um componente A de um nó de contexto C seja incluído em um nó de composição B também contido em C , uma vez que a relação de "estar contido" não é recursiva. A classe de nós de contexto pode ser especializada em outras classes, incluindo as classes contexto de versões, anotação, bases privadas, hiperbase pública e contexto de usuário. As quatro primeiras classes dão suporte ao controle de versão e trabalho cooperativo, não fazendo parte do modelo básico, e serão discutidas na seção 3.

Um *nó de contexto de usuário* é usado para agrupar um conjunto de elos, nós terminais e de contexto de usuário, recursivamente. Como todo nó de composição, nós de contexto de usuário podem ser aninhados em qualquer profundidade, permitindo organizar, hierarquicamente ou não, conjuntos de nós. Por exemplo, para organizar hierarquicamente um livro texto, pode-se primeiro definir um nó de contexto de usuário B , que contém um conjunto de nós representando capítulos do livro e um conjunto de elos indicando a organização dos capítulos, que não é necessariamente em sequência linear. De forma análoga, para o i -ésimo capítulo, pode-se definir o nó de contexto de usuário C_i que contém um conjunto de nós representando as seções do capítulo e um conjunto de elos indicando a organização das seções, e assim sucessivamente.

Um elo conecta dois nós. Como o conteúdo de um nó possui uma estrutura interna, que pode ser bastante complexa, os elos também indicam, para cada extremidade, a região onde o elo "toca" o nó. Por exemplo, para nós texto, a região pode ser simplesmente uma cadeia de caracteres dentro do texto, para nós de imagem bidimensional, pode ser um retângulo determinado por um par de coordenadas, e para nós de contexto, pode especificar uma região dentro de um de seus componentes, conforme será definido nos próximos parágrafos.

A noção informal de região é capturada no modelo pela noção de *âncora*. Cada nó N tem um conjunto de âncoras, que especificam regiões dentro do conteúdo de N , dependente da classe de N . As âncoras de um nó funcionam como uma interface externa do nó, no sentido que outros nós não se referirão diretamente a pontos dentro do conteúdo do nó, mas apenas indiretamente através de um identificador de âncora. Assim, qualquer mudança no conteúdo de um nó não se refletirá nos elos e, portanto, nos nós de contexto que contêm estes elos, se as âncoras de um nó forem sempre atualizadas adequadamente.

Em conformidade com o modelo Dexter, uma âncora é um par $(id, valor)$. O identificador id é único dentro de um nó M onde a âncora é definida, chamado de *base* da âncora. Como mencionado anteriormente, é impossível descrever o que vem a ser uma âncora para um nó genérico. Classes diferentes de âncoras podem ser definidas para as diferentes classes de nós. O

sistema HyperProp não interpreta a representação interna de uma âncora, exceto no caso de nós de contexto, como veremos mais adiante. Contudo, toda classe de âncora tem um valor especial, denotado por λ , representando a ancoragem no nó, como um todo, onde a âncora é definida.

Dada a noção intuitiva, pode-se definir agora, mais precisamente, elo e nó de contexto de usuário.

Um elo contém uma sequência de *pontos terminais* (s, d_1, d_2, \dots) onde s é a *fonte* e d_i é o *destino* do elo. Os pontos terminais consistem de um caminho de nós (N_k, \dots, N_2, N_1) e uma âncora dentro do nó N_1 . O caminho é uma sequência de identificadores de nós, tal que o nó N_{i+1} é um nó de contexto e N_i está contido em N_{i+1} , para $i \in [1, k)$. N_k é chamado de *base* do elo. Um elo é sempre unidirecional, embora possa ser seguido em ambas as direções.

Embora não faça parte do modelo conceitual da estrutura hipermídia, mas parte do modelo de apresentação, os elos têm também associados um conjunto de *especificações de apresentação de nós*, como no modelo Dexter, que contém informação para a apresentação, ao usuário, dos nós de destino referenciados.

A cada elo são associadas condições e ações, em conformidade com a proposta de padrão MHEG [12]. Generalizando, uma vez que o modelo de contextos aninhados engloba todos os objetos em conformidade com a proposta de padrão MHEG, ele também vai englobar todas as características e definições destes objetos. Como extensão ao padrão MHEG, a cada âncora de um nó também são associadas condições e ações [19, 20].

Um *nó de contexto de usuário* C é um nó de contexto que agrupa:

1. Um conjunto S de nós terminais ou de contexto de usuário. Diz-se que nós em S estão contidos em C .
2. um conjunto de elos L , tal que cada elo l em L tem os nós bases contidos em C . Os elos de L também são ditos estarem contidos em C .

Para cada nó em S existe um atributo onde uma *especificação de apresentação do nó* pode ser definida.

No caso específico de um nó de contexto C (em particular um nó de contexto de usuário C), o valor de uma âncora pode ser o valor λ , representando todo o nó; um subconjunto dos nós contidos em C ; ou um caminho de nós (N_k, \dots, N_2, N_1) e uma âncora dentro do nó N_1 . Neste último caso, o caminho é uma sequência de identificadores de nós, tal que o nó N_{i+1} é um nó de contexto, N_i está contido em N_{i+1} , para $i \in [1, k)$, e N_k está contido em C .

Por exemplo, seja A um nó de contexto de usuário que contém um outro nó de contexto de usuário B que por sua vez contém dois nós, C e D . Como B contém C e D , um elo conectando estes nós pode, em princípio, ser definido em B como $\langle C, i \rangle, \langle D, j \rangle$, onde i e j são âncoras válidas para C e D , respectivamente. Se se quer criar um elo em A conectando C e D , ele deve ser definido como $\langle (B, C), i \rangle, \langle (B, D), j \rangle$. Note a diferença entre a definição do elo em A e B . O elo definido em B será visto por todo documento que contenha B , enquanto que o elo definido em A será visto apenas pelos documentos que contenham A . Alternativamente, pode-se definir o elo em A como $\langle B, m \rangle, \langle B, n \rangle$. Neste caso, m e n identificam âncoras de B , cujos valores são iguais a $\langle C, i \rangle$ e $\langle D, j \rangle$. As várias maneiras de se especificar um elo permitem uma maior

flexibilidade na especificação de como um nó deve ser apresentado ao usuário, tornando esta especificação dependente da navegação até o nó.

Continuando as definições, uma hiperbase é qualquer 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 à hiperbase H .

Concluindo, o MCA também permite extensões para acomodar a noção de estruturas virtuais (conteúdos de nós, âncoras e elos), isto é, estruturas que resultam da avaliação de alguma expressão. Um nó virtual pode ter seu conteúdo e âncoras computados quando de sua seleção durante um processo de navegação. Em particular uma âncora pode ser computada quando um elo que a aponta for selecionado. De forma similar um elo pode ter seus pontos terminais computados quando selecionado. A importância destes conceitos nos mecanismos de controle de versões são discutidos na seção 3.

