

## NESTED COMPOSITE NODES AND VERSION CONTROL IN AN OPEN HYPERMEDIA SYSTEM

LUIZ FERNANDO G. SOARES, NOEMI L. R. RODRIGUEZ<sup>1</sup> and MARCO A. CASANOVA<sup>2</sup>

<sup>1</sup>Dept. Informática, PUC-Rio, R. Marquês S. Vicente 225, 22453-900 Rio de Janeiro, Brasil

<sup>2</sup>Rio Scientific Center, IBM Brasil, Av. Pres. Vargas 824, 20071-001 Rio de Janeiro, Brasil

*(Received 18 June 1993; in final revised form 1 November 1993)*

**Abstract** — This paper presents a conceptual model for hypermedia systems that, among other features, supports versioning, permits exploring and managing alternate configurations, maintains document histories, supports cooperative work and provides automatic propagation of version changes. In general, the model was designed to minimize the cognitive overhead imposed on the user by version manipulation. The discussion about version control is phrased in terms of the Nested Context Model, but the major ideas apply to any hypermedia conceptual model that offers nested composite nodes. Briefly, nodes that represent versions of the same object at some level of abstraction are grouped together using the concept of version context, support for cooperative work is based on the idea of public hyperbase and private bases, and the problem of version proliferation is addressed using the concept of current perspective. Finally, the paper also presents a generic layered architecture for hypermedia systems with four major interfaces and shows how it matches the conceptual model.

*Key words:* 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 are being designed to run on heterogeneous platforms, or to be interconnected to offer more sophisticated multimedia services. These services 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 [5], HyperBase [21], MultiCard [20], Hyperform [25], HyperProp [23], and the World Wide Web. HyperProp provides not only a conceptual hypermedia data model, the Nested Context Model [3], but also an open architecture with an interface model which separates the data and object exhibition components [23]. Among other advantages, this permits designing interfaces that are independent of the exhibition platform, as well as adapting storage mechanisms to the performance and bandwidth requirements of particular applications.

We 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 were designed to minimize user cognitive overhead. Although the discussion is phrased as an extension to the Nested Context Model, the major ideas apply to any hypermedia conceptual model offering nested composite nodes.

Even though the need for version control in hypertext systems has long been recognized, the complexity of the interaction between version control and other requirements has apparently delayed the work in the area. In particular, this topic was not covered in the original description of the Nested Context Model [3]. The reader will find in sections 2.3 and 3.3 a comprehensive comparison of the Nested Context Model and the extensions proposed with related worked.

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 matches the concepts introduced in previous sections. Finally, section 5 contains the conclusions.

## 2. THE NESTED CONTEXT MODEL

This section describes the basic Nested Context Model, without version control. Section 2.1 defines the basic concepts, Section 2.2 addresses the presentation issues and Section 2.3 compares the model with related work.

### 2.1. The Basic Model

Figure 1 summarizes the basic class hierarchy of the Nested Concept Model (NCM) [3] (the order in which the concepts are introduced correspond to reading the tree in Figure 1 from top to bottom and from *right* to left).

