Versioning Support in the HyperProp System

Luiz Fernando G. Soares[*] lfgs@inf.puc-rio.br
Guido L. de Souza Filho[+] guido@dimap.ufrn.br
Rogério F. Rodrigues[*] rogerio@telemidia.puc-rio.br
Débora Muchaluat[*] debora@telemidia.puc-rio.br

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

[+] DIMAp, UFRN
Campus Universitário, Lagoa Nova
59072-970 - Natal-RN, Brasil

Versioning Support in the HyperProp System

Luiz Fernando G. Soares[*] lfgs@inf.puc-rio.br
Guido L. de Souza Filho[+] guido@dimap.ufrn.br
Rogério F. Rodrigues[*] rogerio@telemidia.puc-rio.br
Débora Muchaluat[*] debora@telemidia.puc-rio.br

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

[+] DIMAp, UFRN
Campus Universitário, Lagoa Nova
59072-970 - Natal-RN, Brasil

Abstract

This paper discusses version control for hypermedia models with composite nodes that allow distinct representations of the same information segment to be treated as versions. Several problems arise from the possibility of a node being contained in different nested compositions, as well as being presented (edited) with different representations. The paper discusses these issues, relating partial solutions proposed in several hypermedia models and presenting the proposal defined in the Nested Context Model of the HyperProp System.

1 - Introduction

The possibility of defining arbitrary references (links) among parts of a document (nodes) is one of the main characteristics of hypermedia systems. However, we need to combine this flexibility with mechanisms that allow the definition of structured documents. The logical structuring of a document can be achieved by introducing the composition concept.

The composition concept is useful not only for structuring purposes, but also to support cooperative work and efficient version control mechanisms [SoCR95]. However, the complexity of version control increases in hypermedia models with compositions since usual versioning problems grow worst, and link proliferation and node version propagation should be considered.

All these problems become even harder when different representations for a same node (different exhibitions of the same data or different copies of the same information being used at the same time) are allowed. Although the version control mechanism of the HyperProp system, proposed in [SoCR94, SoCR95], highlights the importance of considering different representations of the same information as different versions, it does not consider this case in its versioning operations.

This work analyzes the problem of version support in models with compositions, in particular in models that allow nested compositions, and that allow different node representations to be considered as versions, called representation versions. Link replication in version creation is also discussed. Among other characteristics, the proposed version model supports version groups, allows exploring and managing alternative configurations, maintains documents history, supports cooperative work and provides automatic propagation of version changes.

The paper is organized as follows. A brief description of a hypermedia conceptual model is done in Section 2 in order to introduce the basic concepts used in the paper. Section 3 presents solutions

proposed in the versioning support of the HyperProp system. Comparison with related work is done all over the paper, but specially in Section 4. Section 5 is dedicated to final remarks.

2 - Node Versions in the Nested Context Model

The structured definition of documents is desirable as it carries built-in concepts of modularity, encapsulation and abstraction. Multimedia/hypermedia documents are based on conceptual models with nodes (document's components) and links (relationships among nodes). The introduction of the composition concept brings the notion of structured document, being the central concept of the Nested Context Model - NCM, the conceptual hypermedia model of HyperProp. Figure 1 shows part of the NCM class hierarchy.

