An Open Hypermedia System with Nested Composite Nodes and Version Control

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

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.

The paper also presents a generic layered architecture for hypermedia systems with four major interfaces. The explanation of the layers and interfaces will be followed by a discussion about object organization and a discussion on how it relates to the concepts of the Nested Context Model introduced in the paper.

Keywords: hypermedia, versioning, cooperative work

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 [DeSc86], 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.

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 presents the HyperProp architecture and shows how it accommodates the concepts introduced in previous sections. Finally, section 5 contains the conclusions.

2 - The Nested Context Model

The goal of the construction of 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. For the sake of completeness, a comparison with related work is also included.

2.1 - The Basic Model

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.

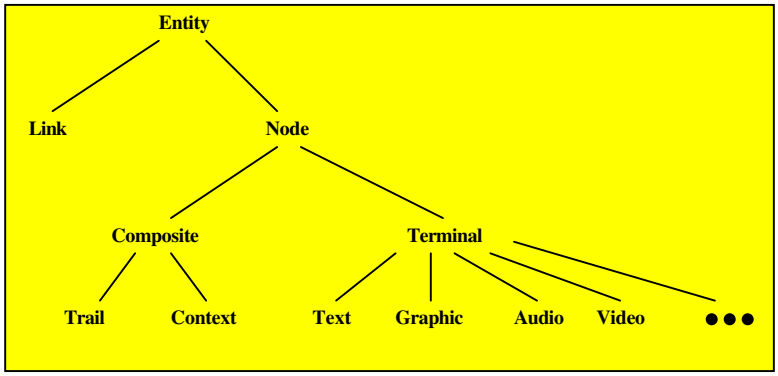

Figure 1 - NCM Basic Class Hierarchy

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. This will be very useful for several navigation mechanisms, as will be seen later. 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 and context nodes. It permits organizing, hierarchically or not, sets of nodes. It thus offers a mechanism to structure documents that helps lessen the so-called "lost in hyperspace" problem [Hala88]. For example, to organize a textbook one may first define a context node *B* that contains a set of nodes standing for the chapters of the book and a set of links indicating the chapter interdependencies, which do not necessarily induce a linear sequence for the chapters. Similarly, for the i^{th} chapter, one may define a context node C_i that contains a set of nodes standing for the sections of the chapter and a set of links indicating the section organization, and so on.

A *link* basically connects nodes. Since the content of a node may have an arbitrarily complex internal structure, links either anchor on whole nodes or indirectly indicate regions where they touch the nodes. A region may correspond to an icon item; for text nodes a region may correspond to a character string within the text; for 2D images it may be determined by a pair of coordinates that define a rectangle; and for context nodes, a region may be inside one of its component nodes, as defined in the next paragraphs.

The informal notion of "region" is described in the model by an *anchor*. Each node has a *mask* that acts as the external interface of the node. That is, links will actually indirectly refer to regions inside the content of the node by addressing entries in the mask, called anchors. 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. As an example, consider a link with a target end point defined on the second paragraph of a text. If this end point were to be defined simply as a displacement measured in bytes, the elimination of the first paragraph in the text would result in an erroneous definition. With the use of anchors, the target point can be made to identify the desired information, no matter what happens to the surrounding text.

Different classes of anchors may thus be defined for different classes of nodes. The system has no understanding of the internal representation of an anchor, except in the case of context nodes, as will be seen below.

To summarize, a node is an object with several attributes, including a content and a mask, which is a set of anchors. The exact definitions of content and anchors depend on the class of the node. In what follows, we give more precise definitions for the key concepts of entity, link and context node.

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[1] or user group to his access rights for each attribute. Questions such as: "May all users create links?" "May any user delete a link?" "May any user edit a node?" "May a user delete a node referenced by other users?" among others, may be answered by the access rights mechanism. 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.

In conformance to the Dexter Model, every anchor has an associated *id* and *value*. As discussed above, it is not possible to describe anchors for generic nodes. Nevertheless, every class of

---

[1] The word *user* in the context of this paper has multiple meanings: it means an user in the sense of a person, an application process and an application programmer. That is, anything or anybody that makes use of the services defined at the several interface layers of HyperProp.

anchor has a special value, denoted by λ, representing the whole node on which the anchor is 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 mask 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. As an example, consider a context *E*, defined in a work document of a Drama Research Team, representing a study of English Poetry in the XVI Century. Inside this context, there may be one context *S* grouping plays by Shakespeare and another context *M* grouping sonnets by Cristopher Marlowe. It may well be that the group wants to document a connection between a specific play by Shakespeare, say Hamlet, stored in a node *H* in *S* ,and a specific play by Marlowe, say Dr. Faustus, stored in a node *F* in *M* (such a link could be used, for example, to document a connection between plays where the main theme is conflict). This link may be defined in context *E*, and its end points will be defined by lists (S,H) and (M,F). The anchors will allow the connection to be made more precisely, for instance, marking the sentences where the common concept first appears.

Although it is not part of the structural conceptual model, but of the presentation model, 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.