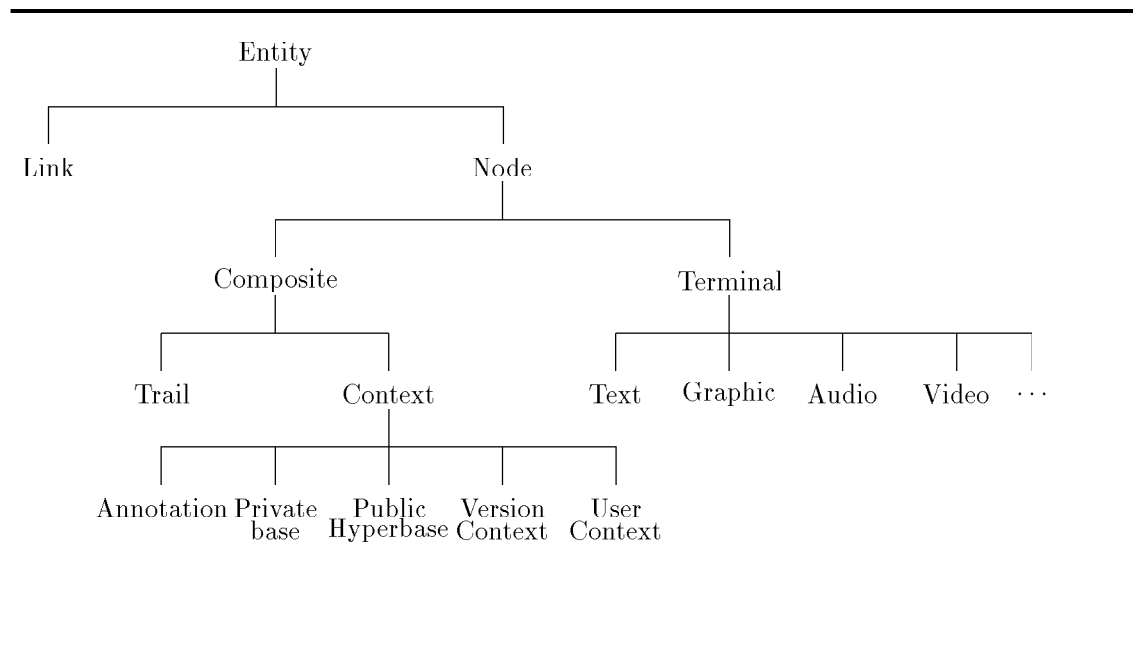


Fig. 1: NCM Class Hierarchy

An *entity* is an object which has as attributes <sup>†</sup> a *unique identifier*, an *access control list* and an *entity descriptor*. The unique identifier has the usual meaning. For each entity attribute, the access control list has an entry that associates a user <sup>‡</sup> or user group to his access rights for the attribute. The entity descriptor contains information determining how the entity should be presented to the user, as in the presentation specification of the Dexter model [22].

A *node* is an entity which has as additional attributes a *content* and an *anchor set*. Each element in the anchor set is called an *anchor* of the node and is an object which has as attributes an *id*, a *region* and a set of *condition/action pairs*. The exact definitions of node content and anchor region

<sup>†</sup>We will frequently use the name of the attribute of an entity to refer to the attribute value. When the context does not allow this simplification, we will explicitly use the term *attribute value*.

<sup>‡</sup>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. That is, anything or anybody that makes use of the services defined at the several interface layers of HyperProp

depend on the class of the node, except that the model requires that the special symbol  $\lambda$  must be a valid region value. The anchor id is the unique identifier for anchors. Condition/action pairs, an extension of the MHEG proposal, are discussed in detail in [24].

A *terminal node* is a node whose content and anchor set is application dependent. The model allows the class of terminal nodes to be specialized into other classes (*text*, *audio*, *image*, etc.). Intuitively, an anchor region defines a segment inside the terminal node content and the special symbol  $\lambda$  represents the entire terminal node content. The anchor set acts as the external interface of the terminal node, in the sense that an entity can access segments of the terminal node content only through the anchor set of the terminal node. Hence, anchors shield other entities from changes to the terminal node content. As an example, consider a text node with just one anchor whose region points to the second paragraph of the text. Then, any changes to the text must be reflected only in the anchor region and do not affect other entities. The anchor may have a condition/action pair indicating that the second paragraph must be shown blinking whenever the user navigates to the text through the anchor and wishes to exhibit it.

A *composite node*  $C$  is a node whose content is a list  $L$  of entities such that every base node of every link occurring in  $L$  is either  $C$  itself or a node occurring in  $L$  (the definition of base node of a link is given below - the definitions of link and composition node are indeed mutually recursive). Moreover, the region of an anchor of  $C$  must be either the special symbol  $\lambda$  or a sublist of  $L$  containing only nodes ( $\lambda$  again represents the entire node content). We say that an entity  $E$  in  $L$  is a *component of*  $C$  and that  $E$  is *contained in*  $C$ . We also say that a node  $A$  is *recursively contained in*  $B$  iff  $A$  is contained in  $B$  or  $A$  is contained in a node recursively contained in  $B$ . Note that the components of  $C$  may be ordered, which will be very useful when defining several navigation mechanisms, and that an entity may be included more than once in  $L$ . The anchor set of a composite node has about the same role as that of a terminal, except that anchors do not entirely shield every entity from changes to the composite node content. In particular, links are potentially sensible to changes to the content of composition nodes (this remark will be clarified immediately after the definition of links).

A *context node*  $T$  is a composite node such that  $T$  contains only links, terminal nodes, trails and context nodes, with no repeated entries, and every anchor region is either  $\lambda$  or does not contain repeated entries. Context nodes are specialized into *annotations*, *public hyperbases*, *private bases*, *version contexts*, and *user contexts*, but only the concept of user context node belongs to the basic Nested Context Model (*trails* will be defined in Section 2.2).

A *user context node*  $U$  is a context node such that  $U$  contains only terminal nodes, user context nodes and links.