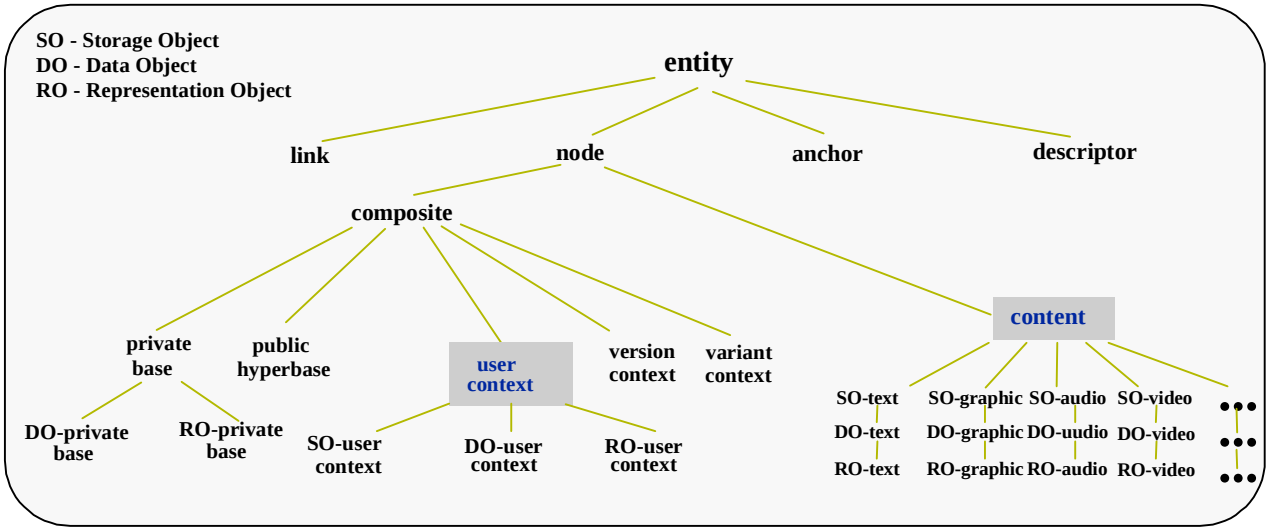

Figure 1 - Partial NCM Class Hierarchy

In Figure 1, an *entity* is an object that has as attributes[1] a *unique identifier* and an *access control list*. The unique identifier has the usual meaning. For each entity attribute, the access control list has an entry that associates a user[2] or user group to his access rights for the attribute.

A *node* is an entity that has as additional attributes a *content*, a list of anchors and a *descriptor*. Each element in the list of anchors is called an *anchor* of the node and defines a set of marked information units in the node's content. There is a default anchor that represents the whole content of a node. The exact notion of information unit and marked information unit is part of the definition of the node. For example, an information unit of a video node could be a frame, while an information unit

[1] We will frequently use the name of an attribute of an entity to refer to the attribute's value. When the context does not allow this simplification, we will explicitly use the term *attribute value*.

[2] The word *user* in the context of this paper has multiple meanings: it means a user in the sense of a person, an application process or an application programmer.

of a text node could be a word or a character. Any subset of information units of a node can be marked.

The descriptor of a node contains information determining how the entity should be presented to the user, as in the presentation specification of the Dexter model [HaSc90].

NCM distinguishes two basic classes of nodes, called content nodes and composite nodes. A *content node* contains data which internal structure is application dependent, modeling traditional hypermedia nodes. Content nodes may be specialized in other classes (text, graphic, audio, video, etc.) as required by applications.

The logical structuring of a document is done using the composition concept (composite node) as a group of document's components (nodes) and, optionally, their relations (links). In some models, composite nodes are defined as a logical assembly of nodes, that can be nested (that is, a composite node can recursively contain another composite node) or not. In these models, references are stored in an independent repository, which sometimes is a single one. In more general models, like NCM, compositions contain not only nodes (that can be nested) but also references (represented by links).

A *composite node C* has, as content, a list *S* of links and nodes, which may be content or composite nodes, recursively. We say that an entity *E* in *S is a component of C* and that *E is contained in C*. We also say that *A is recursively contained in C* if and only if *A* is contained in *C* or *A* is contained in a node recursively contained in *C*. An important restriction is made: a node cannot be recursively contained in itself.

As nested composition models allow a node to be recursively contained in different compositions and composite nodes to be nested to any depth, it is necessary to introduce the concept of perspective in order to better understand the version propagation control proposed. Intuitively, the perspective of a node identifies through which sequence of nested composite nodes a given node instantiation is being observed. Formally, a *perspective* of a node N is a sequence $P = (N_m,...,N_1)$, with $m>0$, such that $N_1=N$, N_m is a node not contained in any composite node, N_{i+1} is a composite node and N_i is contained in N_{i+1}, for $i \in [1,m)$. N_1 is called the *base node* of P. Note that there can be several different perspectives for the same node, if this node is contained in more than one composite node.

Links define relations among nodes recursively contained in a composition. An NCM *link* is an n:m relation composed by a source end point set, a target end point set and a meeting point. Since a node may pertain to more than one composite node, the *source* and *target end points* of a link are identified by pairs of the form $<(N_k,...,N_1),A>$, such that N_1 is a node in the *node list* $(N_k,...,N_1)$, N_{i+1} is a composite node and N_i is contained in N_{i+1}, for all $i \in [1,k)$, with k>0, and A is an event of N_1. N_k is a component of the composite node, that contains the link, or the composite node itself. An event is defined by the presentation of a set of marked information units (an anchor) of a node (presentation event) or by its selection (selection event). The *meeting point* defines *conditions* and *actions*. Conditions must be satisfied at the source end points (for example, the selection of an anchor by a user) in order that *actions* be applied at the target end points (for example, the presentation of an anchor).

NCM links can be defined in any composite node of a perspective of a node. In order to define these contextual links, it is necessary to identify what links effectively anchor on a node in a certain perspective. These links are called *visible links* and will also play an important role in versioning. More precisely, given a node N_1 in a perspective $P = (N_m,...,N_1)$, with $m>0$, we say that a link l is visible in P if and only if there is a composite node N_i in P containing l, for $i \in [1,m]$, such that:
i) if $i>1$ then $(N_{i-1},...,N_1)$ is the node list of one of the end points of l;
ii) else, (N_1) is the node list of one of the end points of l.