2.2 - Algumas Considerações Sobre o Modelo de Apresentação

O modelo conceitual estrutural do MCA trata o hiperdocumento multimídia como uma estrutura de dados passiva. Um sistema hipermídia deve, no entanto, fornecer ferramentas para que o usuário possa ter acesso à estrutura da rede, exibí-la, manipulá-la e navegar por ela. Esta funcionalidade é capturada pelo modelo de apresentação. Nesta seção alguns pontos do modelo de apresentação são salientados. Uma definição mais precisa do modelo de apresentação pode ser encontrada em [20].

Uma vez que o modelo permite que um mesmo nó esteja presente em diferentes composições, e que composições sejam aninhadas em qualquer profundidade, é necessário que exista uma forma de identificar sob que sequência de composições aninhadas um dado nó está sendo pesquisado e que elos de fato tocam este nó a partir do aninhamento. Estas noções são capturadas pela noção de perspectiva de um nó.

Uma perspectiva de um nó N é uma sequência $P = (N_1, \dots, N_m)$, $m \geq 1$, tal que $N_1 = N$, N_{i+1} é uma composição e N_i está contido em N_{i+1} , para $i \in [1, m)$. Diz-se que N_i está *contido recursivamente* em N_{i+1} , N_{i+2} , ..., N_m , para $i \in [1, m)$. Como N é implicitamente dado por P , P é referido simplesmente como perspectiva. Um nó M está presente em uma perspectiva $P = (N_1, \dots, N_m)$ se e somente se $M = N_i$ para algum $i \in [1, m]$. Note assim que podem existir várias perspectivas diferentes para um mesmo nó N , se este nó estiver presente em mais de uma composição. Chama-se perspectiva corrente a perspectiva usada na última navegação para alcançar o nó.

A interação de um usuário com um hiperdocumento no MCA é modelada em conformidade com o modelo Dexter. O conceito fundamental neste modelo é o de *instantiation*: uma apresentação do componente ao usuário. No modelo Dexter uma função chamada *instantiator* é responsável por uma instanciação, dado o componente e sua especificação de apresentação. Instâncias são *versões* no MCA e existem dentro de uma *work session* do modelo Dexter, que são modeladas pelas *bases privadas* do MCA, conforme será visto posteriormente.

A noção de especificação de apresentação já foi mencionada na seção 2.1. Uma especificação de apresentação contém informações para a apresentação de uma entidade. Em particular ela

define métodos para exibição ou edição de entidades. Estes métodos podem ser qualquer programa, em particular, qualquer editor. O modelo provê métodos específicos para a edição de atributos do sistema, e para exibição e edição do conteúdo dos nós de contexto.

As especificações de apresentação podem ser armazenadas como um atributo das entidades, ou, alternativamente, como um nó terminal. Na apresentação de um nó, a especificação de apresentação definida explicitamente por um usuário se sobrepõe àquela definida pelo nó de contexto de usuário (como visto na definição deste nó) que contém o nó, que se sobrepõe à especificação de apresentação definida pelo elo usado para atingir o nó, que por sua vez se sobrepõe àquela definida dentro do nó. Cada tipo de nó tem uma especificação de apresentação *default*, usada quando nenhuma das especificações mencionadas acima for definida.

Cada aplicação hipermídia requer formas específicas de navegação, que podem ser programadas utilizando primitivas de apresentação oferecidas pelo HyperProp. O sistema, no entanto, oferece suporte para alguns tipos de navegação.

Chama-se *navegação em profundidade* a ação de seguir o aninhamento de nós de composição, o que permite ao usuário subir e descer na hierarquia de composições. A *navegação por elos*, essencial à noção de hipermídia, também é fornecida. O modelo também provê suporte à *navegação por consulta*: um usuário pode identificar um nó específico descrevendo propriedades que este nó deve satisfazer.

Exibidores (*browsers*) para nós de contexto apresentam uma visão pictorial do hiperdocumento (ou parte dele). *Navegação em browsers* é uma outra forma pré-definida de navegação no MCA.

O sistema fornece também suporte à *navegação em uma trilha*, possibilitando a usuários seguir uma trilha previamente estabelecida em uma sessão de trabalho anterior. Trilhas são especializações dos nós de composição que contêm uma lista ordenada de nós (e suas perspectivas correspondentes), incluindo trilhas. Um nó pode estar na lista em mais de uma posição.

Depois de ativar uma trilha para navegação, um usuário pode acionar o comando *next* que automaticamente selecionará o próximo item da lista ordenada de componentes. O comando *previous* seleciona o componente anterior, enquanto o comando *home* seleciona o primeiro componente da lista. Trilhas são úteis, entre outras coisas, na linearização de um hiperdocumento. Da mesma forma que no sistema Intermedia [22], uma trilha especial — *system private base trail* — grava toda a navegação realizada durante uma sessão de trabalho, de forma que o usuário pode mover-se aleatoriamente de nó a nó e retornar, passo a passo. Note que esta é uma das maneiras de se criar uma trilha.

3 - Controle de Versões

Esta seção segue a seguinte organização. Alguns requisitos, que os mecanismos de controle de versão para sistemas hipermídia devem obedecer, são apresentados em 3.1. Na subseção 3.2, o Modelo de Contextos Aninhados é estendido de forma a incluir controle de versões. Nesta subseção é mostrado como o modelo estendido atende aos requisitos levantados em 3.1.

3.1 - Alguns Requisitos de Versões em Sistemas Hipermídia

Foram adotadas como métrica para o mecanismo de controle de versões incorporado ao MCA as considerações feitas em [08, 09], [13] e em [07]. Analisando principalmente a primeira referência, pode-se identificar os seguintes requisitos, transcrevendo para cada um a frase original onde ele é colocado.

- R1. Exploração de Configurações Alternativas — *"A good versioning mechanism will also allow users to simultaneously explore several alternate configurations for a single network"*.
- R2. Gerenciamento de Configurações — *"In a software engineering context it should be possible to search for either the version that implements Feature X or the set of changes that implement Feature X"*.
- R3. Manutenção do Histórico do Documento — *"A good versioning mechanism will allow users to maintain and manipulate a history of changes to their network"*.
- R4. Atualização Automática de Referências — *"In particular, a reference to an entity may refer to a specific version of that entity, to the newest version of that entity along a specific branch of the version graph, or to the (latest) version of the entity that matches some particular description (query)"*.
- R5. Suporte a Tratamento de Grupos de Versões — *"Although maintaining a version thread for each individual entity is necessary, it is not a complete versioning mechanism. In general, users will make coordinated changes to a number of entities in the network at one time. The developer may then want to collect the resultant individual versions into a single version set for future reference"*.

Analisando [13], pode-se acrescentar os seguintes requisitos:

- R6. Pequeno Overhead Cognitivo na Criação de Versões — *"Explicit version creation will result in a large cognitive overhead. How can version creation be made manageable?"*.
- R7. Imutabilidade de Versões — *"In hypertext it might be too simplistic to have versions of nodes to be completely immutable. While it is obvious that the contents of a version should be immutable, it is less clear how links and (other) attributes should be treated"*.
- R8. Versões de Elos — *"One must also consider a separate versioning for links"*.
- R9. Versões da Estrutura — *"It is desirable to have a notion of versions of the structure of the hypertext. Similarly, being able to return to a previous state of the entire hypertext is just as desirable as returning to a state of a single node"*.
- R10. Suporte ao Desenvolvimento Exploratório — *"If we want to support exploratory development the problem is how to freeze a state. The entire structure the author is working with needs to be frozen"*.

