Nested Composite Nodes and Version Control in Hypermedia Systems

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 4524
20701-001 - Rio de Janeiro, RJ Brasil
E-mail: casanova@vnet.ibm.com

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 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 version 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.

1 - Introduction

Many application domains, such as education, training, office, business, and sales, have seen an explosion of multimedia services in the last few years. In this context, many multimedia applications will be designed to run on heterogeneous platforms, or to be interconnected to offer more sophisticated multimedia services. These services will use large quantities of structured multimedia objects, which can be either locally stored on a workstation, or retrieved from remote sources through a communication network. Since this multimedia data may represent a significant investment, it becomes vital to ensure that this information is not lost due to incompatibilities in data structures supported by the different applications.

However, most hypermedia systems have been developed as self-contained applications, preventing interoperability, information interchange, and code reusability between applications. Some exceptions must be mentioned in this context, such as the Neptune system [DeSc85], HyperBase [ScSt90], MultiCard [RiSa92], Hyperform [WiLe92] and HyperProp [SoCC93]. HyperProp provides not only a conceptual hypermedia data model, the Nested Context Model, but also an open architecture, with an interface model which separates the data

and object exhibition components. Among other advantages, this allows the constructions of interfaces to be independent of the exhibition platform, as well as the adaptation of the storage mechanism to the performance and bandwidth requirements of particular applications. This architecture is described in [SoCC93].

One issue which was not covered in the original description of the Nested Context Model [Casa91] was version control. Even though the need for this facility in hypertext systems has long been recognized, the complexity of its interaction with all the other requisites in this kind of application has apparently postponed the work in the area.

We then describe in this paper an extension of the Nested Context Model which, among other features, supports version sets, permits exploring and managing alternate configurations, maintains document histories, supports cooperative work and provides automatic propagation of version changes. The facilities we propose for version manipulation are designed so as to impose a minimum of cognitive overhead on the user. Although the version control discussion is phrased as an extension to the Nested Context Model, the major ideas apply to any hypermedia conceptual model offering nested composite nodes, such as HyperBase [ScSt90] and HyperPro [Oste92].

This paper is organized as follows. Section 2 reviews the Nested Context Model. Section 3 extends the model to support versioning and cooperative work. Section 4 compares our model with related work. Finally, section 5 contains the conclusions.

2 - The Nested Context Model

The goal of the construction of the Hyperprop system is to provide an environment for the construction of hypermedia applications, through a library of classes which reflect the conceptual model. The following description of the basic Nested Context Model is thus a description of these classes and their functionality, without version control.

The definition of hypermedia documents in the Nested Context Model (NCM) [Casa91] is based on two familiar concepts, namely nodes and links. *Nodes* are fragments of information and *links* interconnect nodes into networks of related nodes.

The model goes further and distinguishes two basic classes of nodes, called *terminal* and *composite* nodes, the latter being the central concept of the model. Figure 1 illustrates the is-a hierarchy proposed.

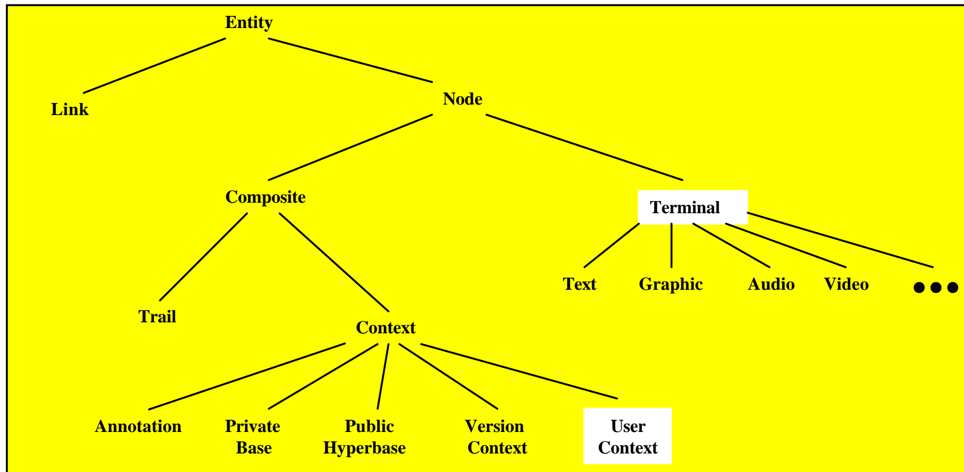

Figure 1 - NCM Class Hierarchy

An *entity* (refer to figure 1) allows attribute/value pairs to be attached to all objects. Every entity has a unique identifier (UID) and an access control list (ACL). Each entry in the ACL associates a user or user group to his access rights for each attribute. Each entity also has an *entity presentation specification.* Although it is not used by the structural conceptual model, the entity presentation specification, as in the Dexter model [HaSc90], contains information for the presentation model about how an entity should be presented to the user. Links also have a set of *node presentation specifications,* as in the Dexter model, which contains information for the presentation model indicating how a referenced node should be presented to the user.