A *link* is an entity with two special attributes, the *source end point set* and the *destination* or *target end point set*. The values of these attributes are sets whose elements, called *end points* of the link, are pairs of the form  $\langle (N_k, \dots, N_2, N_1), A \rangle$  such that  $N_{i+1}$  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 anchor of  $N_1$ . The node  $N_k$  is called a *base node* of the link.

As an example, consider a working document of a Drama Research Team about English Poetry of the XVI Century. Assume that the document is modelled as a user context  $E$  containing a user context  $S$ , grouping plays by Shakespeare, and another user context  $M$ , grouping sonnets by Christopher Marlowe. Assume also that  $S$  contains text nodes  $H$  and  $L$  representing the plays “Hamlet” and “King Lear” and that  $M$  contains a text node  $F$  representing “Dr. Faustus”. It may well be that the group wants to register a connection between “Hamlet” and “Dr. Faustus” (such a link could be used, for example, to register a connection between plays where the main theme is conflict). This link may be defined in  $E$  as  $\langle (S, H), r \rangle, \langle (M, F), s \rangle$ , where the anchors  $r$  and  $s$  allow the connection to be made more precise, for instance, by pointing to the sentences where the common concept first appears.

As another example, since  $S$  contains  $H$  and  $L$ , a link connecting these nodes may, in principle, be defined in  $S$  as  $\langle (H), i \rangle, \langle (L), j \rangle$ , where  $i$  and  $j$  are valid anchors for  $H$  and  $L$ . If one wants to create a link in  $E$  connecting  $H$  and  $L$ , one may define the link as  $\langle (S, H), i \rangle, \langle (S, L), j \rangle$ . Note the difference between defining the link in  $S$  and in  $E$ . A link defined in  $S$  will be seen by every document which includes  $S$  (the user context node grouping plays by Shakespeare

will probably be shared by several documents), while a link defined in  $E$  will be seen in  $S$  only by the readers of document  $E$ .

Links are always directional, although they can be followed in either direction. Multiple source and destination end points allow the definition of many-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. Furthermore note that a link  $l$  may have an end point of the form  $\langle (N_k, \dots, N_2, N_1), A \rangle$ , where  $l$  and the base node  $N_k$  belong to the same composition node, but node  $N_1$  is recursively contained (but not directly contained) in  $N_k$ . Although, this considerably simplifies modelling documents in some cases, if the content of  $N_{i+1}$  is changed by deleting  $N_i$ , the link  $l$  will have to be changed. That is, an entity - the link  $l$  - is affected by a change in the content of  $N_{i+1}$ .

Although not part of the structural conceptual model, but of the presentation model, a link also has a set of *node descriptors*, as in the presentation specification in the Dexter model, which contain information indicating how a referenced node should be presented to the user. Moreover, a link also has *condition/action* pairs, in conformance with the MHEG proposal [17]. A *trigger condition* is associated with changes in the values of attributes of the source end point of the link. Additional conditions may depend on the evaluation of values for attributes of the same or another entity. Actions defined within the link are processed on the indicated target nodes only if the conditions are satisfied and define spatial-temporal relations between the source end point and the target end points of a link. As an example, an action may indicate that, at the end of the presentation (condition) of entity  $Y$  (source end point), entity  $X$  (target end point) must be presented in a window specified by the coordinates  $(x, y)$ , assuming that  $X$  and  $Y$  have an attribute which specifies their exhibition state (sleeping, prepared, running, etc.).

To conclude, we outline the concept of virtual entity, an extension to the basic NCM whose formal definition is outside the scope of this paper (for a fuller discussion see [24]). A *virtual entity*  $E$  is an entity such that the value of at least one attribute  $A$  is an expression  $E$ , 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*. When some operation requests the value of  $A$ , the object resulting from evaluating  $E$  is returned. For example, a virtual node may have its content and anchor regions defined by expressions, which in particular implies that the anchor region will be computed when a link pointing to it is traversed. As another example, a virtual link may have end points computed when selected. Virtual links provide a powerful authoring tool, similar to that found in Microcosm [7].

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

## 2.2. The Presentation Model

The basic NCM treats hypermedia documents as essentially passive data structures. A hypermedia system must, however, provide tools for the user to access, view, manipulate and navigate through the structure of hypermedia documents. These aspects are addressed by the presentation (sub-)model of NCM, defined in conformance with the Dexter Model (DM) and briefly described in this section. The presentation model is also closely related to the versioning concepts discussed in Section 3.

Recall that the fundamental concept of the runtime layer of DM is the *instantiation* - the presentation of a component to the user - which exists within a DM *session*. Given a component and its presentation specification, a function called an *instantiator* is responsible for returning an instantiation. In particular, the instantiation of a node also results in instantiations of its anchors.