A more detailed description of the NCM classes can be found in [SoCR95]. The given partial description of the model is sufficient to allow the definition of its several types of versions.

An authoring environment should offer good editing and browsing tools for defining the node's content and the content granularity, which specifies the set of information units that can be marked, defining anchors. The definition of a node content and its anchors are contained in objects called *data objects* — DO. A data object is created either as a totally new object or as a local version of other data objects or as a local version of already created persistent objects, called *storage objects* — SO. Data objects created from other objects are considered *data versions*.

An authoring environment, however, should also permit the definition of each component expected behavior when presented. For each presentation event, one should be able to specify how and with which tool the associated data object will be presented. These definitions must preferably be specified independent of data objects. In NCM, descriptor objects contain these specifications. Note that the independence between descriptors and data objects will permit different presentations of the same data. For example, a text node can be presented as text, using descriptor D_{e1}, or it can be synthesized as audio using descriptor D_{e2}.

The aggregation of a data object and a descriptor in order to present a component is called a *representation object* — RO and will be considered a *representation version* of the data object. Thus, storage, data and representation objects are related through versioning operations, as will be defined in Section 3.

Every time a data object is presented, it must be associated to a descriptor explicitly defined on-the-fly by the end user or by descriptors defined during the authoring phase. During authoring, descriptors can be defined as an attribute of a link used to reach the data object, or as an attribute of the composite node that contains the data object to be presented, or as an attribute of the data object itself, or even as a default attribute of the data object class. In NCM, a descriptor explicitly defined on-the-fly by the end user bypasses the descriptors defined during the authoring phase. These in turn have the following precedence order: first, that defined in the link used to reach the node; second, that defined in the composite node that contains the node, if it is the case; third, that defined within the node; and finally the default descriptor defined in the node's class.

Figure 2 shows the association of a data object node and a descriptor. This association is represented by dashed lines connecting data objects in the middle plane to representation objects drawn in the upper plane. Note that the HyperProp conceptual model permits combining one node to different descriptors, originating different representation objects of the same node. Figure 2 shows this feature with the association of descriptors D_{e1}, D_{e2} and D_{e3} to node *A'*, creating representation objects A_1",

A_2", and A_3". In the figure, *A'* has three different representations, for example, one for each different way to access it. One can reach *A'* by depth navigation (through the composite node *C'*) using the descriptor D_{e1}, or it can be accessed by one of the links that touches it using the descriptor D_{e2}, or even by the other link using the descriptor D_{e3}.