Each node has a set of *anchors* that acts as the external interface of the node. That is, other entities will actually indirectly refer to regions inside the content of the node by identifying the appropriate anchors. In conformance to the Dexter Model, every anchor has an associated *id* and *value*. Anchors encapsulate the definition of regions. Changes to the content of a node can thus be transparent to links that touch the node, if the anchors are adequately defined.

A *link* contains a source end point and a sequence of destination, or target, end points. Multiple destination end points allow the definition of one-to-many connections, which is intended to support applications where, for example, the selection of a link can lead to the simultaneous exhibition of several nodes.

Each of the end points of a link is defined by a (possibly unary) list of nodes (N_k,..., N_2, N_1) and one anchor, which must belong to the set of anchors of N_1. The node N_{i+1} must be a context node and N_i must be contained in N_{i+1}, for all $i \in [1,k)$. The node N_k is called a *base* of the link. Links are always directional, although they can be followed in either direction. The list of nodes in each end point allows the definition of links connecting information not directly contained in the same composite.

A link has also associated to it conditions and actions, in conformance with the MHEG proposal [MHEG93]. Similar to links, as an extension of the MHEG proposal, the Nested Context Model associates conditions and actions to the anchors of a node.

A *terminal node* contains data whose internal structure, if any, is application dependent and will not be part of the model. The class of terminal nodes may be specialized into other classes (*text, audio, image*, etc.) as required by the applications.

A *composite node* groups together entities, called *components*, including other composite nodes. The components may be ordered, which is very useful for several navigation mechanisms. The components do not necessarily form a set, because an entity may be included more than once in the composite node. The class of composite nodes may be specialized into other classes, including the class of *context nodes*.

A *context node* groups together sets of links, terminal nodes, trails and context nodes, recursively. Note that the components of a context node *C* form a set, different from a composite node. We say that a component *A* of *C is contained* in *C*. Note that the same component *A* can be contained in a composition *B*, also contained in *C*, since the "be contained" relation is not recursive. Context nodes are subclassed, originating five new classes: *annotation, public hyperbase*, *private base*, *version context*, and *user context*. Only the class of user context is used in the basic model of NCM.

A *user context node C* is a context node which groups together:

1. a set *S* of terminal or user context nodes. Nodes in *S* are said to be contained in *C*.
2. a set *L* of links, such that each link *l* in *L* has the base nodes contained in *C*. Links in L are also said to be contained in *C*.

For each node in *S* there is an attribute where a *node presentation specification* may be defined.

In the specific case of a context node *C* (in particular, a user context node), the value of an anchor may be the special value λ, representing the entire node; a subset of the nodes contained in *C*; or a list of nodes (N_k,..., N_2, N_1) and one anchor, which must belong to the set of anchors of N_1. In this last case, the node N_{i+1} must be a context node, N_i must be contained in N_{i+1}, for all $i \in [1,k)$, and N_k must be contained in *C*.

As an example, let *E* be again a user context node that contains another user context node *S,* containing in turn two nodes, *H* and *M* (storing, for instance, *Hamlet* and *Macbeth*). Since *S* contains *H* and *M*, a link connecting these nodes may, in principle, be defined in *S* as (<*M,i*>,<*H,j*>), where *i* and *j* are valid anchors for *M* and *H*. If one wants to create a link in *E* connecting *M* and *H,* one may define the link as (<(*S,M*),*i*>,<(*S,H*),*j*)>). Note the

difference between defining the link in S or in E. A link defined in S will be seen by every document which includes S (the user context node grouping plays by Shakespeare will probably be shared by several documents), while a link defined in E will be seen in S only by the readers of document E. The same link may yet be defined in E as $(\langle S,(M,i)\rangle,\langle S,(H,j)\rangle)$. In this case the source and end points of the link would be node S, but the anchors (M,i) and (H,j) would specify the "effective end points". The different possibilities for specification of the link in E allow more flexibility in specifying how a node should be presented, making this specification dependent on how the user navigated to the node.