A link has also associated to it conditions and actions, in conformance with the MHEG proposal [MHEG93]. The actions defined within the link are processed on the indicated target nodes only if the conditions are satisfied. A *trigger condition* is associated to a variation of the evaluation of values of attributes of the source end point of the link. Additional conditions may be defined on the current evaluation of values for attributes of the same or another entity. Actions define spatial-temporal relations between the source end point and the target end points of a link. One example of an action could be: at the end of the presentation (condition) of entity *Y* (source end point) present entity *X* (target end point) in a window specified by the coordinates (x,y); or even at the beginning of presentation of *Y* prepare for presentation (pre-fetch) of entity *X*. We are assuming in these cases that the entities X and Y have an attribute which specifies their exhibition state (sleeping, prepared, running, etc.).

Similar to links, as an extension of the MHEG proposal, the Nested Context Model associates conditions and actions to the anchors of a node. As an example, a composite node X may have an anchor with value (N,i) (as will be seen later), corresponding to a specific region (described by i) in a node N, contained in X, and an associated action specifying that this region must be

presented 10s after node X begins to be presented. Link and anchor conditions and actions are discussed in further detail in [SSCC94].

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

1. a set *S* of terminal or 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*, 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 mask 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 context node that contains another 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 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 (<*S*,(*M,i*)>,<*S*,(*H,j*)>). 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.

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.

2.2 - Issues on the Presentation Model

The structural conceptual model of NCM treats hypermedia as an essentially passive data structure. A hypermedia system must, however, provide tools for the user to access, view, manipulate and navigate through the network structure. This functionality is captured by the

presentation model, which is closely related to versioning in NCM. This model is briefly described in this section.

The Nested Context Model allows different composite nodes to contain the same node and composite nodes to be nested to any depth. Thus, we need a way of identifying through which sequence of nested composite nodes a given node is being observed. This is captured by the notion of *perspective* of a node.

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.

The user's interaction with a hyperdocument is modeled in conformance to the Dexter Model (DM). The fundamental concept in the runtime layer of DM is the instantiation: a presentation of the component to the user. The instantiation of a node also results in instantiation of its anchors. In DM a function called instantiator is responsible for returning an instantiation given a component and its presentation specification. Instantiations exist within a DM session. Work sessions are modeled in NCM by *private bases*, which will be discussed in section 3. An instantiation in NCM is just a version created in a private base. For now, it suffices to say that the creation of instantiations in NCM, like in the Dexter Model, is based on a presentation specification.

The notion of presentation specifications has already been mentioned in section 2.1. Presentation specifications contains information for the presentation of an entity. In particular, they define methods for exhibition or edition of entities. These methods can be any program, and in particular, any editor. The Nested Context Model provides specific methods for editing system attributes, and browsing and editing the contents of context nodes. One must note that browsers and editors in NCM are seen as methods associated to the nodes and not as objects, as, for instance, in NoteCard.

The presentation specifications can be stored as an attribute of entities, or alternatively, as a type of terminal node. When presenting a node, the presentation specification defined explicitly by the user bypasses the specification defined by the context node (as seen in the definition of context) that contains the node; this in turn bypasses the node presentation specification defined by the link used to reach the node, which bypasses the node presentation specification defined within the node. Each node type has a presentation specification default, that is used if none of the presentation specifications presented above was defined.

The links that touch an instantiation in NCM can be defined in any context node present in any perspective of the node. The notion of *visible link* is used to determine which links actually touch a node instantiation.

If N_1 is a node and $P = (N_1,....,N_m)$ its perspective, then a link l is visible from P by N_1 with an anchor identifier i_1 ,if and only if:

- l is in N_2 and (N_1,i_1) is one of the end points of l; or
- l is visible from P by N_2 with an anchor identifier i_2, and the anchor value of the anchor identified by i_2 is a pair (N_1,i_1).

A link l is visible from P by N_1, if and only if:

- N_1 is in the node path of one end point of l, and l is in N_2 ,or N_3 , ... or N_m; or
- l is visible from P by N_1 with some anchor identifier.

A presentation issue already mentioned in section 2.1 is the spatial-temporal presentation relationship, specified in links and anchors. This topic is discussed in detail in [SSCC94], which presents the HyperProp presentation model.

Each hypermedia application may require specific forms of navigation, which can be programmed using presentation primitives offered by HyperProp. Besides, the system offers support for some navigation mechanisms.

We call *depth* navigation the action of following the nesting of composite nodes. This allows the user to move up and down the composition hierarchy. Navigation through links is essential to the idea of hypermedia, and is also provided. The model also provides navigation through queries. A user can arrive at a specific node by describing properties this node satisfies.

Browsers for contexts (and for the subclasses of contexts described in section 3) show a pictorial view of a hyperdocument (or of parts of it). Navigation in browsers is another predefined form of navigation in NCM.

The system also supports navigation through *trails*, allowing users to follow tracks established in previous sessions or defined as default tracks in the document itself. A trail is a specialization of a composite node that contains an ordered list of nodes (and the corresponding perspective), including trails. A node may be in the list in more than one position.

A trail may only represent tracks within a specific context. We say that the trail is *associated* to the context. A context may have, for instance, an attribute listing all the trails that are associated to it, or the most important ones.