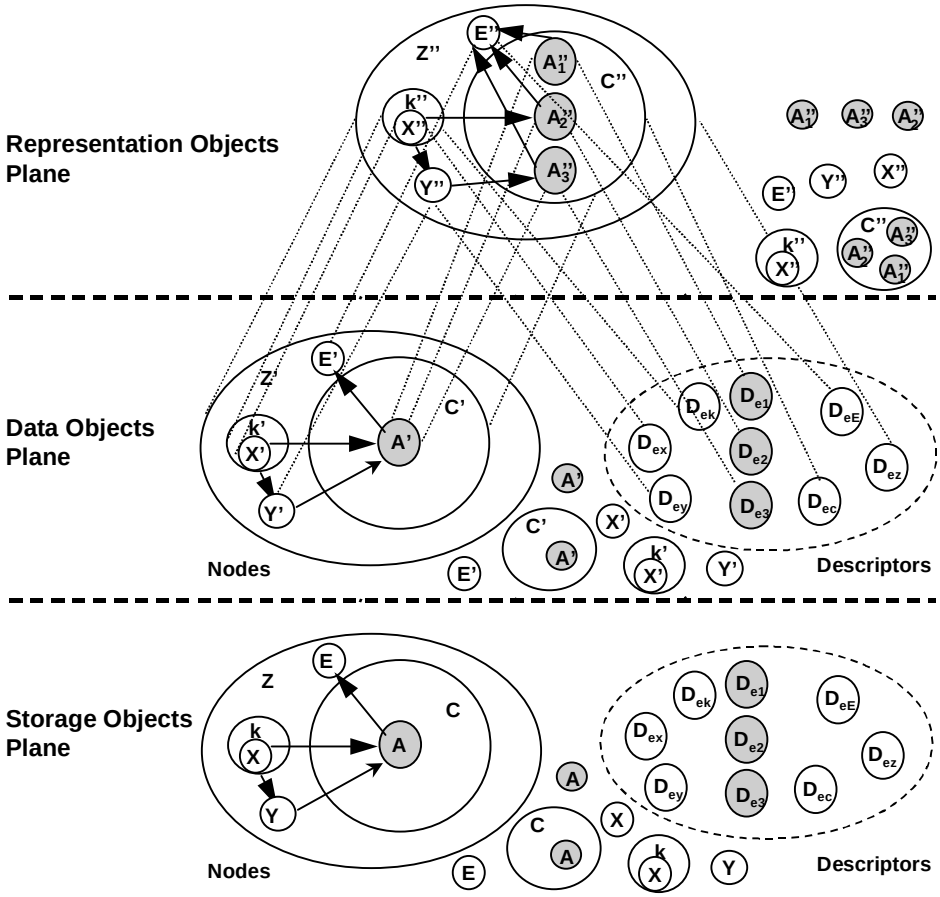

Figure 2 - Object Planes

Note that in representation version creation, links are replicated (see the outgoing link from node *A'*). When we allow several representation versions for the same data node, a rigorous definition of this replication is needed.

3 - Version Control in NCM

Although the literature stresses the importance of considering representation objects derived from the same data object as different versions, there is not any proposal that even discusses that possibility. Indeed, works that mention that issue allow by simplicity only one representation version of a data object. NCM extends the previous works allowing that distinct copies (representations) of the same piece of information (in the same or different media) be 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.

In order to address versioning and cooperative work, the composite node class is specialized in other classes. A *user context node U* is a composite node such that the content attribute of *U* contains only

content nodes, user context nodes and links. User context nodes will allow the structuring of a document, the definition of different views of the same document, and also will alleviate the disorientation problem when we navigate through a document [MSCS97].

In NCM, only content nodes and user context nodes are subject matters to versioning, as seen in gray background in Figure 1. Each attribute (including the content) of a user context or content node may be specified as *versionable* or *non-versionable*. The value of a non-versionable attribute may be modified without creating a new version. Modifications in versionable attribute values have to be made on a new version of the object, if it is already committed, as it will be detailed in the next paragraphs. 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. Moreover, in NCM, the user may specify if the addition of new attributes to a node is allowed without creating a new version.

Version models usually introduce the notion of *state* of a node to control consistency across interrelated nodes, to support cooperative work and to allow automatic creation of versions. The state is just a new node attribute whose value affects and is affected by the execution of certain operations.

In NCM, a content node or a user context node *N* can be in one of 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. A committed node cannot be directly updated or deleted, but the user can invoke another specific operation to make it obsolete, allowing nodes that reference it or that are derived from it to be notified. In other words, obsolete nodes are kept as long as they are referenced by a link or contained in another node, which means that the user can re-visit obsolete nodes as long as they are reachable.

In order to support representation versions, the state definitions of the early NCM [SoCR95] were extended. More precisely, a user context or content 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 that implicitly define the attribute, if the node is virtual[3]);
- it can only contain committed or obsolete nodes, if it is a user context node;
- it can be used to derive new nodes (data or representation versions), if it is not a representation node;
- it cannot be directly deleted;
- it can be made obsolete, but not uncommitted.

[3] A *virtual entity E* is an entity such that the value of at least one attribute *A* is an expression, written in a formally defined hypermedia query language, whose evaluation results in an object of the appropriate type. The attribute *A* is also called *virtual*.

A user context or content 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 used to derive new data nodes (that is, data versions);
- it can be used to derive new representation nodes (that is, representation versions), if it is a data node;
- it can be directly deleted;
- it can be made committed, but it cannot be made obsolete.

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

- none of its attributes can be modified (which means that explicitly defined attributes cannot be modified, as well as queries, if the node is virtual);
- it can only contain committed or obsolete nodes, if it is a user context node;
- it cannot be used to derive new nodes;
- it cannot change state.

A storage object must be in the committed or obsolete state, conforming to MHEG [MHEG95] proposal, where objects in the storage layer are always in its final form, that is, cannot have their versionable attributes modified. In HyperProp an obsolete node is automatically deleted by the system, through a garbage collection process, when it is no longer referenced by a link or contained in another node.

In order to address the problem of maintaining the history of a document, we introduce the classes of version context nodes and variant context nodes. A *version context V* is a composite node that contains only links, user context and content nodes, which are storage objects or data objects. Intuitively, *V* groups nodes that represent data versions of the same entity, at some level of abstraction, without necessarily implying that one version is derived from another. The nodes in *V* are called *correlated versions*, and they do not need to belong to the same node class. 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 from v_1 to v_2 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, except that the "derives from" relation must be acyclic. Note that representation versions are not included in version contexts.