A *hyperbase* is any set of nodes H such that, for any node $N \in$ H, if N is a composite node, then all nodes contained in N also pertain to H.

A *perspective* for a node N is a sequence P = $(N_1,....,N_m)$, with $m \geq 1$, such that $N_1 = N$, N_{i+1} is a composite node and N_i is contained in N_{i+1}, for $i \in [1,m)$. Since N is implicitly given by P, we will refer to P simply as a perspective. Note that there can be several different perspectives for the same node N, if this node is contained in more than one composite node. The *current perspective* of a node is that traversed by the last navigation to that node.

To conclude, NCM also permits extensions to accommodate the notion of *virtual structures* (node contents, anchors and links), that is, structures that result from the evaluation of some expression. A virtual node may have its content and anchors computed when they are selected during the navigation process. In particular, an anchor will be computed when a link that points to it is traversed. Similarly, a virtual link will have its end points computed when selected. The importance of these concepts to our versioning mechanisms is discussed in Section 3.

3 - Versioning

3.1 - Some Versioning Issues in Hypermedia

We adopt as the metric for our versioning mechanism the remarks posed in [Hala88,Hala91], in [Oste92] and in [Haak92]. By analyzing mostly the first reference, we may indeed identify the following requirements (the original sentences are included in italics):

R1. Exploration of Alternate Configurations — "*A good versioning mechanism will also allow users to simultaneously explore several alternate configurations for a single network*".

R2. Configurations Management — "*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. Maintenance of Document History — "*A good versioning mechanism will allow users to maintain and manipulate a history of changes to their network*".

R4. Automatic Update of References — "*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. Support for Versions Sets — "*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*".

By analyzing [Oste92], we may add the following requirements:

R6. Small Cognitive Overhead in Version Creation— "*Explicit version creation will result in a large cognitive overhead. How can version creation be made manageable?*".

R7. Immutability of versions — "*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, its is less clear how links and* (other) *attributes should be treated*".

R8. Versioning of Links — "*One must also consider a separate versioning for links*".

R9. Versions of structure — " *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. Support for exploratory development — "*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*".

By analyzing [Haak92], we may add the following requirements:

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

R12. Support for Alternatives — "*It must be possible to maintain alternatives. Reviewers and authors want to maintain explicit alternatives of sub-parts of a document*".

In addition to these requirements, we believe that the notion of version should also cover two other situations:

R13. Distinct Representation of the Same Information — when distinct objects represent the same piece of information, such as the written and spoken versions of a speech, or two texts prepared by different text formatters, they should be treated as versions of that piece of information.

R14. Concurrent Use of the Same Information — (temporary) copies of the same piece of information, used by distinct running applications, can be usefully treated as versions of that piece of information. This extended use of the notion of version, coupled with a notification mechanism, provides a good basis for cooperative work. Indeed, this generalizes the previous requirement.

3.2 - Extending the Nested Context Model to Include Versioning

3.2.1 - Preliminaries

The basic Nested Context Model already meets Requirement R1 for version control mechanisms. Indeed, the problem of exploring alternative configurations of a document is trivially solved by creating alternative user context nodes over the same set of nodes, reflecting distinct views of the same document tuned to different applications or user classes.

The basic NCM hierarchy, described in section 2, should be extended in order to address the other requirements posed in section 3.1 by adding new entities for versioning and cooperative work. Figure 1 describes the extensions we propose, which are explored in detail in sections 3.2.2, 3.2.3, 3.2.4, 3.2.5 and 3.2.5. Four new subclasses of context nodes, *annotation, public hyperbase*, *private base*, and *version contex* are thus introduced.

In NCM, only terminal nodes and user context nodes are subject to versioning, as seen in white background in the figure 1. Each attribute (including content) of a user context or terminal node may be specified as versionable or non-versionable. The value of a non-versionable attribute may be modified without creating a new version of the object. Modifications on versionable attribute values have to be made on a new version of the object, if it is already committed, as will be detailed in 3.2.3. As stated in R7, “in hypermedia it might be too simplistic to have versions of nodes to be completely immutable... It is not clear how links and attributes should be treated”. It is not even obvious that the contents of a version should ever be immutable. Versionable and non-versionable attributes thus help to meet R7. Of course, some kind of notification mechanism will be needed to enhance version support, specially in the case of concurrent update of non versionable attributes.

The possibility of adding new attributes to a node, without creating new versions, is also attractive. Assume that, after the node was created, a new tool was introduced into the system. The tool might want to store some specific information in the nodes. In NCM, the user may specify if the addition of new attributes is permitted without creating a new version of the object (also helping to meet R7).