In NCM, sessions are modeled as private bases and instantiations are just versions created in a private base, as discussed in Section 3. For now, it suffices to say that the creation of instantiations in NCM is based on presentation specifications contained in *descriptors*, as already mentioned in section 2.1. In general, descriptors define methods for exhibiting or editing entities. The presentation model of NCM provides specific methods only for editing system attributes, and for browsing and editing the content of context nodes. Note that browsers and editors in NCM are

seen as methods associated with the nodes and not as objects as, for instance, in NoteCard.

A descriptor can be stored as an attribute of an entity or, alternatively, as a new type of node. When presenting a node, the 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 composite node that contains the node, if it is the case; second, that defined in the link used to reach the node; third, that defined within the node; and finally the default descriptor defined in the *nodeos* class.

Recall that the basic NCM allows different composite nodes to contain the same node and composite nodes to be nested to any depth. Thus, the presentation model introduces the concept of perspective to identify through which sequence of nested composite nodes a given node instantiation is being observed and the notion of visible link to determine which links actually touch the node instantiation. Formally, a *perspective* of a node  $N$  is a sequence  $P = (N_1, \dots, N_2, N_m)$ , with  $m \geq 1$ , such that  $N_1 = N$  and, for  $i \in [1, m)$ ,  $N_{i+1}$  is a composite node and  $N_i$  is contained in  $N_{i+1}$ . 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. Given a node  $N_1$  and a perspective  $P = (N_1, \dots, N_2, N_m)$ , we say that a link  $l$  is *visible from*  $P$  by  $N_1$  if and only if there is a composite node  $N_i$  such that  $N_i$  occurs in  $P$ ,  $N_i$  contains  $l$  and  $N_1$  occurs in some end point of  $l$ .

For example, assume that nodes  $A$  and  $Z$  contain node  $B$ , that in turn contains nodes  $C$ ,  $D$ ,  $E$  and  $F$ . Let us also assume that the composite nodes contain the following links:

<i>node</i>	<i>links</i>
$A$	$\langle (B, C), i \rangle, \langle (B, E), j \rangle$
$Z$	$\langle (B, C), r \rangle, \langle (B, D), k \rangle$
$B$	$\langle (E), s \rangle, \langle (D), t \rangle$ and $\langle (C), m \rangle, \langle (F), n \rangle$

Then, the exhibition of the instantiation of node  $C$ , through the perspective  $(C, B, A)$ , will show a link from anchor  $i$  of  $C$  to anchor  $j$  of  $E$  and a link from anchor  $m$  of  $C$  to anchor  $n$  of  $F$ , defined in  $A$  and  $B$ , respectively. The exhibition of the instantiation of node  $B$ , through the same perspective  $(C, B, A)$ , will show a link from  $B$  to  $B$ , that is defined in node  $A$ . On the other hand, the exhibition of the instantiation of node  $C$ , through the perspective  $(C, B, Z)$  will show a link from anchor  $r$  of  $C$  to anchor  $k$  of  $D$  and a link from anchor  $m$  of  $C$  to anchor  $n$  of  $F$ , defined in  $Z$  and  $B$ , respectively. The exhibition of the instantiation of node  $B$ , through the perspective  $(B, Z)$ , will show a link from  $B$  to  $B$ , that defined in node  $Z$ .

The HyperProp system [24] contains presentation primitives to define specific forms of navigation, but the presentation model of NCM offers only certain generic navigation mechanisms. Briefly, *navigation through links* is the essential idea of hypermedia. *Navigation through queries* allows a user to arrive at a specific node by describing properties the node satisfies. *Depth navigation* is the action of following the nesting of composite nodes, which lets the user move up and down the composition hierarchy. *Navigation through browsers* uses a pictorial view of a hypermedia document (or of parts of it). Finally, *navigation through trails* permits the user to follow trails established in previous sessions or defined as default trails in the document itself.

More precisely, given a composite node  $C$ , a *trail*  $T$  for  $C$  is a composite node containing only nodes and trails such that all nodes are recursively contained in  $C$  and all trails are trails for  $C$ . Moreover,  $T$  has as attributes *current*, whose value is an integer indicating the *current entity* of  $T$ , and *view*, whose value associates each occurrence of a node  $N$  in the content of  $T$  with a perspective  $(N, \dots, C)$ . We also say that  $T$  is *associated* with  $C$ . Trails come equipped with the following methods: *next* to jump to the next entity of the trail content, relative to the current entity; *previous* to move to the previous entity; and *home* to return to the first entity.