A user may either manually add nodes (to explicitly indicate that they are versions of the same object) and links (to explicitly indicate how versions are derived) to a version context, or create a new node from another by invoking a versioning operation, which will then automatically update the appropriate version context. The creation of derivation links automatically causes the predecessor object to be committed in order to preserve consistency.

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. Specifically, in the model which includes virtual entities based on a query language, the reference may be made through a query. Reference [SoCR95] discusses several ways to define the current version and also the modifications needed in the link definition.

A *variant context V* is a composite node that contains only user context and content nodes, which are representation objects. Intuitively, *V* groups different representation versions of a same data object.

We could have used version context nodes in order to store all versions (representation and data versions) of a node. There are two main reasons for not doing that. The first one is that representation versions can be derived from uncommitted nodes, what means that data consistency is not necessarily maintained on variant context nodes, in contrast to version context nodes. Actually, this consistency will be maintained if changes in data objects are made by check-in primitives between representation and data objects planes, as will be defined in the next section, but the NCM use does not require it. The second reason is that in a client-server architecture implementation, it is natural that the server should maintain version context nodes. However, representation version creation operations of the same data node are usual constrained to only one client. Thus, it is appropriate that clients manage the derivation structure of variants (representation versions), avoiding unnecessary message exchanges. Therefore, a version graph (data and representation versions) may be thought as the concatenation of its distributed components (variant and version contexts), where consistency is assured in the whole graph, except in the sink nodes defined by representation versions.

Public Hyperbase and Private Bases

The notion of composite node can be used to support cooperative work and to abstract the plans of Figure 2. A similar concept can be found in [Haak92, Oste92]. The *public hyperbase*, denoted H_B, is defined as a special type of composite node that groups together sets of content nodes and user context nodes. All nodes in H_B must be storage objects and if a composite node *C* is in H_B, then all nodes in *C* must also belong to H_B. Intuitively, the public hyperbase groups all entities stored in a server, similar to a WWW server, and represents the lower plane of Figure 2.

We also define a private base as a special type of composite node that groups together content nodes, user context nodes and private bases, such that a private base may be contained in at most one private base. The private base class is specialized into DO-private base and RO-private base. It is exactly on the specialization of the private base class that resides the main contribution of this work regarding representation versions. For simplicity, we will assume from here on that private bases do not contain other private bases, since this concept, although important in cooperative environment, brings even more complexity to versioning, hiding concepts we want to stress in this work.

A *DO-private base DOPB* is a data object, such that:

i) it only contains data objects;
ii) if a composite node *N* is contained in *DOPB*, its components are either contained in *DOPB* or in the public hyperbase.

An *RO-private base ROPB* is a representation object, such that:

i) it is derived from a DO-private base;
ii) it only contains representation objects;
iii) if a composite node *N* is contained in *ROPB*, its components are either contained in *ROPB*, or in the DO-private base from where *ROPB* was derived, or in the public hyperbase.

A new *ROPB* derived from a *DOPB* has in addition new attributes and methods for handling the content attribute of the private base. Data objects of *DOPB* will be associated to their descriptors, in order to create the representation objects that will be contained in *ROPB*. Representation versions contained in *ROPB* will be created according to versioning operations as will be specified ahead in this section. Intuitively, *ROPB* collects all entities used by a user during a work session, represented by the upper plane of Figure 2.

Let *P"* be a perspective of a node *N* in an RO-private base *ROPB*, where *N* is a storage or data object. Let *P'* be a perspective corresponding to *P"* in the DO-private base from which *ROPB* was derived, such that *P'* contains *N* as its base node and has all the representation nodes of *P"* replaced by the data nodes from which they were derived. We call *P"* the *correlated perspective* of *P'*, and vice-versa.

A user cannot move a node *S* from the public hyperbase to a DO-private base, but he may create a new node *D* as a data version of *S* in the DO-private base. New data versions may be derived from committed nodes (data or storage nodes) or correspond to the creation of completely new information. Likewise, a node in a DO-private base cannot be moved to an RO-private base, but new representation versions of the node may be derived and included in the RO-private base. New representation versions may be derived from committed or uncommitted data nodes. These versions correspond to instantiations in the Dexter Model.

Two primitives *Check-out (P, D_e)* and *Check-out-one (P, D_e)* allow the creation of a new representation version of a data object *D*, where *D* is the base node of the perspective *P* of an RO-private base *ROPB*[4]. The version created by the aggregation of the descriptor D_e to the node *D* must be included in *ROPB*. The primitives differ when there is already a representation version of *D* in *ROPB*, derived using the same descriptor D_e. In this case, *Check-out (P, D_e)* creates a new version in *ROPB* and *Check-out-one (P, D_e)* does nothing.