Finally, we have not included link versioning in our model, since we believe that this facility adds more complexity than functionality to a system, and that, if necessary, can be modeled through user context node versioning. It remains to study if this facility becomes important when actions and conditions are associated to links.

3.2.2 - Version Contexts

To address the problem of maintaining the history of a document, we extend the Nested Context Model with a special class of context nodes, called *version context*.

A version context V groups together a set of user context or terminal nodes that represent versions of the same object, at some level of abstraction, without necessarily implying that one version was derived from the other. The nodes in V are called *correlated versions*, and they need not belong to the same node class (helping to meet R13). The derivation relationship is explicitly captured by the links in V. We say that v_2 *was derived from* v_1, if there is a link of the form $(\langle v_1,i_1\rangle, \langle v_2,i_2\rangle)$ in V. The anchors in this case simply let one be more precise about which part of v_1 generates which part of v_2. A version context induces a (possibly) unconnected graph structure over all versions. There is no restriction on links (helping to meet R12), except that the "derives from" relation must be acyclic.

It should be noted that version context node can contain user context nodes, since these nodes can be versioned. This provides us with an explicit versioning of the document structure, thus meeting R9.

A user may either manually add nodes (to explicitly indicate that they are versions of the same object) and links (to explicitly indicate how the versions were derived) to a version context, or he may create a new node from another by invoking a versioning operation, which will then automatically update the appropriate version context.

An application has several options to define the node it considers to be its *current version* in a version context V, according to a specific criteria. One of them is to reserve an anchor of V to maintain the reference to the current version. Other anchors may specify other versions following other criteria of choice. Specifically, in the extended model, which includes virtual entities based on a query language, the reference may be made through a query. The query does not need to be part of the anchor of the version context, since it may be defined in a link (see section 2.1) and even in a more general way (helping to meet R5 and R4), as we will see when we discuss private bases in section 3.2.4. It should be noted that the query which defines the current version may return several versions (for example, “all versions created by John”), which can be interpreted as alternatives and presented as a user context (a version context view). Therefore, version contexts meet R4 since they provide an automatic reference update facility.

In the basic model of NCM, we defined an end point of a link contained in a user context node C, and in a similar way, a possible anchor's value of a user context node C, as being a pair consisting of a list of nodes $(N_k,..., N_2, N_1)$ and an anchor α, such that:

- α belongs to the set of anchors of N_1.
- For all $i \in [1,k)$, the node N_{i+1} is a user context node, N_i must be contained in N_{i+1}, and N_k must be contained in C.

In the extended model, N_1 must be either a terminal node or a user context node (as in the basic model) or a version context node, in which case, we say that N_2 does not contain N_1, but contains the node specified by the anchor α.

An application may use version contexts to maintain the history of a document *d* (R3), as well as for automatic reference update (R4), for example, as follows. Suppose that *d* has a component *c* whose versions the application is interested in. Let *C* be the version context containing the nodes $C_1,...,C_n$ that represent the versions of *c*. The application will refer to *C*, and not directly to any of the C_i's, in the user context node *D* it uses to model *d*. All links in *D* touching *C* will point to the same anchor of *C*, which will always point to the node C_i the application considers to be the current version. If the application wants to recover previous versions of *d* with respect to *c*, it simply navigates inside *C*. Indeed, since version contexts are just a special class of context nodes, users may, in principle, navigate through the document history using the basic navigation mechanisms of NCM [Casa91]. Alternatives of the sub-part *c* of the document can be accessed, for example, by a query, which may return a set of alternatives, meeting R12.

A set of selected versions for a configuration is often referred to as a *baseline*. Simply storing the configuration does not indicate how a system has evolved over time, since the selection criteria for a current version in a version context can deliver different versions at different times. It is therefore important to be able to record a static configuration, where each reference is made to a specific version of the node and not a version context. Support for this will be provided by the private base concept, defined in section 3.2.4.

3.2.3 - Consistency

We introduce the notion of *state* of a terminal node and a user context node to control consistency across interrelated nodes, to support cooperative work and to allow automatic creation of versions (see also Section 3.2.4).

A terminal node or a user context node *N* can be in one the following states: *committed*, *uncommitted* or *obsolete*. *N* is in the uncommitted state upon creation and remains in this state as long as it is being modified. When it becomes stable, *N* can be promoted to the committed state either explicitly at the user's request, or implicitly by certain operations the model offers (helping to meet R5). As an example of implicit change of state, an uncommitted user context or terminal node *N* becomes committed when a primitive for version creation is applied

on it. A committed node cannot be directly updated or deleted, but the user can make it obsolete, allowing nodes that reference it or that are derived from it to be notified.