Note that the content of  $T$  is an ordered list, which means that a node or trail may occur in the list more than once. Moreover, each node occurrence is associated with a perspective from the point of view of  $C$ , so to speak. Among other things, trails are useful to linearize hypermedia documents. As in Intermedia [27], 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. In general, one may define a class of composite nodes that introduces a new attribute whose value lists all trails that are associated with each composite node in the class, or just the most important ones, according to some criteria.

### 2.3. Related Work

The concept of context node in NCM generalizes the homonym concept introduced in the Neptune system [5, 6], which was in turn based on some ideas from PIE [8], Intermedia webs [16], Notecard fileboxes and browsers [10], and the Tree Items of KMS [2]. In particular, NoteCard does not associate any semantics to the browser nodes and it does not differentiate a reference link from a Filebox inclusion link, treating link navigation as depth navigation. By contrast, NCM context nodes allows the definition of depth navigation in nested composition and induces the notions of perspective and visibility of links.

Also, HyperPro contexts [18] and HyperBase composite objects [21] 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.

In Intermedia, attributes can be attached to links and anchors. In contrast, attributes in NCM and HAM (Neptune’s storage subsystem) can be attached to nodes and links.

As for anchoring, NCM provides the same facilities as Intermedia and Neptune, allowing anchors to point to regions inside both source and target nodes. However, in KMS, NoteCard and HyperCard, the target anchor must be a whole node. Anchors are modelled as node attributes in NCM and as link attributes in HAM. Therefore, changes to the content of a node do not require modifying links in NCM, as they might in HAM.

NCM provides the same support for trail navigation as Intermedia, which has indeed influenced our decisions about trail creation and navigation. However, NCM extends Intermedia since, by treating trails as composite nodes, NCM allows the nesting of trails, which permits defining a trail for a composite document out of trails for its components. None of the related work mentioned above supports object presentation specifications or spatial- temporal relationships between objects, such as those defined in NCM.

## 3. VERSIONING

This section extends the basic Nested Context Model to include versioning. Section 3.1 lists some requirements for version control mechanisms in hypermedia systems. Section 3.2 presents the extensions to the model. Finally, Section 3.3 compares 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 [10, 11], in [18] and in [9]. 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 [18], 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 [9], we may add the following requirements:

- R11.** Tailorability – *“Versioning must be tolerable 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 a versioning mechanism 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.*

We believe that memory saving strategies, such as delta techniques, pertain to the storage layer and should not be part of the conceptual model. Thus, they will not generate requirements for the versioning mechanism.

### 3.2. Extending the Nested Context Model to Include Versioning

#### 3.2.1. Preliminaries

The basic NCM already meets Requirement R1 for version control mechanisms. Indeed, the problem of exploring alternative configurations of a document is trivially solved by creating, over the same set of nodes, alternative user context nodes that reflect distinct views of the same document and that are tuned to different applications or user classes. However, the basic NCM must be extended with new concepts in order to address the other requirements listed in section 3.1. Figure 1 also shows the new context node classes introduced - annotation, public hyperbase, private base, and version context - explored in detail in sections 3.2.2 to 3.2.5.

In NCM, only terminal nodes and user context nodes are subject to versioning, as seen in white background in Figure 1. Each attribute (including content) of a user context or terminal node may be specified as *versionable* or *non-versionable*. The value of a non-versionable attribute may be modified without creating a new version of the object. Changes to versionable attribute values have to be made on a new version of the entity, if it is already committed, as we will detailed in Section 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 content attribute of a version should 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 allowed without creating a new version of the object (also helping to meet R7).

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

To conclude, we observe that the version support mechanisms we describe 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.

### 3.2.2. 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). This notion is relevant only for user context and terminal nodes that have versionable attributes. The state is just a new node attribute whose value affects, and is affected by, the execution of certain operations.

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 the an specific operation or implicitly by certain operations the model offers (helping to meet R5). 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.

More precisely, 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 used to derive 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:

- 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 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 it is no longer referenced by a link or contained in another 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.

It may 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.

### 3.2.3. Version Contexts

To address the problem of maintaining the history of a document, we extend the NCM with the class of version context nodes. A *version context*  $V$  is a context node that contains only user context, terminal nodes, and links. Intuitively,  $V$  groups nodes that represent versions of the same entity, 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 (see Section 3.2.4.2),

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 criterion. 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 extended model, which includes virtual entities based on a query language, the reference may be made through a query. The query does not need to be part of the anchor of the version context, since it may be defined in a link (see section 2.1) and even in a more general way (helping to meet R5 and R4), as we will see when we discuss private bases in section 3.2.4. It should be noted that the query which defines the current version may return several versions (for example, “all versions created by John”), which can be interpreted as alternatives and presented as a user context (a version context view). Therefore, version contexts meet R4 since they provide an automatic reference update facility.