Let us assume that *R* is the representation version of the data object *D* in *ROPB* by the aggregation of the descriptor D_e through the *Check-out (P, D_e)* or *Check-out-one (P, D_e)* operation, or that *R* had already been created in *ROPB* when *Check-out-one (P, D_e)* was executed. Let us also assume that *D*, in the perspective *P*, is contained in a composition *C* different from *ROPB*. If there is no other reference (by link or by composition) to node *D*, defined in any node in perspective *P*, with

[4] Note that version control involves the "movement" of data between bases. In our previous work [SoCR95], we created versions through *check-in* primitives, since we analyzed the problem having the target base as our point of view, where new versions should check-in to enter. In this work we have changed this primitive's name to *check-out*, adopting the source base as our point view, in order to be in agreement with other related works in the literature.

other descriptors, then *R* is included in *C* and *D* is removed from *C*. Otherwise, *R* is included in *C* but *D* is not removed from *C*. In both cases, all visible links of *D* in *P*, if they exist, must be handled to reflect the new situation.

Two primitives, *open* and *check-out*, are available for creating a new uncommitted data version of a node *N* (storage or data object) in a DO-private base *DOPB*.

The *Check-out (P)* primitive, where *N* is the base node of perspective *P*, creates a new uncommitted data version of *N* in *DOPB*. The created version *D* is identical to *N*, in the sense that, if *N* is a user context node, *D* will contain the same nodes and links of *N*. *D* will be included in *DOPB* and if *N* in the perspective *P* is contained in a composition *C* different from *DOPB*, then *D* is included in *C*. If *N* is a storage object, it must be removed from *C*. In this case, all visible links of *N* in *P*, if exists, must be updated in order to have the new node *D*, replacing node *N*, in their end points.

On the other hand, the *Open(P)* primitive creates an uncommitted version *D* of *N* in *DOPB*, as well as of each component recursively contained in *N*, if *N* is a user context node. *D* will recursively contain the new data versions of the components recursively contained in *N*, and its links will be created so as to appropriately reflect links defined in *N* and in its recursively contained user context nodes. If a committed component pertains to more than one context, only one uncommitted data version will be created for this node. *D* will be included in *DOPB* and if *N* in the perspective *P* is contained in a composition *C* different from *DOPB*, then *D* is also included in *C*. If *N* is a storage object, it must be removed from *C*. In this case, all visible links of *N* in *P*, if they exist, must be updated in order to appropriately reflect in their end points the new node *D* and all new data versions created, recursively contained in *D*, if *D* is a user context node.

Both in *Open(P)* and *Check-out(P)*, when the base node *N* of *P* is a storage object, *N* must also be replaced in the RO-private base derived from the DO-private base where *P* is defined. *N* must be replaced by the new version *D* in all correlated perspectives *P'* of *P*. All visible links of *N* in *P'*, if they exist, must be updated in order to appropriately reflect in their end points the new node *D* instead of *N*. In the case of *Open (P)*, the visible links must also reflect in their end points all new data versions created, recursively contained in *D*, if *D* is a user context node, replacing all nodes from which the new versions were derived.

Using the four previous primitives we can define compound version operations in accordance to requirements of a particular application in order, for example, to create a new representation version of a storage node *N* in an RO-private base, by aggregating a descriptor.

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

Let a data object *D*, the base node of one or more perspectives P_i of a DO-private base, be a version of a storage object *S*. Let C_i be the data object that contains *D* in P_i. Let also C'_i be a context, called related context to C_i, which contains one or more representation versions of *D*, such that: if C_i is a DO-private base, then C'_i is an RO-private base; else C'_i is a representation version of C_i. When

more than one representation object R_1,.., R_j,.. R_n was derived from *D*, if any representation object R_j changes its state from uncommitted to committed, we have the following alternatives.
i) Any versionable attribute of R_j was modified and, or R_j is not the last representation node that changes to the committed state, or *D* is in the committed state and R_j is the last representation node that changes to the committed state. In both cases a new data object D_j is created, corresponding to R_j. D_j will be committed and included in the appropriate version context as derived from *S*. D_j will be also included in all compositions C_is, related to C'_is in which R_j is contained. In these compositions C_is, all visible links of *D* in P_i must be handled to reflect the new situation. A notification will be sent to all other representation nodes.
ii) Any versionable attribute of R_j was modified, *D* is in the uncommitted state and R_j is the last representation node that changes to the committed state. In this case *D* will be modified to have the same contents of R_j and then it will pass to the committed state. If there is any reference to *D* having a different descriptor from that used to create R_j, a notification should be sent remarking that the referenced node has been modified. An alternative approach, in this last case, would be to allow, if desired by a user, the creation of a new data object D_j, corresponding to R_j, taking into account all circumstances discussed in the previous item 1.
iii) R_j remains unchanged and *D* is in the uncommitted state. In this case, *D* will be made committed.
iv) R_j remains unchanged and *D* is in the committed state. In this case nothing needs to be done.