After starting navigation over a trail, a user can issue a *next* command that will automatically jump to the next item of the ordered list of components. The *previous* command gives the previous component and the *home* command gives the first component of the list. Among other things, trails are useful in constructing a linear document from a hyperdocument. Like Intermedia [YaMe85] a special *system private base trail* keeps track of all navigation made during a session, so that a user can move at random from node to node, and go back, step by step. Note that this can be one way to create a trail.

2.3 - Related Work

The concept of context node generalizes the homonym concept introduced in the Neptune system [DeSc86,DeSc87], which was in turn based on some ideas from PIE [GoBo87]. It also generalizes Intermedia's webs [Meyr86], Notecard's fileboxes and browsers [Hala88], and the hierarchy structures made from Tree items of KMS [AkCY88].

The use of a NoteCard browser as a composition mechanism is not appropriate. It does not permit, for instance, the inheritance of the links of its several components. This limitation and others come from the fact that NoteCard does not associate any semantics to the browser nodes. Fileboxes present similar problems. For instance, NoteCard does not differentiate a reference link from a Filebox inclusion link, treating link navigation as depth navigation. Context nodes allow depth navigation in nested composition and, through the notion of perspective of a node, allow inheritance of links, thus subsuming the functionality of NoteCard's Browsers and Fileboxes.

Tree items in KMS allow the structuring of hierarchical documents. The inclusion relation of a context node in NCM is far more general, allowing hierarchical structures among other benefits. Context nodes allow the partitioning of networks thus subsuming the functionality of Neptune's contexts. Nesting in context nodes generalizes Intermedia's webs functionalities.

The HyperPro [Oste92] context and the HyperBase [ScSt90] composite object are very similar to NCM context nodes. It is not clear, however, how they address the problems of a perspective of a node and the anchoring in nested nodes.

As for anchoring, NCM provides the same facilities as Intermedia and Neptune, allowing the definition of anchoring regions inside the nodes, both source and target nodes. In KMS, NoteCard and HyperCard the target anchor is a whole node.

In Intermedia, attributes can be attached to links and anchors. Attributes in NCM, like in HAM (Neptune's storage subsystem), give semantic to the objects, but can be attached to nodes and links. Anchors are attributes both in NCM and HAM. Anchors in HAM are attributes of links. Anchors in NCM are attributes of nodes, thus allowing that changes in a node do not imply changes in links.

None of the related work mentioned above supports presentation specifications for objects, or spatial-temporal relationships between objects, such as those defined in NCM.

3 - Versioning

This section is organized as follows. We first define, in section 3.1, some requirements for version control mechanisms in hypermedia systems. We then extend, in section 3.2, the Nested Context Model to include versioning. Finally, in section 3.3, we compare our versioning approach to related work.

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.

Finally, we observe that, in order to save space, rather than storing each module of the version group in its entirety, only differences between the modules may be stored; a mechanism known as delta technique. The disadvantage of this technique is that retrieving a specific version requires computation - starting from a whole element, and applying changes. Another problem comes from the fact that delta algorithms are media specific; the same algorithm will not be efficient for both text and audio. We believe that memory saving techniques, such as representing only changes rather than wholes, is a task for the storage layer, through its storage algorithms, and should not be part of the data model. Thus, delta techniques will not be considered a requirement of the versioning mechanism.

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. Returning to our example about a book structure, the authors may define different chapter organizations for distinct classes of readers by defining different user context nodes containing the same nodes (the same chapters), but with a different set of links. All such user context nodes can be seen as different views of the same book.

The basic NCM hierarchy, shown in Figure 1, 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 2 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 3.2.6. Context nodes are subclassed, originating five new classes: *annotation, public hyperbase*, *private base*, *version context*, and *user context*. User contexts must be used in the same way as the context nodes were used in the basic model, as a facility for generic provision of views, nesting, and hierarchy. The difference is that a user context can only contain terminal nodes, and user context nodes, whereas context nodes may contain any type of context node, terminal nodes and trails. Context nodes, as previously defined, now constitute an abstract class, in the sense that no instances of it may be defined. This class remains in the hierarchy because it factors out common behavior of the classes beneath it.