In the basic model of NCM, we defined an end point of a link contained in a composition node  $C$  as a pair of the form  $\langle (N_k, \dots, N_2, N_1), A \rangle$ , with  $k > 0$ , such that  $N_k$  must be contained in  $C$ , for all  $i \in [1, k)$ ,  $N_{i+1}$  is a composite node and  $N_i$  is contained in  $N_{i+1}$ , and  $A$  is an anchor of  $N_1$ . 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  $\alpha$ .

An application may, for example, use version contexts to maintain the history of a document  $d$  (R3), as well as for automatic reference update (R4), 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  $C_1, \dots, C_n$  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 [3]. 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 grouped 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 the node and not a version context. Support for this will be provided by the private base concept, defined in section 3.2.4.

### 3.2.4. Public Hyperbase, Private Bases and Annotations

#### Basic Definitions

In general, a cooperative environment must allow users to share information, must provide some

form of private information for security reasons and must permit 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 as follows. 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  $\mathcal{H}_B$ , that corresponds to public, stable information. The other subsets will form the *private bases*, used to model the user's interaction with a hypermedia document, 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  $\mathcal{H}_B$  must be committed or obsolete and, as in all hyperbases, if a composite node  $C$  is in  $\mathcal{H}_B$  then all nodes in  $C$  must also belong to  $\mathcal{H}_B$ .

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:

- a private base may be contained in at most one private base;
- 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 recursively contained in  $PB$ ;

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

Facilities that allow users to define annotations to recent work (theirs or of other users) are important for cooperative and exploitative development. Intuitively, 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.

More precisely, an *annotation*  $A$  is a special context node that groups together sets of links, terminal nodes, user context nodes and trails. Intuitively, nodes in  $A$  contain user comments and links in a private base  $B$  from these nodes to different points in  $B$  indicate the nodes being commented upon and the nodes containing replies to these comments. Annotations can only be contained 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.

### *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  $\mathcal{H}_B$ , then all terminal and user context nodes in  $C$  must also be moved into  $\mathcal{H}_B$ .

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 NCM, 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 the presentation model of NCM, as in the Dexter model, a function called *converter* is responsible for returning a node version given a node and its descriptor (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 difference in behavior between the *open* and *check-in* primitives. Let  $N'$  contain 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.

#### *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 large number 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 [9], private bases may guide automatic version creation and version identification (helping to meet R5 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 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 R5 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 NCM offers 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 have a large number 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 hypermedia document 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. We outline in this section how we approach this problem in NCM, referring the reader to [15] for a complete discussion.

Recall that a node may be contained in many different composite nodes in NCM. Thus, version propagation may cause the creation of a large number of often undesirable nodes. Figures 2(A) and 2(B) illustrate this problem. The initial hyperbase, schematically shown in Figure 2(A), has a node  $D_0$  that belongs to user context nodes  $E_0, B_0$  and  $F_0$ ; node  $E_0$  is in turn in  $C_0$  and  $B_0$  in  $A_0$ . The creation of a version  $D_1$  of  $D_0$  generates five new user context nodes, as shown in Figure 2(B), if exhaustive version propagation is applied.

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

Figures 2(C) and 2(D) illustrates these points. Assume that the initial hyperbase is the same

used in Figure 2(A), that the current perspective is  $(D_0, B_0, A_0)$ , shaded in Figure 2(C), and that version  $D_1$  of  $D_0$  is created. If  $B_0$  and  $A_0$  are committed nodes, then new versions of these two nodes are created, as in Figure 2(B). On the other hand, if  $A_0$  and  $B_0$  were uncommitted, then no new version of these two nodes would be created, but rather  $B_0$  would be altered to include  $D_1$  ( $A_0$  would be left unchanged), as in Figure 2(D).

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.