The concept of a node state is in fact only relevant for user context and terminal nodes that have versionable attributes. Therefore when we say, for example, that committed nodes cannot be modified, we mean that the versionable attributes cannot be modified. We also observe, as stated before, that in NCM the user may specify if the addition of new attributes to a committed node is permitted without creating a new version of the object. In what follows, we give more precise definitions for the states of a user context node and of a terminal node.

A user context or terminal node in the committed state, called a *committed node*, has the following characteristics:

- the versionable attributes of the node cannot be modified (which means that explicitly defined attributes cannot be modified, as well as queries, if the node is virtual);
- it can contain only committed or obsolete nodes, if it is a user context node;
- it can be used to derive new nodes;
- it cannot be directly deleted;
- it can be made obsolete, but not uncommitted.

A user context or terminal node in the uncommitted state, called an *uncommitted node*, has the following characteristics:

- all its attributes can be modified;
- it can contain nodes in any state, if it is a user context node;
- it cannot be the source of derivation of new nodes;
- it can be directly deleted;
- it can be made committed, but it cannot be made obsolete.

A user context or terminal node in the obsolete state, called an *obsolete node*, has the following characteristics:

- all its attributes cannot be modified (which means that explicitly defined attributes cannot be modified, as well as queries, if the node is virtual);
- it can contain only committed or obsolete nodes, if it is a user context node;
- it cannot be used to derive new nodes;
- it is automatically deleted by the system, through a garbage collection process, when no longer needed (for example, when it is not referenced by or included in any node);
- it cannot change state.

It follows that, if a node *V* is directly or transitively derived from *W*, then *W* is either committed or obsolete. We also stress that these restrictions guarantee that a committed or obsolete user context node contains only committed or obsolete nodes, which in turn implies that: (i) it also contains only links whose end nodes are committed or obsolete nodes; (ii) the query that defines its content returns a set of committed or obsolete nodes, if it is a virtual context node; and (iii) the queries in its links and anchors always result in a set of committed or obsolete nodes. However, these restrictions do not imply these properties for an uncommitted user context node.

As mentioned in section 3.2.2, derivation links can be explicitly created by a user. The creation of derivation links automatically cause the predecessor object to be committed in order to preserve consistency.

3.2.4 - Public Hyperbase and Private Bases

In general, a cooperative environment must allow users to share information, provides some form of private information for security reasons and permits fragmentation of the hyperbase into smaller units to reduce the navigation space. Cooperative authoring is understood here as the process of creating or modifying the hyperbase, or a subset of the hyperbase, by a group of users.

The notion of context node can be used to support cooperative work. Consider the set of all user context nodes and terminal nodes to be partitioned into several subsets. One and only one of them will form the *public hyperbase*, denoted H_B, that corresponds to public, stable information. The other subsets will form the *private bases*, used to model the user's interaction with a hyperdocument, according to the paradigm (work session) proposed by the Dexter Model. A private base may contain other private bases, permitting organization of a work session into several nested subsessions. Note that one specific (version of a) terminal or user context node can pertain to one and only one of these bases (public or private).

More precisely, we define the *public hyperbase* as a special type of context node that groups together sets of terminal nodes and user context nodes. All nodes in H_B must be committed or obsolete and, as in all hyperbases, if a composite node *C* is in H_B, then all nodes in *C* must also belong to H_B.

We also define a *private base* as a special type of context node that groups together any entity, except the public hyperbase and version context nodes, such that:

i) a private base may pertain to at most one private base;
ii) if a composite node *N* is contained in a private base *PB*, its components are either contained in *PB* or in the public hyperbase or in any private base of a private base nesting contained in *PB*; and
iii) if a link is contained in a private base, its source base end point must be an annotation node.

Intuitively, a private base collects all entities used during a work session by a user.

A user may move a user context node or a terminal node from a private base into the public hyperbase through the use of the *check-out* primitive, as long as the node is committed. If a committed user context node *C* is moved into H_B, then all terminal and user context nodes in *C* must also be moved into H_B.

Note that moving a new version into the public hyperbase need only take place when some modification has been made to the original node. Suppose, for instance, that the user creates a node *V* in a private base as a version of a node *N* of the public hyperbase. Suppose also that *V* is not modified. Then, when he moves *V* to the public hyperbase, *V* is simply destroyed, since there is no need to duplicate information. However, any composite node in the private base that contains *V* must be updated to now contain *N*. Likewise, if *V* is an unmodified version of *N*, all versions created from *V* must be transformed into versions of N in the version context node.

