NCM: A Conceptual Model for Hyperdocuments

Luiz Fernando G. Soares Noemi L. R. Rodriguez Marco A. Casanova

September 21, 1995

1 Introduction

The Nested Context Model, described initially in [CTL$^+$91], defined a conceptual framework for the definition, presentation, and browsing of hyperdocuments. 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.

This model is the basis for the Hyperprop system [SCC93], which is is a toolkit for the construction of hypermedia applications.

2 The Nested Context Model

Figure 1 summarizes the basic class hierarchy of the Nested Concept Model (NCM) [CTL$^+$91] (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).

An *entity* is an object which has as attributes [1] 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 [2] 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 [SD90].

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 depend on the class of the node. Condition/action pairs, an extension of the MHEG proposal [**?**], are discussed in detail in [SS94].

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 λ represents the entire terminal node content.

[1] 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*.

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

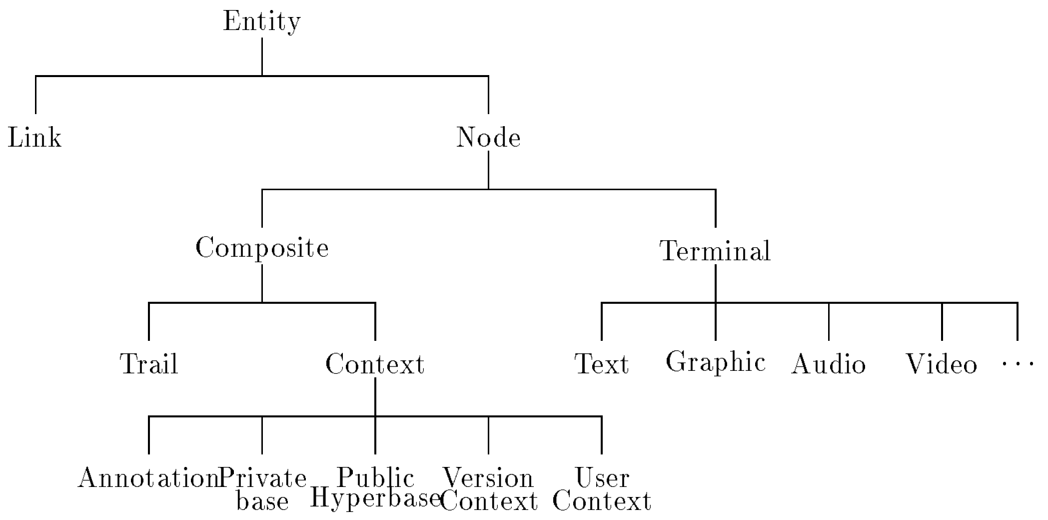

Figure 1: NCM Class Hierarchy

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 λ or a sublist of L containing only nodes (λ 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.

A *context node* T is a composite node such that T contains only links, terminal nodes, trails and context nodes, with no 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.

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 $< (N_k, ..., N_2, N_1), A >$ 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.

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 $< (N_k, ..., N_2, N_1), A >$, 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.

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 $\mathcal{H}$.

2.1 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. The presentation model is also closely related to the versioning concepts discussed in Section 3.

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 nodeïs 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 $P = (N_1, ..., N_2, N_m)$, 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 $P = (N_1, ..., N_2, N_m)$, 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.

The HyperProp system [SS94] 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 $(N, ..., C)$. 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.

3 Extending the Nested Context Model to Include Versioning

3.1 Preliminaries

The basic NCM already provides support for some 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 other version requirements. Figure 1 also shows the new context node classes introduced in this extension - annotation, public hyperbase, private base, and version context - explored in detail in sections 3.1.2 and 3.2.

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 will detailed in section 3.1.1.

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.

3.1.1 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. This notion is relevant only for user contexts 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.

3.1.2 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). 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 $(< v_1, i_1 >, < v_2, i_2 >)$ in V.

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.

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.

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.

3.2 Public Hyperbase, Private Bases and Annotations

Basic Definitions

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, private bases model the user's interaction with a hypermedia document, according to the paradigm (work session) proposed by the Dexter Model.

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.

3.3 Version Propagation

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 [LC92] 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.

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).

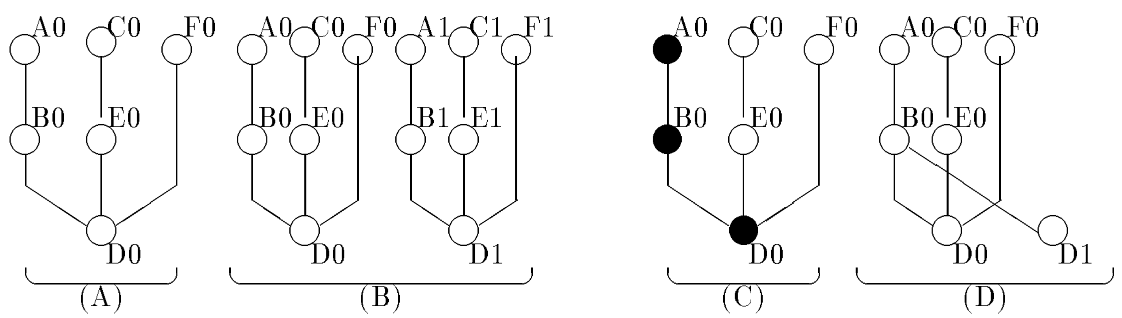

Figure 2: Proliferation of versions (left) and Propagation guided by perspective (right)

4 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.

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

[CTL+91] 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, December 1991.

[LC92] M.J. Lima and M.R. Cavalcanti. Version propagation in hypermedia systems. In *Proc. IX Brazilian Symposium on Databases*, São Carlos, Brazil, May 1992.

[SCC93] 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, April 1993.

[SD90] M. Schwartz and N. Delisle. The dexter hypertext reference model. In *Proc. NIST Hypertext Standardization Workshop*, Gaithersburg, January 1990.

[SS94] 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, 1994.