The *Check-in(R)* of a representation object *R* makes *R* committed and then (obsolete and) destroys it. The corresponding data object *D* is included in compositions where *R* was, if *D* was not there yet. All links recursively defined in the RO-private base must be appropriately changed in order to contain *D* in replacement to *R*.

The *Check-in (D)* of a data object *D* implies in a check-in of all its derived representation objects. In a data object *check-in*, the data object moves to the public hyperbase becoming a storage object. Note that moving a new version into the public hyperbase needs only to take place when some modification has been made to the original node. For instance, suppose that a user creates a node *D* in a DO-private base as a version of a node *S* of the public hyperbase. Suppose also that *D* is not modified. Then, when *D* is moved to the public hyperbase, *D* is simply destroyed, since there is no need to duplicate information. However, any composite node in the private base that contains *D* must be updated to then contain *S*. Likewise, if *D* is an unmodified version of *S*, all versions created from *D* must be transformed into versions of *S* in the corresponding version context node.

Compound operations can also be derived from the check-in primitives, similarly to compound operations for version creation. For example, we can define the check-in of a node in an RO-private base to the public hyperbase.

A user can remove a node *N* from a private base *PB* through a *delete* primitive. If *N* is uncommitted, it is effectively destroyed and deleted from its version context (in the case of a data node *N*); if *N* is committed, it will be made obsolete. When a committed node is made obsolete in a DO-private base, it is transferred to the public hyperbase. If it is an obsolete user context node, all its node components are also transferred. When a node is removed from a DO-private base, all representation objects derived from it must be removed from the corresponding RO-private bases. A private base

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

4 - Related Work

Versioning has been investigated specially in software engineering and design databases. Several models have been proposed in the literature to describe the organization and manipulation of documents in environments for cooperative work, specially in areas such as CAD and Office Automation. Several other systems mentioned throughout the text address version support in hypermedia systems [DeSc85, Haak92, Oste92, HaHa93, HaHi96, DHHV94, WhAT94, WebD98]. Reference [SoCR95] brings a detailed comparison among several models. In this section we will only focus on representation versions support, which is not discussed in that reference.

None of the mentioned hypermedia systems considers different representations of the same object as versions. Usually, the problem is simplified by allowing only one representation of each object. The WebDAV group [WebD98] defines *resource variants* as different content of the same information, but does not define how these variants are created. Indeed variants are not treated as versions. Moreover, they have only defined the Web versioning protocol to trigger some versioning operation, but did not specify what would happen, for example, to links referring a node when versions of this node are created. Different representation versions of a same node may not only generate a useless proliferation of data versions and links, but may also add an additional difficulty to the maintenance of the history of correlated versions, if certain rules are not well defined.

HAM [DeSc85] and CoVer [Haak92, HaHa93, Haak94] support link versions, but they do not explain how they are handled. We do not consider link versions in our first prototype, since we believe that composite node versions are enough. Note that in representation versions creation, links are replicated as illustrated in Figure 2 by the outgoing links of k (k = 3) nodes *A"*. The problem would be worse if we had j representation versions of node *C'*, when we would have $j*k$ replicated links. The situation would be even worse if the outgoing link from node *A'* was n:m, that is, if it had n source end points. In this case, we would have $n*j*k$ possible replicated links. Actually, these replicated links are versions in a system with link versioning. When a data object may generate several representation objects, a severe definition of this replication is very important, since, among other things, it will impose link version proliferation, which seems to be a more complex matter than node version proliferation control.

The restrictions and concessions on node states for version derivation are very important, both for representation and data versions, and this issue is not mentioned in any other system. Node state definitions may seem excessively restrictive at first glance. For example, it may seem rather restrictive not to allow an uncommitted node to be the source of derivation of new data nodes. This possibility would make it very hard for the system to guarantee consistency of version history. Nevertheless, it is worth noting that an application may easily offer an interface where asking for the creation of a new version of an uncommitted data node *D* automatically implies in the creation of a new version of the committed node from which *D* was derived. On the other hand, allowing new representation versions to derive from uncommitted data objects is very important to avoid data version propagation. For instance in Figure 2, if it was necessary that *C'* be in the committed state, a

modification in representation object A_2" followed by a *Check-in(A_2")* operation would imply in the creation of new data versions of *C'* and of all data nodes that recursively contain *C'*, inducing a propagation of possible useless versions.

NCM provides several mechanisms to avoid unnecessary version proliferation based on the concepts of private bases, node states and on different version creation primitives (*open*, *check-out* and *check-out-one*). There is nothing similar to *open* and *check-out-one* primitives in any of the mentioned models. These two special primitives for version creation also help supporting coordinate changes in sets of nodes, providing a better support than that offered by the other systems.