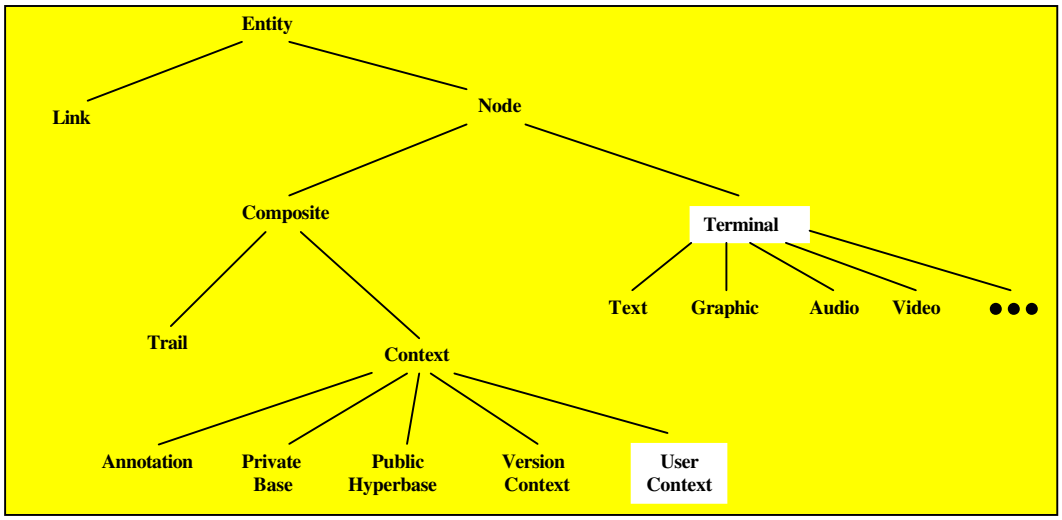

Figure 2 - NCM Class Hierarchy

In NCM, only terminal nodes and user context nodes are subject to versioning, as seen in white background in the figure 2. 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 R6 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.

The browser of a version context node, as in any composite node, shows a pictorial view of the components, that may not have any relation to their ordering. For example, the browser of a version context node can exhibit the versions ordered in time and not as a derivation graph.

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 (see [Casa91, Soar92]). 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.

Version contexts also help meeting R2. In Software Engineering there are two levels of versioning. The lowest level corresponds to the different modules that make up the programs. Naturally, all versions of a module may be grouped in a version context node. The other level is the configuration, that is, the description of which modules the program is made from, and how the modules should be put together to compose the program. A configuration can be modeled by a user context node. The nodes in the user context node will correspond to the software components (modules) of the system and the links to the various dependencies between the components, such as the dependency between source and object code. The match between configurations and user context nodes is quite reasonable because different configurations may share the same components, as different user context nodes may share the same nodes, but the relationships between components are specific to a configuration, as links are private to a user context node. Configurations can also be interpreted as versions of the same object and group together in a version context, helping to meet R2.

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 a 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 R6). 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.

The corollaries mentioned in the previous paragraph may seem excessively stringent at first sight. One could, for instance, consider it useful to have virtual committed user context nodes grouping nodes which are possibly uncommitted, in order to reflect recent work, eventually resulting in the presentation of uncommitted work. However, it is important to remember that, every user context node, even if virtual, still represents a grouping of other nodes. Hence it makes no sense to consider a grouping committed when some of the grouped entities are not committed. Moreover, we will see in the next section that committed nodes can be made available for public access, while uncommitted nodes cannot. Thus, allowing virtual user context nodes to return uncommitted components would imply in public nodes with non-public components, which also does not makes sense.

It may also seem rather restrictive not to allow an uncommitted node to be the source of derivation of new 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 node N automatically implies in the creation of a new version of the committed node from which N was derived.

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

3.2.4.1 - Basic Definitions

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.

3.2.4.2 - Version Operations in Private Bases and in the Public Hyperbase.

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.

In HyperProp presentation model, as in the Dexter model, a function called *converter* is responsible for returning a node version given a node and its presentation specification (helping to meet R13 and R14). This function is also responsible for the conversion of an anchor to its visible (or audible) manifestation. When a user wants to store modifications made in a new version, an inverse function of the *converter* is used to convert it back to the format in which it will be stored in the public hyperbase, as a committed node.

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.

3.2.4.3 - Additional Remarks

Some systems avoid the cognitive overhead of version management by creating new versions implicitly, either at regular time intervals, or as a side effect of other commands. The problem with timed version creation is that a lot of versions, which do not represent a consistent step in the evolution of a work, are created. The same problem may happen with version creation as a side effect of other commands. Using private bases, versions can be implicitly created in a controlled manner.

Just as tasks in [Haak92], private bases may guide automatic version creation and version identification (helping to meet R6 and R4). Each time a committed node is modified through a private base, an uncommitted version of this node, where the modifications will be made, is automatically created in the same private base, through the use of the check-in operation previously described. The creation of the new version may then trigger the version propagation algorithm, explained in section 3.2.5.

In section 3.2.2, we discussed how to specify current versions in a version context nodes. Selection of a current version based on a query language is a powerful technique, but increases the cognitive overhead for the user, who has to specify a selection criterion each time a link is created. In order to avoid this cognitive overhead, every version context node has two special anchors, defined by default queries, for retrieval of a current version. One of these default queries is specified in the version context itself. The other one is specified in a more general

way (helping to meet R6 and R4), in an attribute of the private base. When a link is created, the destination node is examined. If the node is a component of a version context not explicitly specified by the user (either directly or through a query), the link will be created using the selection criterion defined in the private base *PB* which contains the context node where the link is in. If this query is not specified in this private base, the link will use the default query defined in the version context.

Any of the navigation mechanisms offered by HyperProp can be used to visit a node. Frequently, navigation is based on a query. If each time a query was resolved, for instance in depth navigation, a new version was created in the private base, the work session would soon become a profusion of versions (consider the case when the user is navigating up and down a given perspective). To avoid this, once a query is resolved in a private base, its result becomes permanent for that base (intuitively, this means "for the rest of the work session").