The user context and terminal nodes of a private base *PB* can be moved in block to the public hyperbase through a special primitive, *shift*. In this case we say that the private base was shifted to the public hyperbase. When the *shift* operation is applied, all user context and terminal nodes of the private base are committed and moved to the public hyperbase, and all its private bases are recursively shifted to the public hyperbase. At the end of this process the private base *PB* that was shifted will contain only trails, annotations (and associated links) and private bases, which contain only trails, annotations and private bases, recursively.

A user cannot move a user context or terminal node *N* from the public hyperbase to a private base, but he may create a new node *N'* as a version of *N* in the private base. In HyperProp, work on a document implies in the creation of new versions of all visited user context or terminal nodes in the current private base. These new versions may be derived from committed nodes or correspond to the creation of completely new information (the first node in a version context node). As mentioned in section 2.2, these versions correspond to instantiations in the Dexter Model.

Two primitives, *open* and *check-in*, are available for the creation of a new uncommitted version of a user context or terminal node *N* in a private base *PB*. They differ when *N* is a user context node. In this case, *open* creates an uncommitted version *N'* of *N* in *PB*, as well as of each of the components in *N*, and so on recursively. *N'* will contain the new versions of the components in *N*, and its links will be created so as to appropriately reflect links in *N*. If a committed component pertains to more than one context, only one uncommitted version will be created for this node. On the other hand, *check-in* creates an uncommitted version *N'* of *N,* in *PB*, that contains the original nodes contained in *N*.

Interesting consequences arise from the different behavior between the *open* and *check-in* primitives. Let *N'* contains nodes C_1 and C_2, that in turn contain the same node *M*. If *N'* is created through the *check-in* operation, and node *M* is modified through the two perspectives, C_1 and C_2, two different versions, *M'* and *M''*, will be created. On the other hand, if *N'* is created through the *open* operation, a single new uncommitted version *M'* will be created and will suffer modifications through both perspectives.

The recursive creation of new versions, associated with the *open* operation, does not necessarily occur at the moment the operation is applied. Versions of the nodes (contained in a context node) can be deferred and created only when such nodes are visited. For example, consider a context node *C* containing nodes *I*, *H* and *M*. When an application wants to access *C*, a version *C'* of *C* is created in its private base, which in principle contains the same nodes as *C*. If the application selects *H*, for example, a new version *H'* of *H* is then created in the same private base, as discussed above, which then replaces *H* in *C'*. If *I* and *M* are not accessed, no new versions are created for them.

Committed versions in a private base *PB* can be used to derive versions, or be included in a user context node, in all private bases that contain *PB*, and in all private bases containing these bases, and so on recursively. This is reasonable, since they represent a state of work consistent from the point of view of the job being performed in some private base. Uncommitted versions are only accessible for manipulation in the private base *PB* where they reside.

A user can remove a node *N* from a private base *PB* through a *delete* primitive. If *N* is a trail or an annotation node (this concept will be introduced later), it is simply removed from the private base and destroyed. If *N* is a user context or terminal node, the result depends on the status of *N*. If *N* is uncommitted, it is effectively destroyed and deleted from its version context; if *N* is committed, it will be made obsolete. When a committed node is made obsolete, it is transferred from the private base in which it is contained to the public hyperbase. If it is an obsolete user context node, all its node components are also transferred.

A private base *PB* can also be deleted. In this case, all its nodes, including private bases, are also deleted, recursively. The private base *PB* is then destroyed.

3.2.5 - Version Propagation

In any system with composite nodes, one may ask what happens to a node when a new version of one of its components is created. A system is said to offer *automatic version propagation* when new versions of the composite nodes that contain a node *N* are automatically created each time a new version of *N* is created.

In our system, a node may be contained in many different user context (composite) nodes. Thus, version propagation may cause the creation of a large number of often undesirable nodes. Figure 2 illustrates this problem. The initial hyperbase, schematically shown in Figure 2(a), has a node *D0* that belongs to user context nodes *E0*, *B0* and *F0*; *E0* is in turn in *C0* and *B0* in *A0*. The creation of a version *D1* of *D0* generates five new user context nodes, as shown in Figure 2(b), if exhaustive version propagation is applied.