Analisando [07], pode-se acrescentar os seguintes requisitos:

R11. Adaptabilidade — "*Versioning must be tailorable by applications*".

R12. Suporte a Documentos Alternativos — "*It must be possible to maintain alternatives. Reviewers and authors wants to maintain explicit alternatives of sub-parts of a document*".

Além dos pontos abordados acima, discutiremos neste trabalho ainda dois outros, que julgamos pertinentes na definição de um sistema distribuído para tratamento de documentos multimídia:

R13. Representações Distintas da Mesma Informação — é importante que o sistema tenha como capturar a noção de que diferentes representações, em particular, em diferentes mídias, podem ser usadas para descrever uma mesma informação (como, por exemplo, a versão escrita e falada de uma palestra). Representações diferentes de uma mesma informação devem ser tratadas como diferentes versões desta informação.

R14. Uso Concorrente da Mesma Informação — cópias (temporárias) da mesma informação em uso em determinado instante devem ser consideradas versões desta informação. O acoplamento desta noção de versão com o mecanismo de notificação vai proporcionar um bom suporte a trabalhos cooperativos. Na verdade este caso é uma generalização do caso anterior.

Finalmente, pode-se observar que, com o propósito de diminuir o espaço de armazenamento, ao invés de se armazenar cada elemento de um grupo de versões, apenas as diferenças entre estes elementos poderiam ser armazenadas: um mecanismo conhecido como técnica delta. A desvantagem desta técnica é que a recuperação de uma versão específica vai requerer computação — partindo-se do primeiro elemento e incluindo as mudanças. Ainda outro problema vem do fato que algoritmos delta devem ser específicos para cada mídia (o mesmo algoritmo não será eficiente tanto para texto quanto para áudio, por exemplo). Nós acreditamos que técnicas para economia de memória, tais como representar apenas as diferenças entre objetos ao invés do todo, são tarefas do sistema de armazenamento, através de seus algoritmos de armazenamento, e não devem fazer parte dos modelos de dados. Desta forma a técnica delta não será considerada um requisito do mecanismo de controle de versões.

3.2 - O Modelo de Contextos Aninhados Estendido

O modelo de contextos aninhados básico já satisfaz o requisito R1 para mecanismos de controle de versões. De fato, o problema de explorar configurações alternativas de um documento é endereçado através da criação de nós de contexto de usuário alternativos, contendo o mesmo conjunto de nós, mas refletindo diferentes visões do mesmo documento sintonizadas para aplicações ou classes de usuários distintas. Retornando ao exemplo da seção 2.1 sobre a estrutura de um livro, os autores poderão definir organizações diferentes para os capítulos, orientadas para classes específicas de leitores, através da criação de nós de contexto de usuário diferentes contendo os mesmos nós (os mesmos capítulos), mas com diferentes conjuntos de elos. Cada um destes nós de contexto de usuário oferecerá uma visão diferente (uma leitura especializada) do mesmo livro.

Para acomodar os outros requisitos listados na seção 3.1, a hierarquia básica do MCA necessita ser estendida com novas entidades para versionamento e trabalho cooperativo. A Figura 1 descreve resumidamente as extensões que propomos, exploradas nas seções 3.2.1 a 3.2.5.

Quatro novas subclasses da classe contexto são introduzidas: anotação, hiperbase pública, base privada e contexto de versões.

No MCA, apenas os nós terminais e nós de contexto de usuário estão sujeitos a versionamento, como salientado pelo fundo branco na figura 1. Além disso, cada atributo (incluindo o conteúdo) de um nó terminal ou contexto de usuário pode ser especificado como versionável ou não. Um atributo não versionável pode ter seu valor modificado sem a necessidade da criação de uma nova versão. Ao contrário, modificações no valor de um atributo versionável devem ser realizadas em uma nova versão do objeto, se este já estiver no estado permanente, como será detalhado na seção 3.2.2. Atributos versionáveis e não versionáveis contribuem para o atendimento do requisito R7. Obviamente, algum tipo de mecanismo de notificação será necessário, como extensão ao mecanismo de suporte a versões, especialmente no caso de atualizações concorrentes de atributos não versionáveis.

A possibilidade de se adicionar novos atributos a um nó sem a criação de novas versões é bastante atraente. Suponha, por exemplo, que após um nó ser criado, uma nova ferramenta é introduzida no sistema. A ferramenta pode querer guardar algumas informações específicas nos nós. Em sintonia com esta possibilidade, no MCA o usuário pode especificar se a adição de novos atributos é permitida sem a criação de novas versões do objeto (contribuindo assim no atendimento a R7).

Concluindo, versionamento de elos não foi incluído no modelo, uma vez que acreditamos que esta facilidade adiciona mais complexidade do que funcionalidade ao sistema, e além disso, se necessário, pode ser modelada pelo versionamento dos nós de contexto de usuário. Resta porém avaliar se esta extensão tornar-se-ia relevante quando elos carregam ações e condições.

3.2.1 - Contexto de Versões

Para equacionar o problema de manutenção da história de um documento, estendemos o modelo de contextos aninhados com uma classe especial de nós de contexto, chamados de contextos de versões.

Os nós de um contexto de versões V são chamados de versões correlatas e não precisam pertencer à mesma classe (o que ajuda a atender R13). Um elo da forma $(\langle v_1, i_1 \rangle, \langle v_2, i_2 \rangle)$ em V indica que v_2 é derivada de v_1 . Os elos de V não estão sujeitos a restrições (o que ajuda a atender R12), exceto que o grafo induzido por eles deve ser acíclico. Os nós em V representam então versões do mesmo objeto, no mesmo nível de abstração, sem que necessariamente uma versão seja derivada da outra, e os elos de V capturam explicitamente o relacionamento de derivação. As âncoras de um elo $(\langle v_1, i_1 \rangle, \langle v_2, i_2 \rangle)$ em V servem apenas para indicar com mais precisão quais partes de v_1 geraram quais partes de v_2 .

Note que um nó de contexto de versões pode conter nós de contexto de usuário, uma vez que tais nós também estão sujeitos a versionamento. Desta forma, o modelo provê um mecanismo para versionar a estrutura de um documento, atendendo ao requisito R9.

Um usuário poderá manualmente adicionar novos nós a um contexto de versões V , para indicar explicitamente que tais nós são versões do mesmo objeto, e novos elos, para indicar como as versões foram derivadas. Ele também poderá alterar V através de operações específicas de versionamento.

Uma aplicação poderá definir, de várias formas, um nó em um contexto de versões V como a sua versão corrente, escolhida de acordo com um critério específico. No caso mais simples, a aplicação poderá reservar uma âncora de V para manter uma referência à sua versão corrente em V . Outras âncoras podem especificar outras versões de acordo com outros critérios de seleção.