Typically, a hyperdocument will be a highly dynamic entity, having several components defined through queries. It is sometimes necessary to record a static configuration (a *snapshot* facility), helping to meet R5 and R9. As previously mentioned, when a query is resolved in a private base *PB*, the resulting version becomes the permanent result for that query in *PB*. When nodes in *PB* are moved to the public hyperbase, the user may choose to store the static configuration (with the mentioned queries resolved) present in *PB* instead of the original dynamic configuration (stored by default).

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 3 illustrates this problem. The initial hyperbase, schematically shown in Figure 3(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 3(b), if exhaustive version propagation is applied.

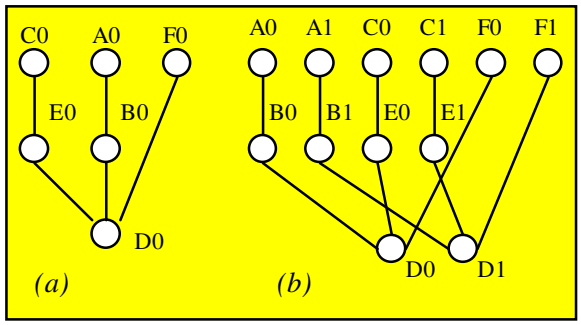

Figure 3 - Proliferation of versions

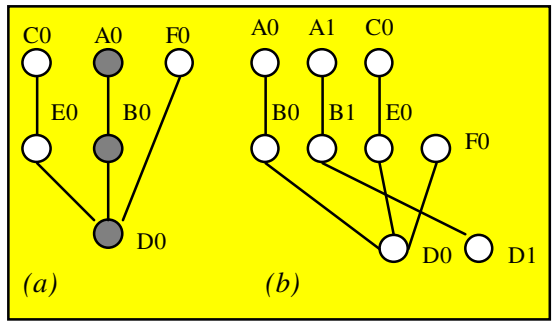

Figure 4 -Propagation guided by perspective

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, R6 and R10.

Figure 4 illustrates these points. Assume that the initial hyperbase is the same used in Figure 3(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 4(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.

3.2.6 - Annotations

Facilities for users to annotate recent work (of others or their own) are important for cooperative and exploitative development. An annotation consists of a comment (in any format or media: text, sound, etc.) and holds references to the versions it annotates and references to the versions that are considered replies to the statement contained in the annotation.

We define an *annotation* as a specialization of a context node that groups together sets of links, terminal nodes, user context nodes and trails. Intuitively, one or more nodes in this context node contain the remarks made by the user; private base's links from these nodes to different points in the same private base indicate the nodes being commented and nodes containing replies to these comments. Annotations can only be included in private bases.

Annotations allow the introduction of new remarks referring to committed nodes, without the creation of new versions, thus helping to meet R7.

To conclude, we observe that the version support mechanisms we described may be tailored by the applications. Versions can be either automatically created or created by explicit commands. These can be used by the applications to implement various versioning policies, thus meeting R11.

3.3 - Related work

Versioning has been investigated especially 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 [KaCB86] and Office Automation [Zdon85]. Detailed approaches to the problem of handling object versions can be found, for example, in [AhNa91, ChKi86, KaCB86, Katz90, Kim87, WoKL86, DiMa91]. Some work has also been done on version support in hypertext systems [DeSc87, Oste92, Haak92]. This section compares the NCM solution mainly with respect to these last mentioned works.

In PIE [GoBo87], a layer groups changes of several components into an identifiable unit. Layers can be put on top of each other, with the top ones dominating the lower ones, to combine changes to a version of the system. Layers are collect into PIE contexts. Note that the concept of context in PIE is totally unrelated to the concept of context in NCM. The notion of context as used in this paper closely resembles the notion of contexts in Neptune. Like in HyperPro [Oste92] and CoVer [Haak92], only the final result of layers superposition are represented in NCM. We believe that the memory saving techniques, such as representing only changes rather than whole versions is a task of the storage layer, and should not be part of the data model. Besides, the retrieval of a specific configuration always requires computation. In PIE, it is also possible to shuffle layers in arbitrary ways, which might result in inconsistent combinations. The model does not offer selection of nodes based on queries, and it is not easy to see how this facility could be provided.

In the extended version of HAM [DeSc87], a link points to a specific version, or to a *current* element; this is always the newest element in a version group. There is no freedom to specify the notion of current version, as in NCM. We say that HAM supports a *time based* versioning mechanism. In NCM versions are organized in a graph, and can be selected independently of the time of their creation. In Neptune, it is not possible to track the derivation history; if a new version is not derived from the current version, but from an older one, this derivation cannot be recorded anywhere. HyperPro organizes versions in a tree-like structure, and is therefore more general than HAM. In NCM, version context structures are organized as acyclic graphs, as in CoVer. This, in conjunction with the provision of explicit link inclusion, allows for the representation of derivation of a version from multiple nodes. Explicit link inclusion can also be used to add derivation information that cannot be automatically inferred by the system.