As commented before, reference [SoCR95] may be considered for a detailed comparison of mentioned related work, regarding several other issues.

5 - Final Remarks

The first prototypes of the HyperProp system have been developed in the TeleMídia Laboratory of the Computer Science Department of PUC-Rio, as part of a project to build an open hypermedia system. The goal of the project is to create tools to incorporate hypermedia document handling in any application through the use of a set of classes initially implemented in C++. The prototypes incorporate all facilities for authoring and rendering documents with temporal and spatial constraints, for version control and for cooperative work. Due to the lack of an adequate query language, the current implementation does not offer virtual structures. In fact, without having a powerful querying mechanism it is impossible to offer a good support for virtual structures, which makes the version control system incomplete. This is one of the following issues to be addressed in the next extension of the model. Support for cooperative work has always been one of the project's requirements, that started with document handling in teleconferencing systems. Notification control mechanisms, important for cooperative work, should also be part of the next model extension, currently being implemented in Java and providing the integration of NCM documents to HTML documents. In this new prototype, the presentation of NCM documents in Web browsers is already available [RoMS98].

This work was partially granted by the Brazilian Telecommunications Enterprise (Embratel), CNPq and FINEP.

REFERENCES

[DHHV94] Durand, D.; Haak, A.; Hicks, D.; Vitali, F. (Eds). *Proceedings of the Workshop on Versioning in Hypertext Systems*, held in connection with ECHT'94. Disponível como *Arbeitspapiere der GMD 894*, GMD-IPSI, Darmstadt, Alemanha.

[DeSc85] Delisle, N.; Schwartz, M. "Context - A Partitioning Concept for Hypertext". *Proceedings of Computer Supporteed Cooperative Work*. December 1985

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

[HaHa93] Haake, A.; Haake, J. "The CoVer: Exploiting Version Support in Cooperative Systems". *Proceedings of INTERCHI'93 - Human Factors in Computing Systems*. Amsterdam. Abril de 1993.

[Haak94] Haake, A. "Under CoVer: The Implementation of a Contextual Version Server for Hypertext Applications". *Proceedings of the European Conference on Hypertext*. Edimburgo. Setembro de 1994.

[HaHi96] Haake, A.; Hicks, D. "VerSe: Towards Hypertext Versioning Styles". *Proceedings of Hypertext 96*. Washington, EUA. Setembro de 1996.

[HaSc90] Halasz, F.G.; Schwartz, M. "The Dexter Hypertext Reference Model". *NIST Hypertext Standardization Workshop*. Gaithersburg. Janeiro 1990.

[MHEG95] MHEG. "Information Technology - Coded Representation of Multimedia and Hypermedia Information Objects - Part1: Base Notation. *Committee Draft ISO/IEC CD 13522-1*. Julho 1995.

[MSCS97] Muchaluat, D.C; Soares, L.F.G., Costa, F.R.; Souza, G.L. "Graphical Structured-Editing of Multimedia Documents with Temporal and Spatial Constraints". *Proceedings of the Fourth International Conference on Multimedia Modelling*, Cingapura. Novembro de 1997; pp. 279-295.

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

[RoMS98] Rodrigues, R.F.; Muchaluat, D.C.; Soares, L.F.G. "Composite Nodes and Links on the World-Wide Web", Technical report of TeleMídia Laboratory, Computer Science Department, PUC-Rio, Rio de Janeiro, Brazil, February 1998.

[SoCR94] Soares, L.F.G.; Casanova, M.A.; Rodriguez, N.R. "Nested Composite Nodes and Version Control in Hypermedia Systems". *Proceedings of the Workshop on Versioning in Hypertext Systems*, in connection with ACM European Conference on Hypermedia Technology, Edinburgh. Setembro de 1994.

[SoCR95] Soares, L.F.G.; Casanova, M.A.; Rodriguez, N.R. "Nested Composite Nodes and Version Control in an Open Hypermedia System". *International Journal on Information Systems; Special issue on Multimedia Information Systems*, vol 20, no. 6. Elsevier Science Ltd. England. Setembro de 1995; pp.501-520.

[WhAT94] Whitehead, E.J.; Anderson, K.M.; Taylor, R.N. "A Proposal for Versioning Support for the Chimera System". *Proceedings of the Workshop on Versioning in Hypertext Systems*, held in connection with ECHT'94. Available as *Arbeitspapiere der GMD 894*, GMD-IPSI, Darmstadt, Alemanha.

[WebD98] WebDAV - WWW Distributed Authoring and Versioning publications available in http://www.ietf.org/html.charters/webdav-charter.html.