No modelo estendido, a aplicação poderá se valer de consultas para localizar dinamicamente a sua versão corrente. A consulta poderá ser armazenada em uma âncora ou definida em um elo fora do contexto de versões (ver seção 2.1), ou mesmo especificada através de mecanismos mais gerais (auxiliando a atingir R6 e R4), conforme discutiremos na seção 3.2.3. Note que tal consulta poderá em certos casos retornar mais de uma versão (como por exemplo, "retorne todas as versões criadas por João"), que deverão ser interpretadas como alternativas e apresentadas dentro de um nó de contexto de usuário. Assim, contextos de versões atendem R4, provendo um mecanismo de atualização automática de referências.

No modelo básico MCA, o ponto terminal de um elo contido em um contexto de usuário C , e de forma similar um possível valor de âncora de um contexto de usuário C , foram definidos como um par consistindo de uma lista de identificadores dos nós (N_k, \dots, N_2, N_1) e uma âncora α , tal que:

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

No modelo estendido, N_1 deve ser ou um nó terminal, ou um nó de contexto de usuário (como no modelo básico), ou um nó de contexto de versões, quando então não dizemos que N_2 contém N_1 , mas sim que contém o nó especificado pela âncora α .

Uma aplicação poderá usar contextos de versões para manter a história de um documento d (R3), bem como para resolver o problema de atualizar referências automaticamente (R4), por exemplo, da seguinte forma. Suponha que d tenha um componente c em cujas versões a aplicação está interessada. Seja C o contexto de versões contendo os nós que representam as versões de c . A aplicação deverá então se referenciar a C , e não diretamente a quaisquer das versões de c , no nó de contexto de usuário que modela d . Todos os elos em D tocando C apontarão para a mesma âncora de C , que por sua vez sempre apontará para o nó que a aplicação considerar como sendo a versão corrente de c . Se a aplicação quiser recuperar versões anteriores de d com respeito a c , ela simplesmente navegará em C . De fato, como contextos de versões são apenas uma classe especial de nós de contexto, os usuários poderão em princípio navegar pelo histórico de um documento utilizando os mecanismos genéricos de navegação do MCA [03, 19]. A aplicação poderá obter alternativas para o sub-documento c , por exemplo, através de uma consulta (retornando um ou mais nós), em conformidade com o requisito R12.

Contextos de versões também ajudam a atender o requisito R2. De fato, recorde inicialmente que em Engenharia de Software versionamento é possível em dois níveis. O nível mais baixo corresponde a versionar os diferentes módulos que compõem os programas. Naturalmente, todas as versões de um módulo podem ser agrupadas em um contexto de versões. O outro nível cobre as chamadas configurações, que são descrições da forma de compor consistentemente versões de módulos para formar programas. Uma configuração pode ser modelada por um nó de contexto de usuário cujos nós correspondem às versões dos módulos associados à configuração e cujos elos capturam as dependências entre as versões dos módulos. Modelar

configurações através de nós de contexto de usuário é bastante natural, pois configurações diferentes podem compartilhar as mesmas versões de módulos, da mesma forma que nós de contexto de usuário podem compartilhar os mesmos nós, mas o relacionamento entre as versões dos módulos é específico de cada configuração, como os elos são privativos de um nó de contexto de usuário. Naturalmente configurações podem ser interpretadas como versões de um mesmo objeto e agrupadas também em um contexto de versões, ajudando a equacionar assim o problema de gerência de versões de configuração, coberto pelo requisito R2.

Um conjunto de versões correntes específicas incluídas em uma configuração é frequentemente chamada de *baseline*. Note que o simples armazenamento de uma configuração não indica como o sistema evoluiu no tempo, pois o critério de seleção de uma versão corrente pode retornar versões diferentes em tempos diferentes. Desta forma, torna-se importante podermos armazenar uma configuração estática, onde cada referência é feita a uma versão específica e não a um contexto de versões. Retornaremos a este ponto na seção 3.2.3 quando da discussão do conceito de base privada.

3.2.2 - Consistência

Incluímos no nosso modelo a noção de estado de um nó terminal ou um nó de contexto de usuário para facilitar o controle de consistência entre nós, o suporte a trabalho cooperativo e a criação automática de versões (veja também a seção 3.2.3).

Um nó terminal ou um nó de contexto de usuário N pode estar em um dos seguintes estados: *permanente*, *temporário* e *obsoleto*. Um nó é criado no estado temporário e permanece neste estado enquanto está sendo modificado. Quando se torna estável, o nó pode ser promovido ao estado permanente explicitamente através de uma operação do usuário ou implicitamente como efeito colateral de certas operações do modelo (o que ajuda a atender R6). Como um exemplo de mudança de estado implícita, temos que um nó temporário pode se tornar permanente quando uma primitiva para criação de versão é a ele aplicada. Um nó permanente não pode ser diretamente removido, mas o usuário pode torná-lo obsoleto, permitindo que outros nós que o referenciem, ou dele derivados, sejam notificados.

O conceito de estado de um nó é relevante apenas para nós terminais ou de contexto de usuário que têm atributos versionáveis. Assim, quando dizemos, por exemplo, que nós no estado permanente não podem ser modificados, queremos dizer que os atributos versionáveis não podem ser modificados. Salientamos também que, como mencionado anteriormente, a adição de novos atributos em um nó no estado permanente pode ser realizada sem, necessariamente, a criação de uma nova versão do nó.

Mais precisamente, um nó terminal ou de contexto de usuário no estado permanente, chamado de um *nó permanente*, possui as seguintes características:

- não pode ter seus atributos versionáveis modificados (o que significa que os atributos explicitamente definidos não podem ser modificados, bem como as consultas que o definem, se o nó é virtual);
- só pode conter nós permanentes ou obsoletos, se for um nó de contexto de usuário;
- pode ser usado para derivar outros nós (isto é, versões);
- não pode ser diretamente removido;
- pode se tornar obsoleto, mas não pode se tornar temporário novamente.

Um nó terminal ou de contexto de usuário no estado temporário, chamado de um *nó temporário*, possui as seguintes características:

- todos os seus atributos podem ser modificados;
- pode conter nós em qualquer estado, se for um nó de contexto de usuário;
- não pode ser usado para derivar outros nós (isto é, versões);
- pode ser diretamente removido;
- pode se tornar permanente, mas não pode se tornar obsoleto.

Um nó terminal ou de contexto de usuário no estado obsoleto, chamado de um *nó obsoleto*, possui as seguintes características:

- não pode ter nenhum de seus atributos modificados (o que significa que os atributos explicitamente definidos não podem ser modificados, bem como as consultas que o definem, se o nó é virtual);
- só pode conter nós permanentes ou obsoletos, se for um nó de contexto de usuário;
- não pode ser usado para derivar outros nós (isto é, versões);
- é automaticamente removido pelo sistema, através de um processo de coleta de lixo, quando não é mais necessário (por exemplo, quando não é mais referenciado nem está incluído em nenhum outro nó);
- não pode mais mudar de estado.

Pode-se deduzir das restrições acima que, se uma versão V for direta ou transitivamente derivada de W , então W ou é permanente ou obsoleta. Pode-se também concluir que um nó de contexto de usuário N que é permanente ou obsoleto só contém nós permanentes ou obsoletos, o que por sua vez implica em que: (i) N só contém elos cujos extremos são nós permanentes ou obsoletos; (ii) se N é virtual, a consulta que define o seu conteúdo só retorna nós permanentes ou obsoletos e as consultas nos seus elos e âncoras sempre resultam em um conjunto de nós permanentes ou obsoletos. Não existem restrições análogas para os nós temporários.