Like CoVer, HAM supports link versioning, although it is not clear in either system how versioned links appear to the user or how to navigate in a version group of links. In our first prototype we do not consider versioning of links, since we believe that versioning of context nodes will be enough. The problem with versioning of links is whether links should be

considered as independent objects or as values of a relation maintained by the context node. One advantage of treating links as attributes of context nodes is to avoid the cognitive overhead of naming links. With the addition of actions, conditions and synchronization information to NCM links, link versioning may become interesting and is one of the topics for our future research.

In Neptune new versions are created at each editor save. One problem with this approach is that a lot of versions, which do not represent consistent steps in the evolution of the hypertext, are created. NCM provides several mechanisms to avoid useless proliferation of versions, such as version propagation, private base definition, node status definition, two different version creation primitives, etc.

Our committed and uncommitted states are very similar to the "frozen" and "updatable" notions used in HyperPro and CoVer. However, in HyperProp, these states are used in several decisions that aid automatic version creation, differently from those versioning models. In addition, the inclusion of the obsolete state seems to be very useful in system management; this is an open issue that we will treat in the near future.

In HAM, nodes are classified as *archived*, *nonarchived* or *append-only*. Changes in an *archived node* create a new version of the node. Changes in a *non archived node* do not create a new version. When an *append-only node* is modified the new content is added to the previous content. NCM, like HyperPro, permits the change of some attribute (defined by the user) values without creating new versions (in our understanding, CoVer does not have this facility). HyperPro provides this facility associating the immutability aspect to the type of the node. For each entity type it specifies which attributes can be mutated after an entity has been frozen; for nodes it also specifies which types of links can be attached to a frozen node. NCM goes further. Type immutability specification is only a default definition that can be bypassed in a particular instance of a node. We believe that the content of a node gives more information as regards its immutability than its type; for instance, what is the difference between a node content presented (versioned) as a text node or as an audio node, through some conversion process? We also believe that the definition of the links that can be attached to a frozen node is a responsibility of the user context node where the link is included. If a new version of some entity will be created by a change in a link, or by the inclusion of a link, this entity will be a user context. Therefore, for consistence, user context nodes must define the immutability of its links. It should be noted that, in NCM, links are attributes of context nodes, which is consistent with our solution.

NCM also allows the inclusion of annotations in a private base without creating any versions. CoVer annotations are more general than ours. CoVer integrates annotations into the task structure: it records which task has produced an annotation, and if anybody has already set up a task to work on the annotation.

In HyperPro, a context provides a place to attach the selection criterion used by its generic version links. The default selection criterion selects the newest version from the version group the link refers to, though not newer than the cut-off date that is set when a context is frozen. Selection criteria in NCM may be other than time based. Like CoVer, the selection criterion is a query that can return more than one alternative (for example: all the versions created by Jane).

It is not clear from [Oste92] how HyperPro treats selection criteria other than its default. We also believe that sometimes we want to use different selection criteria within a context, therefore we also allow the selection criterion to be attached to a link, or to an anchor value of the version context node, with higher priority of resolution than the selection criterion defined in a composite node (in our case the private base and not the context). However, we agree with HyperPro that having the possibility to define a selection criterion in a more general way (in contexts in HyperPro, and in private bases and version context in NCM) can be of great value in lowering the cognitive overhead. CoVer lacks this facility.

We also believe that the selection criterion provided by HyperPro contexts may still result in a great cognitive overhead in documents with many nested contexts, since the selection criteria must be defined in each context. A related problem arises when we want to change the selection criteria of a whole work session. In HyperPro, we will have to change the selection criterion of each context, and maybe create a new version of each context. Take as an example a document, with many nested contexts, accessed by John and Mary. In her work session, Mary wants to work with the newest versions edited by her. On the other hand, John wants to work with the newest versions edited by him. Shall we have versions for all the contexts, with the different criteria defined by John and Mary? One solution to the two afore mentioned problems is to allow the inheritance of the selection criteria in a nesting. Our solution is to attach the selection criterion not to the context node, but to the private base node.

Neither mobs in CoVer nor version groups in HyperPro (concepts similar to NCM version context) allow the maintenance of queries for dynamic references as version contexts in NCM do. In NCM, queries can be specified in anchors. This facility is very important, once it allows the definition of several different selection criteria as we navigate inside a nested context. Queries specified in anchors also allow the definition of a general default query for all links (independent of the context in which they are included) that refer to a specific version context.

HyperPro (and CoVer) allows the creation of new versions of its atomic nodes (concept similar to NCM terminal nodes), as well as of its contexts. As in NCM, if a HyperPro context node or Cover composite node is made committed, all of its node components are also made committed. In HyperPro and CoVer, however, just the check-in primitive of NCM is provided. There is nothing similar to the open primitive, which is very important in limiting version propagation, as mentioned in 3.2.4. In HyperPro, there is no implicit version creation, except the operation of freezing a context. In CoVer and NCM, it is possible to let the system play a more active role in the generation of new versions than in HyperPro. Like tasks in CoVer, private bases in NCM can guide automatic version creation. Each time a committed node is edited in a private base, an uncommitted version of this node, where the modifications will be made, is automatically created in the same private base, as in CoVer.