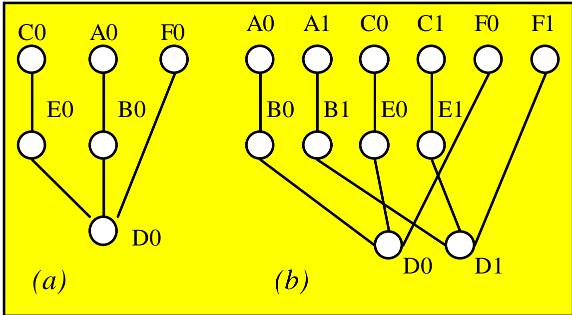

Figure 2 - Proliferation of versions

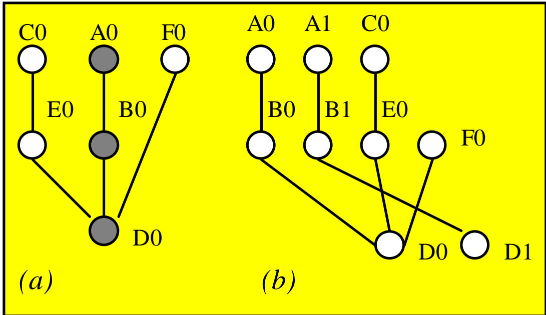

Figure 3 -Propagation guided by perspective

As a solution to this problem, we propose to let the user decide whether he wants automatic version propagation or not, and to limit automatic propagation to those user context nodes that belong to the perspective through which the new version was created. We also limit propagation to those user context nodes that are committed, in line with the restriction that an uncommitted node cannot be used to derive versions. This amounts to providing a mechanism that supports sets of coordinated changes, thus meeting R5, R5 and R10.

Figure 3 illustrates these points. Assume that the initial hyperbase is the same used in Figure 2(a), and that the current perspective is (*D0,B0,A0*). If *B0* and *A0* are committed nodes, then new versions of these two nodes are created, as in 3(b). On the other hand, if *A0* and *B0* were uncommitted, then no new version of these two nodes would be created, but rather *B0* would be altered to include *D1* (*A0* would be left unchanged).

The user may be asked to interfere in situations where the system does not have sufficient elements to decide whether or not to propagate versions.

4 - Related work

This section briefly compares the Nested Context Model with other hypermedia conceptual models, emphasizing the versioning aspects.

The NCM contexts permit partitioning a hypermedia network, generalizing the homonym concept introduced the HAM conceptual model of the Neptune system [DeSc85, DeSc87], that was in turn based on some ideas found in PIE [GoBo87]. The nesting of context nodes offered by the NCM generalize Intermedia's webs [Meyr85]. The KMS tree items also permit the hierarchical structuring of documents [AkCY88], but the inclusion relationship of context nodes in the NCM is much more general. The NCM allows navigation in depth through a set of nested context nodes and, with the help of the notion of perspective, it permit link inheritance, thus generalizing the functionality offered by the notions of browser and filebox found in NoteCard [Mala88]. The notions of context in HyperPro [Oste92] and of composition in HyperBase (CoVer) [SeSt90] are very similar to the concept of context in the NCM, but it is not clear how these systems address the problems of defining the perspective of a node and the notion of anchor for nested nodes.

In the extended version of HAM, a link refers to a specific version or to the newest element (in time) of a set of versions. The user is not free to define the current version, as in the NCM. Slightly more general than HAM, HyperPro organizes the the sets of versions as trees, whereas CoVer and the NCM use acyclic graphs and permit the user to include new derivation links.

Both the NCM and the HyperPro system assign considerable importance to the definition of criteria to select a version from a set of versions, in a global way, thus reducing the user cognitive overhead during the creation of links. In HyperPro, the selection criteria is defined in a context node and is based solely on temporal aspects. In the NCM, the selection criteria, defined in a private base, is more general and may involve a query that does not involve just temporal aspects. Moreover, the NCM permits including in the anchors the definition of queries that define dynamic links to versions within a versions context.

The permanent and temporary states of the NCM are similar to the notions of frozen and updatable used in HyperPro and CoVer. The NCM provides various make a set of nodes permanent. For example, when a context node becomes permanent, all its components also become permanent. Private bases can also be moved in block to the public hyperbase [SoCC93]. These functions surpass the support offered by HAM, HyperPro and CoVer for exploratory work.

The immutability of the attributes of a node in the NCM generalizes the same facility introduced in HyperPro, where it is associated with the node class. It is also more general than that of HAM, where complete nodes (and note their attributes) are defined as versionable or not.