As definições de estado de um nó podem parecer excessivamente restritivas à primeira vista. Por exemplo, poder-se-ia considerar interessante permitir que novos nós fossem derivados de um nó temporário. No entanto, esta possibilidade tornaria ainda mais difícil para o sistema garantir a consistência do histórico das versões. Nada impede porém que uma aplicação implemente uma modificação da operação de criação de versões tal que a criação de uma nova versão de um nó N temporário resulte na criação de uma nova versão do nó permanente do qual N é derivado, por exemplo.

Conforme mencionado na seção 3.2.1, o usuário poderá explicitamente criar elos representando derivação de versões. Porém, a criação de um elo deste tipo automaticamente muda o estado do objeto-fonte para permanente a fim de preservar consistência.

3.2.3 - Hiperbase Pública e Base Privada

3.2.3.1 - Definições Básicas

Entendemos por autoria cooperativa o processo de criar ou modificar a hiperbase, ou um subconjunto da hiperbase, por um grupo de usuários. De forma geral, um ambiente de autoria cooperativa deve permitir que usuários compartilhem informação, oferecer alguma forma de

criar informação privada, e permitir a fragmentação do espaço global de trabalho em frações menores para reduzir o espaço de navegação.

A noção de nó de contexto oferece um ponto de partida para organizar autoria cooperativa. Considere que o conjunto de todos os nós de contexto de usuário e nós terminais está particionado em vários subconjuntos. Um e apenas um deles forma a *hiperbase pública*, denotada por H_B , que corresponderá à informação pública e estável. Os outros subconjuntos formam as bases privadas, usadas na modelagem da interação do usuário com o hiperdocumento, de acordo com o paradigma de sessão de trabalho proposto pelo Modelo Dexter. Uma base privada pode conter outras bases privadas, o que permite organizar a sessão de trabalho em várias subsessões aninhadas. Note que uma versão específica de um nó terminal ou de contexto de usuário pode pertencer a apenas uma destas bases, incluindo-se aqui a hiperbase pública.

Mais precisamente, uma *hiperbase pública* é um nó de contexto que agrupa nós terminais e nós de contexto de usuário. Todos os nós em H_B devem ser permanentes ou obsoletos e, como em toda hiperbase, se um nó de contexto de usuário C está em H_B , então todos os nós contidos em C devem também estar contidos em H_B .

Uma *base privada* é também definida como um nó de contexto, que agrupa qualquer entidade, exceto nós de contexto de versões e a hiperbase pública, tal que:

- i) uma base privada pode pertencer a no máximo uma base privada;
- ii) se um nó de composição N está contido na base privada PB , seus componentes ou estão contidos em PB , ou na hiperbase pública, ou em qualquer base privada de um aninhamento de bases privadas contido em PB ; e
- iii) se um elo está contido em uma base privada, o nó base de seu ponto terminal de origem deve ser um nó de anotação.

Intuitivamente, uma base privada agrupa todas as entidades usadas pelo usuário durante uma sessão de trabalho.

3.2.3.2 - Operações sobre Versões em Bases Privadas e na Hiperbase Pública

Um usuário pode mover um nó terminal ou de contexto de usuário de uma base privada para a hiperbase pública através da primitiva de *check-out*, desde que o nó seja permanente. Se um nó permanente de contexto de usuário C for movido para H_B , então todos os nós contidos em C devem também ser movidos para H_B .

Note que a movimentação de uma versão para a hiperbase pública só precisa ser realizada quando ocorrer alguma modificação no conteúdo original. Se após a criação de uma versão V de um nó N da hiperbase pública em uma base privada, V não sofrer qualquer modificação, V é simplesmente destruída ao ser movida para a hiperbase pública, pois não há a necessidade de replicação de nós. Os nós de composição que contêm V devem ser atualizados para conterem agora N , ao invés de V . Da mesma forma, todas as versões criadas a partir de V devem ser transformadas em versões de N no contexto de versões apropriado.

Um nó na hiperbase pública não pode migrar desta hiperbase para uma base privada, embora possam ser criadas versões destes nós nas diversas bases privadas. No HyperProp, manipular um documento implica na criação de novas versões, na base privada corrente, de todos os nós terminais e de contexto de usuário visitados. Estas novas versões podem ser derivadas de um nó

permanente, ou corresponder à criação de uma informação completamente nova (o primeiro nó em um contexto de versões). Como mencionado em 2.2, estas versões correspondem às instâncias do Modelo Dexter.

Existem duas primitivas, *open* e *check-in*, disponíveis para a criação de uma nova versão temporária de um nó terminal ou de contexto de usuário N em uma base privada PB . Estas primitivas diferem no tratamento de nós de contexto de usuário, da seguinte forma. Suponha que N seja um nó de contexto de usuário. A primitiva *check-in* cria em PB uma versão temporária N' de N , exatamente idêntica a N , ou seja, com os mesmos nós e elos. Já a primitiva *open* cria em PB uma versão temporária N' de N e, recursivamente, de cada componente de N . N' conterá as novas versões dos componentes de N , e seus elos serão criados de forma a refletir apropriadamente os elos de N . Se um componente permanente pertencer a mais de um contexto, apenas uma versão temporária será criada para este nó.

Algumas consequências interessantes decorrem do comportamento diferente das primitivas *open* e *check-in*. De fato, seja N um nó da hiperbase pública e M um nó contido recursivamente em N através de duas perspectivas. Suponha inicialmente que N' tenha sido criada de N pela primitiva *check-in*. Observe que, pela definição de *check-in*, temos que N e consequentemente M são necessariamente permanentes e que N' conterá recursivamente M exatamente como N contém. Ao atualizar M através de uma das perspectivas, o usuário criará então uma versão de M , que não será visível pela outra perspectiva. Portanto, ao tentar atualizar M pela outra perspectiva, o usuário criará uma segunda versão. Suponha agora que N' tenha sido criada através da primitiva *open*. Neste caso, N' já conterá uma (única) versão M' de M , temporária, que poderá ser atualizada por quaisquer das duas perspectivas.

Uma implementação de *open* poderá postergar a criação das versões dos componentes de N para o momento em que de fato forem atualizados através de N' .

Versões permanentes em uma base privada PB podem ser usadas para derivar novas versões, ou serem incluídas em um nó de contexto de usuário, em todas as bases privadas que contêm PB , e em todas as bases privadas que contêm estas bases, e assim recursivamente. Isto parece bastante razoável, uma vez que estes nós representam um estado de trabalho consistente sobre o ponto de vista da tarefa que está sendo realizada em alguma base privada. Versões temporárias, por outro lado, são apenas acessíveis para a manipulação na base privada onde está contida.