To support exploratory development we have to solve the problem of how to freeze a specific state. The entire structure the author is working on needs to be frozen. This should not be done manually, entity by entity. NCM provides several ways to freeze sets of nodes. We can commit user context nodes and thus commit all of its components. A private base can also be shifted to the public hyperbase. All these operations subsume facilities found in HAM, HyperPro and CoVer to support exploratory development.

When a query is resolved in a private base *PB*, the resulting version becomes the permanent result for that query in *PB*. This enables the storage of static configurations, where each node is a specific version and not a version context node, an unsolved problem in HyperPro, and not even mentioned by the other systems.

To support sets of coordinated changes, several notions are included in NCM that we cannot find in HAM, HyperPro and CoVer. Among these we can include the open primitive and version propagation. Automatic version propagation guided by the perspective is a very important mechanism to avoid useless proliferation of versions.

Intermedia [Meyr86] provides concurrent access to the hypermedia network. The facility is however very simple and does not represent any real support to cooperative work. To support cooperative work, an authoring environment must naturally allow users to share information. However, the environment must also provide some form of private information, for security reasons as well as to allow fragmentation of the hyperbase into smaller units so as to reduce the navigation space. HyperPro does not supply facilities for cooperative work, in opposition to private bases and the public hyperbase in NCM and tasks in CoVer.

NCM provides the same support for trail navigation as Intermedia which has, indeed, influenced our decisions about trail creation and navigation. Our trail concept, however, extends Intermedia's, providing nesting when we treat a trail as a special type of composite node.

4 - HyperProp Architecture

We introduce in this section a brief description of the HyperProp architecture. The explanation of the layers and interfaces will be followed by a discussion about object organization and a discussion on how it relates to the concepts of the nested context model introduced in earlier sections. The reader must report to reference [SoCC93] for more details.

The architecture comprises three layers and four interfaces, as shown in Figure 5.

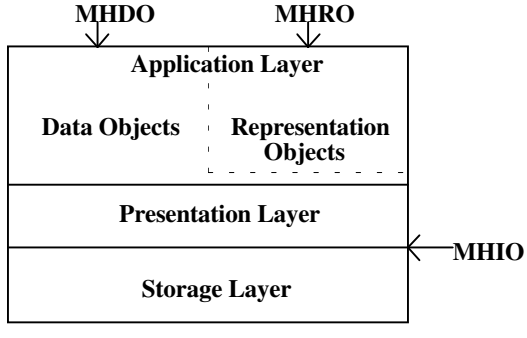

Figure 5 - Layered Hypermedia Architecture

From the point of view of the layered architecture, the notions of public hyperbase and private bases of the NCM should be understood as follows. In general, all node objects managed by the

various layers correspond to nodes in different bases. The storage layer manages all nodes in the public hyperbase, whereas the presentation layer creates nodes in the applications' private bases (application layer) and moves nodes from a private base to the public hyperbase.

The *storage layer* implements persistent *storage objects*, that have a unique identifier and a specific type, as well as other attributes (in NCM these objects make up the public hyperbase and version context nodes). It offers an interface for hypermedia data interchange, called the *multimedia hypermedia interchangeable objects interface* (MHIO Interface).

The MHIO interface is the key to providing compatibility among applications and equipments, since it establishes two points at which the storage and the presentation layers must agree: (1) the coded representation for the multimedia objects to be interchanged, which corresponds to the ISO MHEG standard [MHEG93]; and (2) the messages, requests, confirmations etc., used by these layers to ask for the required object, content or action. These two points together are the subject of the ITU T.170 series of recommendations (not yet provided) that will include the ISO MHEG standard as its T.171 recommendation. Special applications and other hypermedia systems may directly use this interface.

The *application layer* introduces the *data objects* and *representation objects* (similar concepts can be found in [PuGu91]). A data object is created either as a totally new object or as a local version of a storage object, adorned with new (non-persistent) attributes that are application-dependent. It contains methods to manipulate the new attributes, as well as methods to manipulate information originally pertaining to the storage object, if it is the case. The storage format of a data object corresponds to an internal concrete representation of an MHEG object. A representation object class is a specialization of a data object class with new methods to exhibit the multimedia data contents in the format most appropriate to that particular use of the data. A representation object therefore acts as a new version of a storage object, derived from a data object, as defined in R13 and R14 of section 3.1. Representation objects are also directly accessible to the applications and offer, in a sense, different views of data objects (in NCM these objects make up the private bases).

The application layer offers two interfaces for hypermedia data manipulation, called the *multimedia hypermedia data objects interface* (MHDO Interface), and the *multimedia hypermedia representation objects interface* (MHRO Interface), which contain the methods associated with the data and representation objects, respectively, among others. Typical applications will directly use just the MHRO interface, while special applications may use both interfaces.

The main purpose of the *presentation layer* is to convert to and from the storage format of the data objects used by particular applications and platforms and the storage format of the storage layer, or the coded representation for the multimedia objects defined by the MHIO interface. We note that the presentation layer does not implement any of the methods associated with data objects.