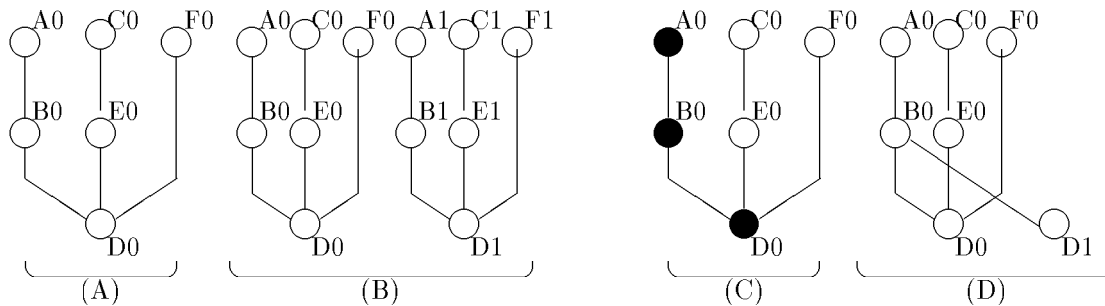


Fig. 2: Proliferation of versions (left) and Propagation guided by perspective (right)

### 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 [13] and Office Automation [28]. Detailed approaches to the problem of handling object versions can be found, for example, in [1, 4, 13, 12, 14, 26]. References [6, 18, 9, 16], against which we compare NCM in this section, specifically address version support in hypermedia systems.

In the extended version of HAM [6], a link refers to a specific version, or to the newest element (in time) in a version group. The user is not free to define the current version, as in NCM. 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. Slightly more general than HAM, HyperPro organizes versions in a tree-like structure, whereas CoVer and the NCM use acyclic graphs. 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 of these systems how versioned links are exhibited to the user or how the user navigates through a group of link versions. 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 link versioning 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, indeed, becomes a topic for future research.

The committed and uncommitted states of NCM are very similar to the “frozen” and “updatable” notions used in HyperPro and CoVer. However, in NCM, 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.

In HAM, nodes are classified as *archived*, *non-archived* 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 by 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 anyone has already set up a task to work on the annotation.

Both NCM and the HyperPro system assign considerable importance to the definition of criteria to select a version from a set of versions, in a global way, thus reducing the user cognitive overhead during the creation of links. CoVer lacks this facility. In HyperPro, 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, for instance, a query that can return more than one alternative (for example: all the versions created by Jane). It is not clear from [18] 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 region 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).

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 these two 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.

To support exploratory development NCM provides several ways to make a set of nodes committed (for example, one can commit a user context nodes and thus commit all of its components), private bases can be moved in block to the public hyperbase, etc. All these operations subsume

facilities found in HAM, HyperPro and CoVer to support exploratory development.

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

Intermedia [16] 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.

#### 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 [23] for more details. The architecture comprises three layers and four interfaces, as shown in Figure 3.

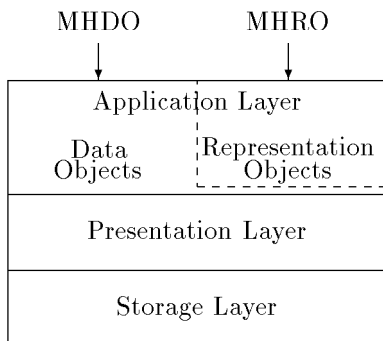


Fig. 3: Layered Hypermedia Architecture

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 equipment, 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 [17]; 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 [19]). 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.

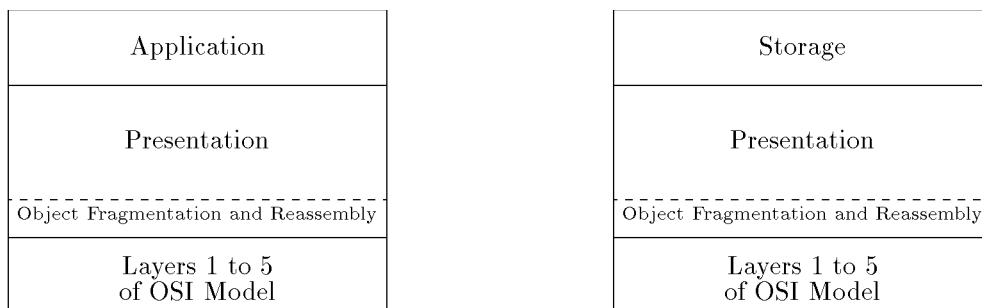


Fig. 4: Client (left) and Server (right) Architecture

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 con-

taining 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, Figure 4 shows 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.