Um usuário pode remover um nó N de uma base privada PB através da primitiva *delete*. Se N for uma trilha, ou um nó de anotação (conceito a ser discutido mais adiante), ele é simplesmente removido da base privada e destruído. Se N é um nó terminal ou de contexto de usuário, o resultado depende do estado do nó. Se N é temporário, ele é destruído e removido do contexto de versões apropriado; se N é permanente, ele é tornado obsoleto. Quando um nó se torna obsoleto em uma base privada, ele é movido para a hiperbase pública. Se ele é um nó obsoleto de contexto de usuário, todos os seus nós componentes são também transferidos.

Uma base privada PB também pode ser removida. Neste caso, todos os seus nós, incluindo bases privadas, são também removidos, recursivamente. A base privada PB é então destruída.

3.2.3.3 - Observações Adicionais

Alguns sistemas evitam o problema cognitivo adicional de versionamento através da criação automática de versões, ou a intervalos regulares ou como efeito colateral dos comandos. Estas

duas opções podem gerar versões que não representam necessariamente um passo consistente na evolução de um trabalho. Da mesma forma que o conceito de tarefas em [07], o conceito de bases privadas ajuda a manipular versões de uma maneira mais controlada (contribuindo no atendimento de R4 e R6).

Na seção 3.2.1, foi discutido como especificar versões correntes em nós de contexto de versões. A seleção de versão corrente baseada em uma linguagem de consulta é uma técnica poderosa, mas pode aumentar o *overhead* cognitivo do usuário, que tem de definir o critério de seleção para cada elo criado. Para minimizar este problema, no MCA cada nó contexto de versões tem duas âncoras especiais, definidas por consultas *default*, para a recuperação da versão corrente. Uma destas consultas *default* é definida no próprio contexto de versões. A outra é especificada, de uma forma mais geral, em um atributo da base privada (contribuindo para o atendimento de R6 e R4). Quando um elo é criado, o nó de destino é examinado. Se o nó é um componente de um nó contexto de versões não especificado explicitamente pelo usuário (diretamente ou através de uma consulta), o elo é criado usando o critério de seleção especificado na base privada que contém o nó de contexto onde o elo está contido. Se a consulta não estiver especificada nesta base privada, o elo usará a consulta *default* definida no contexto de versões.

Qualquer dos mecanismos de navegação oferecidos pelo HyperProp podem ser usados para visitar um nó. Frequentemente a navegação é baseada em uma consulta. Se cada vez que uma consulta for resolvida, por exemplo em uma navegação em profundidade, uma nova versão é criada na base privada, a sessão de trabalho rapidamente se encherá de versões (considere o caso de um usuário navegando para baixo e para cima de uma perspectiva). Para evitar o problema, no MCA, uma vez que a consulta é resolvida, ela se torna permanentemente resolvida para aquela base (intuitivamente isto significa "para o resto da sessão de trabalho").

Quando nós são movidos de uma base privada para a hiperbase pública, o usuário pode escolher armazenar a configuração estática (com as consultas resolvidas) presente na base privada, ao invés da configuração dinâmica original, armazenada por *default* (contribuindo para o atendimento dos requisitos R5 e R9).

3.2.4 - Propagação Automática de Versões

Na presença de nós contendo outros nós, quando o usuário cria uma nova versão de um dos componentes de um nó N , o sistema pode criar automaticamente uma nova versão de N de acordo com um critério pré-estabelecido. Chamamos este mecanismo de propagação automática de versões.

Como no MCA um nó pode estar contido em diferentes nós de contexto de usuário, propagação automática de versões poderá causar a criação de um grande número de versões indesejáveis, conforme ilustrado na Figura 2. A hiperbase inicial, mostrada na Figura 2(a), possui um nó $D0$ que pertence aos nós de contexto $E0$, $B0$ and $F0$; $E0$ por sua vez está contido em $C0$ e $B0$ em $A0$. A criação da versão $D1$ de $D0$ gera cinco novos nós de contexto de usuário, conforme mostrado na Figura 2(b), se a propagação de versões for aplicada exaustivamente.

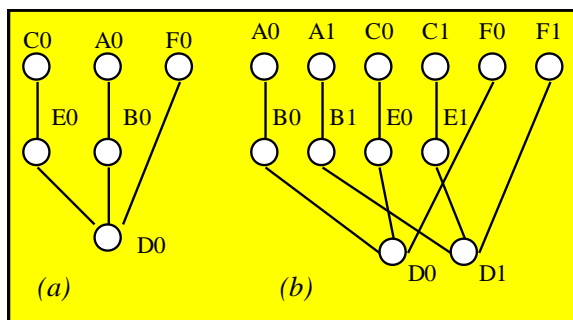


Figura 2 - Proliferação de versões pela perspectiva

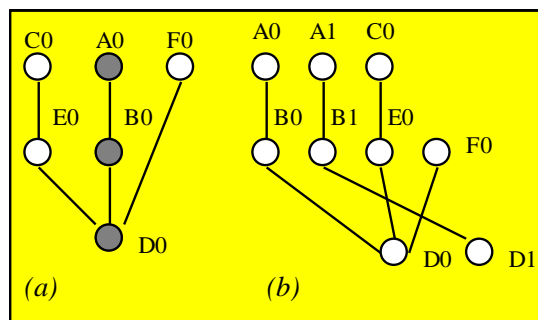


Figura 3 - Propagação guiada

Para minimizar a proliferação de versões, o MCA permite ao usuário decidir se ele quer propagação automática ou não. Além disto, o MCA limita a propagação automática aos nós permanentes de contexto de usuário que pertencem à perspectiva através da qual a nova versão está sendo construída, em linha com a restrição de que versões não podem ser derivadas de versões que não sejam permanentes. Esta estratégia provê então mais uma forma de coordenar atualizações em grupo, atingindo assim R5, R6 e R10.

A Figura 3 ilustra estes pontos. Assuma que a hiperbase inicial é a mesma usada na Figura 2(a) e que a perspectiva corrente é $(D0, B0, A0)$. Se $B0$ e $A0$ são nós permanentes, então novas versões destes nós serão criadas, como indicado na Figura 3(b). Porém, se $A0$ e $B0$ não forem permanentes, então nenhuma versão destes nós será criada, apenas $B0$ será alterada para incluir $D1$ ($A0$ não será alterada).

3.2.5 - Anotações

Anotações consistem de um comentário (em qualquer formato ou mídia) e mantêm referências às versões objeto do comentário e às versões consideradas respostas ao comentário.

O MCA define anotações como um nó de contexto que agrupa um conjunto de elos, nós terminais, nós de contextos de usuários e trilhas. Intuitivamente, elos de uma base privada podem ligar os componentes de um nó de anotação a nós da mesma base privada, indicando nós que originaram o comentário e nós de resposta aos comentários. Anotações permitem que observações sejam feitas a um nó permanente, sem a criação de novas versões, contribuindo assim para o atendimento de R7.

Para concluir toda a seção 3.2, observamos que o mecanismo de suporte a versões descrito pode ser sintonizado para aplicações específicas. Versões podem tanto ser automaticamente criadas quanto criadas através de comandos explícitos. Estas facilidades podem ser usadas pelas aplicações para a criação de várias políticas de versionamento, contribuindo para o atendimento de R11.