Therefore, to access multimedia data, an application proceeds roughly as follows. Using query or navigational facilities of the hypermedia system, it first indirectly identifies a storage object

containing the desired data and requests the creation of a data or representation object corresponding to it.

The architecture of HyperProp provides a framework for building monolithic hypermedia systems, as well as for introducing hypermedia features into a given application. It guarantees flexibility, for example, by letting an application interact through interfaces at distinct levels and by isolating the storage mechanisms, which can then be tuned to specific applications. It also increases interoperability by offering an MHEG object interchange interface.

To conclude this section, figures 6 and 7 show a generic model for a client server implementation that leaves the application layer and the presentation layer on the client side and the storage layer and again the presentation layer on the server side.

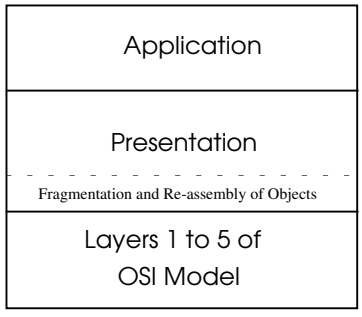

Figure 6 - Client Architecture

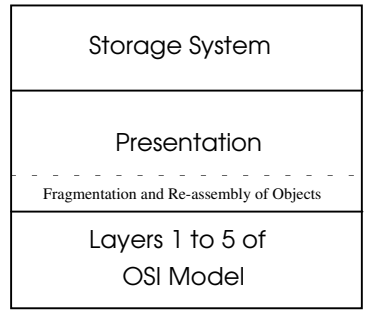

Figure 7 - Server Architecture

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.

To conclude, we again observe that the version support mechanisms we described in this paper, although based on the Nested Context Model, may be adapted to any model offering nested composition nodes, such as HyperBase [ScSt90], HyperPro [Oste92].

Acknowledgments:

The authors wish to thank Guido Souza, Maria Júlia Lima, Paulo Jucá, Sérgio Colcher and Thaís Batista for their contributions to the ideas here presented. The implementation work they conducted, jointly with other students, permitted the continuous refinement of the Nested Context Model.

REFERENCES

[AhNa91] Ahmed, R.; Navathe, S.B. "Version Management of Composite Objects in CAD Databases", *ACM SIGMOD*, Vol. 20, No.2. June 1991, pp. 218-227.

[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.

[CaGo88] Campbell, B.; Goodman, J.M. "HAM: A General Purpose Hypertext Abstact Machine". Communications of the ACM. Vol. 31, No. 7. July 1988, pp. 856-861.

[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.

[ChKi86] Chou H.T. e Kim W. "A Unifying Framework for Versions in a CAD Environment". *Proceedings of the International Conferenceon Very Large Data Bases*. August 1986, pp. 336-344.

[DeSc86] Delisle, N.; Schwartz, M. "Neptune: A Hypertext System for CAD Applications". *Proceedings of ACM SIGMOD '86*. Washington, D.C. May 1986.

[DeSc87] Delisle, N.; Schwartz, M. "Context - A Partitioning Concept for Hypertext". *Proceedings of Computer Supporteed Cooperative Work*. December 1986

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

[GRAQ91] Ghandeharizadeh, S.; Ramos, L.; Asad, Z.; Qureshi, W. "Object Placement in Parallel Hypermedia Systems". *Proceedings of Hypertext '91*. Texas. December 1991.

[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.

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

[HaMT87] Halasz, F.G.; Moran, T.P.; Trigg, T.H. "NoteCards in a Nutshell". *Proceedings of the ACM Conference on Human Factors in Computing Systems*. Toronto, Canadá. April 1987.

[KaCB86] Katz, R.H.; Chang, E.; Bhateja, R. "Version Modeling Concepts for Computer-Aided Design DataBases". *Proceedings of the ACM SIGMOD'86 International Conference on Management of Data*. Washington, D.C.. May 1986, pp. 379-386.

[Katz90] Katz R.H. "Toward a Unified Framework for Version Modeling in Engineering Databases". *ACM Computing Surveys*, Vol.22, No.4. December 1990, pp. 375-408.

[Kim87] Kim W.; Banerjee J.; Chou H.T.; Garza J.F.; Woelk D. "Composite Object Support in an Object-Oriented Database System". *Proceedings of the Second International Conference on Object-Oriented Programming Systems, Language and Applications.* Orlando, FL. October 1987, pp. 118-125.

[Meyr86] 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 1986.

[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.*

[SoCa93] 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. May 1993.

[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.

[SoCR93] 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. October 1993.

[SSCC94] 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. Submitted.

[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.

[WoKL86] Woelk D.; Kim W.; Luther W. "An Object-Oriented Approach to Multimedia Databases". *Proceedings of the ACM SIGMOD Conference on Management of Data.* Washington D.C. May 1986, pp. 311-325.

[YaMe85] Yankelovich, N.; Meyrowitz, N. "Reading and Writing the Electronic Book". *IEEE Computer*. October 1985.

[Zdon85] Zdonik, S.B. "An Object Management System for Office Applications". *Languages for Automation,* ed. S.K. Chang, Plenum Press. New York. 1985, pp. 197-222.