HAM and CoVer support link versioning, although it is not clear how link versions are manipulated in these systems. We did not consider link versioning in the first prototype since we believe that versioning of context nodes suffices. However, link versioning may become interesting when links carry actions, conditions and

synchronization information, besides node references, as in the case of the NCM. This is indeed a topic for future research.

The NCM provides mechanisms to avoid the proliferation of useless versions, based on the concepts of private base and node state, on a version propagation mechanism and on different primitives (open and check-in) to create versions. In the NCM, as in CoVer, the hypermedia system may play a more active role in the generation of new versions than in HAM and HyperPro. Indeed, just as tasks in CoVer, primitive bases in NCM may guide the automatic creation of versions, thus reducing the version creation cognitive overhead. There is nothing similar to the open primitive or to the version propagation concept in any of the models already quoted. The two version creation special primitives and the version propagation mechanism also facilitate the task of coordinating the changes to a set of nodes, thus going beyond the support offered by the other systems.

5 - Conclusions

The Nested Context Model with versioning is the conceptual basis for the hypermedia project under development at the Computer Science Department of the Catholic University of Rio de Janeiro and the Rio Scientific Center of IBM Brazil. A single-user prototype system incorporating the basic Nested Context Model has been concluded. Currently, some applications run on this prototype. A second prototype, conforming with the MHEG proposal and including versioning, is nearly completed. The goal of the project is to create a toolkit for the construction of document processing applications. The toolkit comprises a set of object classes in C^{++} for increased portability and flexibility.

The model is being extended in several directions. First, we plan to cover virtual objects, which involves the difficult task of defining a query language to specify the virtual nodes, links, regions, etc. We are also designing a notification mechanism to enhance version support. Finally, we are studying the inclusion of link versioning, in much the same way as we treat node versioning, which becomes interesting when links carry actions, conditions and synchronization information.

We are also working on other aspects, such as system and data management, protocols, storage and retrieval of multimedia objects, spatial-temporal composition of multimedia objects, etc., although we do not address these issues in this paper. They will be treated in more detail in future works.

REFERENCES

[AkCY88] Aksscyn, 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. June 1988.

[Casa91] 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. December 1991.

[DeSc85] Delisle, N.; Schwartz, M. "Neptune: A Hypertext System for CAD Applications". *Proceedings of ACM SIGMOD '85*. Washington, D.C. May 1985.

[DeSc87] Delisle, N.; Schwartz, M. "Context - A Partitioning Concept for Hypertext". *Proceedings of Computer Supporteed Cooperative Work*. December 1985

[GoBo87] Goldstein, I.; Bobrow, D. "A Layered Approach to Software Design". *Interactive Programming Environments*. McGraw Hill, pp. 387-413. Nova York. 1987.

[Haak92] Haake, A. "Cover: A Contextual Version Server for Hypertext Applications". *Procceedings of European Conference on Hypertex, ECHT'92.* Milano. December 1992.

[Hala88] Halasz, F.G. "Reflexions on Notecards: Seven Issues for the Next Generation of Hypermedia Systems". *Communications of ACM*, Vol.31, No. 7. July 1988.

[Hala91] Halasz, F.G. "Seven Issues Revisited". *Final Keynote Talk at the 3rd ACM Conference on Hypertext*. San Antonio, Texas.December 1991.

[Meyr85] 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. September 1985.

[MHEG93] MHEG. "Information Technology - Coded Representation of Multimedia and Hypermedia Information Objects - Part1: Base Notation. *Committee Draft ISO/IEC CD 13522-1*. July 1993.

[Oste92] Osterbye, K. "Structural and Cognitive Problems in Providing Version Control for Hypertext". *Procceedings of European Conference on Hypertex, ECHT'92.* Milano. December 1992.

[PuGu90] Puttress, J.J.; Guimarães, N.M. "The Toolkit Approach to Hypermedia". *Procceedings of European Conference on Hypertex, ECHT'90*. 1990.

[RiSa92] Rizk, A.; Sauter, L. "MultiCard: An Open Hypermedia System". *Procceedings of European Conference on Hypertex, ECHT'92.* Milano. December 1992.

[ScSt90] Schütt, H.A.; Streitz, N.A. "HyperBase: A Hypermedia Engine Based on a Relational Database Management System". *Procceedings of European Conference on Hypertex, ECHT'90. 1990.*

[SoCC93] 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. April 1993.

[WiLe92] Wiil, U.K.; Leggett, J.J. "Hyperform: Using Extensibility to Develop Dynamic, Open and Distributed Hypertext Systems". *Procceedings of European Conference on Hypertex, ECHT'92.* Milano. December 1992.