4 - Trabalhos Relacionados

Esta seção faz uma breve comparação entre o MCA e outros modelos conceituais hipermídia. Devido à importância dos mecanismos de versionamento, modelos que possuem tal facilidade são o alvo principal de comparação.

O MCA permite a navegação em profundidade no aninhamento de contextos e, através da noção de perspectiva, permite a herança de elos, generalizando as funcionalidades oferecidas pelos *browsers* e *fileboxes* do NoteCard [08]. *Tree items* no KMS permitem a estruturação hierárquica de documentos [01]. A relação de inclusão de um nó de contexto no MCA é muito mais geral, permitindo a estruturação hierárquica além de outros benefícios. Os contextos MCA permitem o particionamento da rede hipermídia generalizando o conceito homônimo introduzido no modelo conceitual HAM do sistema Neptune [04, 05], que por sua vez se baseou em algumas idéias do PIE [06]. O aninhamento permitido pelos nós de contexto MCA generalizam as *webs* do Intermedia [11]. Contextos HyperPro [13] e composições HyperBase (CoVer) [15] são bastante semelhantes aos contextos MCA. Não fica claro, no entanto, como aqueles sistemas endereçam o problema da perspectiva de um nó e da ancoragem em nós aninhados.

Em PIE, uma *layer* agrupa em uma unidade identificável mudanças em vários componentes. *Layers* podem ser superpostas, combinando mudanças, para compor uma versão no sistema. Como em HyperPro e CoVer, apenas o resultado final da superposição é representado nos contextos MCA. Acreditamos que as técnicas para economia de espaço em memória, armazenando apenas as diferenças, sejam tarefa para o sistema de armazenamento e não devam fazer parte do modelo de dados. Além do que, a recuperação de dados em sistemas que utilizam tal técnica sempre requer uma computação muitas vezes intolerável em sistemas interativos. Outro problema de PIE vem do fato que as *layers* podem ser superpostas arbitrariamente, o que pode resultar em combinações inconsistentes.

Na versão estendida de HAM, um elo referencia ou uma versão específica ou o elemento mais novo (no tempo) em um grupo de versões. Não existe a liberdade de se definir a versão corrente como no MCA. Um pouco mais geral que HAM, HyperPro organiza suas versões em árvores. Já CoVer e MCA, com estruturas ainda mais gerais, permitem a inclusão de elos de derivação pelo usuário, organizando suas versões em grafos acíclicos.

Tanto no MCA como em HyperPro considera-se importante a definição de um critério de seleção de uma versão, dentro de um grupo de versões, de uma forma global, diminuindo o *overhead* cognitivo do usuário na criação de elos. No HyperPro, o critério de seleção é definido no nó de contexto e é baseado apenas no tempo. No MCA, o critério de seleção, definido na base privada, é mais geral, podendo ser uma consulta não baseada apenas no tempo. Além disso, o MCA permite a definição, nas âncoras, das consultas para referência dinâmica a versões dentro de um contexto de versões. Esta facilidade é extremamente importante, pois possibilita a definição de diferentes critérios de seleção para diferentes elos em um mesmo contexto.

Os estados permanente e temporário do MCA são semelhantes às noções de "frozen" e "updatable" usadas em HyperPro e CoVer. Entretanto, estes estados são usados na tomada de várias decisões para criação automática de versões, o que não acontece de forma similar naqueles modelos. Além disso, a inclusão do estado obsoleto é bastante útil no gerenciamento do sistema.

O MCA provê várias maneiras para tornar o estado de um conjunto de nós permanente; por exemplo, ao tornar permanente um nó de contexto, todos os seus componentes se tornam permanentes. Bases privadas também podem ser movidas em bloco para a hiperbase pública [17]. Estas funções suplantam o suporte apresentado em HAM, HyperPro e CoVer para trabalho exploratório.

A imutabilidade de atributos de um nó MCA generaliza a mesma facilidade introduzida no HyperPro, onde a imutabilidade é associada à classe do nó, que por sua vez é mais geral que HAM, onde nós inteiros (e não atributos) são classificados como versionáveis ou não.

HAM e CoVer suportam versões de elos, embora não fique claro como são manipuladas. Não consideramos versões para elos em nosso primeiro protótipo, pois acreditamos que versões de contextos serão suficientes. No entanto, versões de elos talvez sejam interessantes quando os elos carregam ações, condições e informação de sincronização além de referências, como é o caso no MCA. Este é um tópico para estudos futuros.

O nó de anotação MCA é bastante semelhante ao conceito homônimo do CoVer. Anotações em CoVer são, no entanto, mais gerais. CoVer integra anotações na estrutura da *task* (conceito semelhante às bases privadas MCA); ele pode, por exemplo, registrar qual *task* produziu uma anotação, e se alguém já criou alguma *task* para trabalhar sobre uma anotação.

O MCA provê vários mecanismos para evitar a proliferação inútil de versões, baseados nos conceitos de bases privadas, nos estados de um nó, no esquema de propagação de versões e em diferentes primitivas (*open* e *check-in*) para criação de versões. No MCA e em CoVer, o sistema pode exercer um papel mais ativo na geração de novas versões do que em HAM e HyperPro. Como as *tasks* em CoVer, bases privadas no MCA podem guiar a criação automática de versões, diminuindo a carga cognitiva na criação. Não existe nada parecido com a primitiva *open* ou com os conceitos de propagação de versão em qualquer dos modelos já citados. As duas primitivas especiais para criação de versões e a propagação de versões vão ainda adicionar facilidades ao suporte para mudanças coordenadas em conjuntos de nós, suplantando o suporte oferecido pelos outros sistemas.

No MCA, quando uma consulta é resolvida na base privada *PB*, as versões resultantes se tornam o resultado permanente da consulta em *PB*. Isto vai possibilitar o armazenamento de configurações estáticas, onde cada nó é uma versão específica e não uma versão corrente (mutável) dentro de um nó de contexto de versões. Este é considerado um problema não resolvido no sistema HyperPro, e nem ao menos mencionado nos outros sistemas.

O Intermedia fornece acesso concorrente à rede hipermídia. O controle de concorrência é, no entanto, muito simples e não fornece de fato um suporte real a trabalho cooperativo. Em um ambiente onde se deseja que vários usuários trabalhem de forma cooperativa, é necessário que estes possam compartilhar informação. Por outro lado, também é importante permitir que cada usuário tenha informações não compartilhadas, não só como proteção aos seus dados como para não onerar todos os usuários com a navegação em um espaço grande demais de informações. Nenhum dos sistemas já mencionados oferecem facilidades para trabalho cooperativo como as bases privadas e hiperbase pública do MCA, que suplantam as facilidades oferecidas pelas *tasks* de CoVer.

O MCA provê o mesmo suporte para navegação em uma trilha que o sistema Intermedia. Este sistema, na realidade, foi quem influenciou as decisões tomadas no projeto do MCA com relação a criação e navegação em trilhas. O conceito de trilhas no MCA, contudo, estende o conceito homônimo Intermedia, fornecendo o aninhamento de trilhas, quando as trata como um tipo especial dos nós de composição.