## 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.

*Acknowledgements* — 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

- [1] R. Ahmed and S.B. Navathe. Version management of composite objects in cad databases. *ACM SIGMOD*, **20**(2):218–227 (1991).
- [2] R.M. Aksscyn, D.L. McCracken, and E.A. Yoder. Kms: A distributed hypermedia system for managing knowledge in organizations. *Communications of ACM*, **31**(7) (1988).
- [3] M.A. Casanova, L. Tucherman, M.J. Lima, J.L. Rangel Netto, N.R. Rodriguez, and L.F.G. Soares. The nested context model for hyperdocuments. In *Proc. 3rd ACM Conference on Hypertext*, San Antonio, Texas (1991).
- [4] H.T. Chou and W. Kim. A unifying framework for versions in a cad environment. In *Proc. of the 1985 Int'l. Conf. on Very Large Data Bases* (1985).
- [5] N. Delisle and M. Schwartz. Neptune: A hypertext system for cad applications. In *Proc. 1985 ACM Int'l. Conf. on Management of Data*, Washington, DC (1985).
- [6] N. Delisle and M. Schwartz. Context - a partitioning concept for hypertext. In *Proc. Computer Supported Cooperative Work* (1987).
- [7] A. Fountain, W. Hall, I. Heath, and H. Davis. Microcosm: An open model for hypermedia with dynamic linking. In *Proc. 1990 European Conference on Hypertext*. Cambridge University Press (1990).
- [8] I. Goldstein and D. Bobrow. A layered approach to software design. In *Interactive Programming Environments*, pp. 387–413. McGraw Hill, New York (1987).
- [9] A. Haake. Cover: A contextual version server for hypertext applications. In *Proc. 1992 European Conference on Hypertext*, Milano (1992).
- [10] F.G. Halasz. Reflexions on notecards: Seven issues for the next generation of hypermedia systems. *Communications of ACM*, **31**(7) (1988).

- [11] F.G. Halasz. Seven issues revisited. In *Proc. 3rd ACM Conference on Hypertext - Final Keynote Speech*, San Antonio, Texas (1991).
- [12] R.H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, **22**(4) (1990).
- [13] R.H. Katz, E. Chang, and R. Bhateja. Version modeling concepts for computer-aided design databases. In *Proc. 1985 ACM Int'l. Conf. on Management of Data*, Washington, DC (1985).
- [14] W. Kim, J. Banerjee, H.T. Chou, J.F. Garza, and D. Woelk. Composite object support in an object-oriented database system. In *Proc. 2nd Int'l. Conf. on Object-Oriented Programming Systems, Language and Applications*, Orlando, FL (1987).
- [15] M.J. Lima and M.R. Cavalcanti. Version propagation in hypermedia systems. In *Proc. IX Brazilian Symposium on Databases*, São Carlos, Brazil (1992).
- [16] N. Meyrowitz. Intermedia: The architecture and construction of an object-oriented hypermedia system and applications framework. In *Proc. Conf. on Object-Oriented Programming Systems, Languages and Applications*, Portland, Oregon (1985).
- [17] MHEG. *Information Technology - Coded Representation of Multimedia and Hypermedia Information Objects - Part1: Base Notation, Committee Draft ISO/IEC CD 13522-1* (1993).
- [18] K. Osterbye. Structural and cognitive problems in providing version control for hypertext. In *Proc. 1992 European Conference on Hypertext*, Milano (1992).
- [19] J.J. Puttress and N.M. Guimarães. The toolkit approach to hypermedia. In *Proc. 1990 European Conference on Hypertext*. Cambridge University Press (1990).
- [20] A. Rizk and L. Sauter. Multicard: An open hypermedia system. In *Proc. 1992 European Conference on Hypertext*, Milano (1992).
- [21] H.A. Schutt and N.A. Streitz. Hyperbase: A hypermedia engine based on a relational database management system. In *Proc. 1990 European Conference on Hypertext*. Cambridge University Press (1990).
- [22] M. Schwartz and N. Delisle. The dexter hypertext reference model. In *Proc. NIST Hypertext Standardization Workshop*, Gaithersburg (1990).
- [23] L.F.G. Soares, M.A. Casanova, and S. Colcher. An architecture for hypermedia systems using mheg standard objects interchange. In *Proc. Workshop on Hypermedia and Hypertext Standards*, Amsterdam, The Netherlands (1993).
- [24] G.L. Sousa and L.F.G. Soares. Synchronization aspects of the nested context hypermedia presentation model. Technical report, Dept. Informática, PUC-Rio, Rio de Janeiro, Brasil.
- [25] U.K. Wiil and J.J. Leggett. Hyperform: Using extensibility to develop dynamic, open and distributed hypertext systems. In *Proc. 1992 European Conference on Hypertext*, Milano (1992).
- [26] D. Woelk, W. Kim, and W. Luther. An object-oriented approach to multimedia databases. In *Proc. 1985 ACM Int'l. Conf. on Management of Data*, Washington, DC (1985).
- [27] N. Yankelovich and N. Meyrowitz. Reading and writing the electronic book. *IEEE Computer* (1985).
- [28] S.B. Zdonik. An object management system for office applications. In S.K. Chang, editor, *Languages for Automation*, pp. 197-222. Plenum Press, New York (1985).