Nenhum dos trabalhos citados reportam qualquer suporte para especificação de como um objeto deve ser apresentado ao usuário, nem suporte para definição de relações espaço-temporais entre objetos, tais como as definidas no MCA.

5 - Conclusão

O Modelo de Contextos Aninhados com suporte a versões é a base conceitual de uma máquina hipermídia em desenvolvimento conjunto pelo Departamento de Informática da PUC-Rio e o Centro Científico da IBM Brasil. Um protótipo mono-usuário do sistema incorporando o modelo básico já foi concluído. Um segundo protótipo, em conformidade com a proposta MHEG e incluindo versionamento, encontra-se em fase adiantada. O objetivo do projeto é criar uma ferramenta que possibilite a fácil incorporação de processamento de documentos multimídia/hipermídia a aplicações. Julgou-se, assim, que a melhor forma de oferecer esta ferramenta seria sob a forma de uma biblioteca de tipos e classes em uma linguagem orientada por objetos. A linguagem escolhida para esta implementação foi C++ por sua disponibilidade em um grande número de plataformas.

Pela falta de uma linguagem de consulta adequada, a implementação corrente não possui estruturas virtuais. De fato, sem um bom mecanismo de consulta é impossível um bom mecanismo de estruturas virtuais, e sem estes mecanismos um sistema de tratamento de versões não é completo. Estes são os principais pontos que pretendemos atacar na próxima extensão do modelo.

Suporte a trabalho cooperativo sempre foi um dos requisitos do projeto, que começou com o tratamento de documentos em sistemas de teleconferência. Mecanismos para controle de notificação, necessários para trabalho cooperativo, devem também fazer parte da próxima extensão do modelo.

Resta mencionar mais uma vez ao fim deste artigo que os mecanismos para tratamento de versões aqui apresentados, embora tenham tomado como exemplo o Modelo de Contexto Aninhado, podem ser adaptados a qualquer modelo que contenha nós de composição que possam ser aninhados, tais como HyperBase [15] e HyperPro [13].

Agradecimentos:

Os autores gostariam de agradecer a Guido Souza, Maria Júlia Lima, Paulo Jucá, Sérgio Colcher e Thaís Batista pela suas contribuições nas idéias aqui apresentadas. O trabalho de implementação que eles conduziram junto com outros estudantes permitiu o refinamento contínuo do Modelo de Contextos Aninhados.

REFERENCES

- [01] Akscyn, R.M.; McCracken, D.L.; Yoder, E.A. "KMS: A Distributed Hypermedia System for Managing Knowledge in Organizations". *Communications of ACM*, Vol.31, No. 7. Junho de 1988.
- [02] Campbell, B.; Goodman, J.M. "HAM: A General Purpose Hypertext Abstract Machine". *Communications of the ACM*. Vol. 31, No. 7. Julho de 1988, pp. 856-861.
- [03] Casanova, M.A.; Tucherman, L.; Lima, M.J.; Rangel Netto, J.L. Rodriguez, N.R.; Soares, L.F.G. "The Nested Context Model for Hyperdocuments". *Proceedings of Hypertext '91*. Texas. Dezenbro de 1991.

- [04] Delisle, N.; Schwartz, M. "Neptune: A Hypertext System for CAD Applications". *Proceedings of ACM SIGMOD '86*. Washington, D.C. Maio de 1986.
- [05] Delisle, N.; Schwartz, M. "Context - A Partitioning Concept for Hypertext". *Proceedings of Computer Supporteed Cooperative Work*. Dezembro de 1986
- [06] Goldstein, I.; Bobrow, D. "A Layered Approach to Software Design". *Interactive Programming Environments*. McGraw Hill, pag. 387-413. Nova York. 1987.
- [07] Haake, A. "Cover: A Contextual Version Server for Hypertext Applications". *Proceedings of European Conference on Hypertext, ECHT'92*. Milano. Dezembro de 1992.
- [08] Halasz, F.G. "Reflexions on Notecards: Seven Issues for the Next Generation of Hypermedia Systems". *Communications of ACM*, Vol.31, No. 7. Julho de 1988.
- [09] Halasz, F.G. "Seven Issues Revisited". *Final Keynote Talk at the 3rd ACM Conference on Hypertext*. San Antonio, Texas. Dezembro de 1991.
- [10] Halasz, F.G.; Schwartz, M. "The Dexter Hypertext Reference Model". *NIST Hypertext Standardization Workshop*. Gaithersburg. Janeiro de 1990.
- [11] Meyrowitz, N. "Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework". *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*. Portland, Oregon. Setembro de 1986.
- [12] MHEG. "Information Technology - Coded Representation of Multimedia and Hypermedia Information Objects - Part1: Base Notation. *Committee Draft ISO/IEC CD 13522-1*. Julho de 1993.
- [13] Osterbye, K. "Structural and Cognitive Problems in Providing Version Control for Hypertext". *Proceedings of European Conference on Hypertext, ECHT'92*. Milano. Dezembro 1992.
- [14] Rizk, A.; Sauter, L. "MultiCard: An Open Hypermedia System". *Proceedings of European Conference on Hypertext, ECHT'92*. Milano. Dezembro 1992.
- [15] Schütt, H.A.; Streitz, N.A. "HyperBase: A Hypermedia Engine Based on a Relational Database Management System". *Proceedings of European Conference on Hypertext, ECHT'90*. 1990.
- [16] Soares, L.F.G.; Casanova. "Modelo de Contextos Aninhados com Intercâmbio de Objetos MHEG em Arquiteturas Distribuídas". *Anais do XI Simpósio Brasileiro de Redes de Computadores*. Campinas, São Paulo, Brazil. Maio de 1993.
- [17] Soares, L.F.G.; Casanova, M.A.; Colcher, S. "An Architecture for Hypermedia Systems Using MHEG Standard Objects Interchange". *Proceedings of the Workshop on Hypermedia and Hypertext Standards*. Amsterdam, The Netherlands. Abril de 1993.
- [18] Soares, L.F.G.; Casanova, M.A.; Rodriguez, N.L.R. "Um Modelo Conceitual HiperMídia com Nós de Composição e Controle de Versões". *Simpósio Brasileiro de Engenharia de Software*. Rio de Janeiro, Brazil. Outubro de 1993.
- [19] Soares, L.F.G.; Casanova, M.A.; Rodriguez, N.L.R. "An Open Hypermedia System with Nested Composite Nodes and Version Control". *Relatório Técnico PUC-Rio - Departamento de Informática*. Rio de Janeiro. Novembro de 1993.
- [20] Sousa, G.L.; Soares, L.F.G.; Casanova, M.A.; Colcher S.; Sousa, C.S. "HyperProp Presentation Model". *Research Report Departamento de Informática, PUC-Rio*. Rio de Janeiro, Brasil. Submetido.
- [21] Wiil, U.K.; Leggett, J.J. "Hyperform: Using Extensibility to Develop Dynamic, Open and Distributed Hypertext Systems". *Proceedings of European Conference on Hypertext, ECHT'92*. Milano. Dezembro de 1992.
- [22] Yankelovich, N.; Meyrowitz, N. "Reading and Writing the Electronic Book". *IEEE Computer*. Outubro de 1985.