Structural and Cognitive Problems in Providing Version Control for Hypertext

Kasper Østerbye
Department of Mathematics and Computer Science, Aalborg University[*]

Abstract

This paper discusses issues related to providing version control in hypertext systems. Many of the software engineering issues in versioning also apply to hypertext, but the emphasis on linking and structure in hypertext raises some new problems. The issues can roughly be divided into two categories. Datamodel issues, which will be referred to as *structural issues*, and user interface issues, which will be referred to as *cognitive issues*. Both structural and cognitive issues will be described and divided into simpler problems which will be named and described, and it will be shown that composites serve as a good starting point for solving both structural and cognitive problems of versioning.

Keywords: Hypertext, Version Control, Data models.

1 Introduction

There are two main types of hypertext systems, those for *browsing*, and those for *authoring*. The former allow the user to browse through information provided by someone else, but not to add new information. These systems can be found at for instance museums, or as instruction books.

The latter type is typically used for development of products, such as the above mentioned hyperdocuments used at museums, or for e.g. software engineering. In such systems the very nature of production makes the hypertext evolve over time.

[*]Full address is at the end of the paper.

Version control is the discipline of controlling and tracking the evolution of a product over time. The main problem in providing version control facilities for a hypertext system is determining just what to keep versions of. Is it desirable to keep versions of the individual nodes and links, or do we only want to track entire "hyperdocuments"?

Software engineers face the same problem. Are versions kept of the individual modules or the entire program? In software engineering, the solution is to split the issues. Managing different versions of a single module is called version control, and managing the program as a whole is called configuration management.

This paper will discuss version control for production hypertext systems, with emphasis on five problems that are particularly important for hypertext:

- **Immutability of versions.** If an element is versioned, a specific version represents a *state* in the development. Will it then be possible to annotate it, or to add new attributes?
- **Versions of links.** Do we want versions of links, or should only nodes be versioned?
- **Versions of structure.** We would like to be able to return to a consistent previous state of the entire network, rather than to just a previous state of a single node or link. What is the right model for this?
- **Version creation.** Explicit version creation will result in a large cognitive overhead[1], due to the large number of entities in hypertext. How can version creation be made manageable?
- **Element selection.** When we create a link to a versioned element, cognitive overhead

[1]The term "cognitive overhead" is adapted from Conklin [2]. A general definition could be *the amount of mental effort that must be devoted to the non-task-related aspects of doing something.*

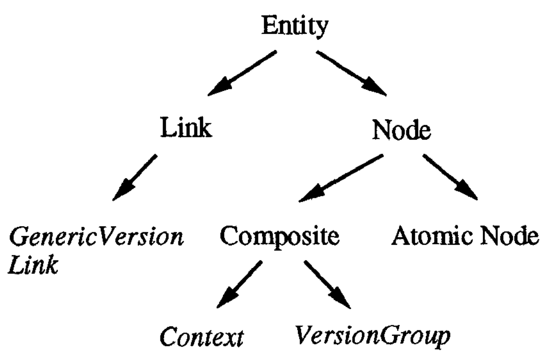

Figure 1: Datamodel. The types in italics pertain to versioning.

will be increased if we have to specify which version we want to point to. How can we implicitly provide the right selections?

Datamodel The datamodel used in this paper is illustrated in figure 1. The top of the hierarchy is an entity which allows attributes to be attached to all entities. Links are one-to-one, and can be anchored to nodes in both ends.[2] Nodes are entities which have contents, and are specialized into atomic nodes which do not contain other entities, and composites which do contain other entities. We will return to the datamodel in sec. 4.1 where context, version group, and generic version links are discussed.

The paper is organized as follows: First, a brief overview of state-of-the-art version control systems for software engineering is given. Second, the five problems are dealt with in detail. Finally, we discuss how we address these problems in The Aalborg Hyperstructure Programming Environment, (HyperPro) [15].

2 Versioning in software engineering

This section describes the state-of-the-art of version control and configuration management for file based software engineering environments. For a good introduction and overview paper, see Tichy [13].

[2]Therefore, if a composite is a specialization of a node, then we can maintain the simple notion that nodes are connected through links. We want to be able to link to a composite, but we do not want links between links. This is not important for this paper.

In software engineering there are two levels of versioning. The lowest levels are the different modules that make up the programs. Each module can exist in several versions, and all the versions of a module is often referred to as a *version group*.

The other level is the configuration. A configuration describes which modules the program is made from, and how those modules should be put together to make a program. If a module exists in several versions the configuration often describes how the version to be used in the final program is selected. A set of selected versions for a configuration is sometimes referred to as a *baseline*. The baseline describes which modules were selected, so that one can regenerate exactly the same program at a later time.
The state-of-the-art can be described as:

Version groups are organized as trees, with some support for merging versions from different branches into each other. A tree captures the evolution of a module. It also serves as the basis for compact representations (see delta techniques below).

Selection from version groups is based on the attribute values of the modules in the version group. Such attributes can typically contain information about authors, creation date, and release state (tested, experimental, etc.).

Configurations are described in a system model, which gives selection criteria and dependency information. The process of creating an actual running program consists of two phases; first a module must be selected from each version group, then the selected modules must be compiled and linked into a single program.

Baselines are supported in order to track which systems have been delivered over time. Just storing the configuration does not work, as the selection criteria might denote other modules at a later date. It is therefore important to be able to record a static configuration, where each module is a specific version and not a version group.

Change oriented. Changes performed on a number of different modules are grouped according to which overall change they are part of. To be able to track changes in this manner is especially helpful in two ways. It allows better management of change requests, bug reports etc, when the system explicitly supports a notion of changes across modules. And understanding the individual modules is made easier, because it is possible to find the other modules that were altered at the same time, which allows for understanding the individual change in relation to the overall change.

Delta techniques. Rather than storing each module of the version group in its entirety, only differences between the modules are stored. For large version groups this can help to save space. The disadvantage is that retrieving a specific version requires computation - starting from a whole element (typically the first or last version), and applying changes.

The main thesis of our approach to solving problems arising with respect to versioning in hypertext, is that we can provide the same division of labour as known from software engineering. Configurations are responsible for selection of versions. In our approach contexts take the place of configurations, but are further extended to handle versioning in hypertext. Nodes correspond to modules; nodes will normally be short, e.g. sections or paragraphs rather than chapters, or routines rather than files.

3 Versioning issues in hypertext

This section is devoted to a discussion of the five issues mentioned in the introduction, which we believe to be idiosyncratic to hypertext systems. A solution will be presented in section 4.

The mechanisms just outlined in section 2 will also be needed in version mechanisms for hypertext.

Version groups should be structured as revision trees. Just as in software engineering, it is evident that a new version will sometimes be derived directly from a version other than the most recent. *Selection, Configurations,* and *Baselines* will be redesigned especially for hypertext. *Change orientation* is very appealing in hypertext, where we can link change requests to the affected elements. *Delta techniques* pose a special problem because delta algorithms are media specific, the same algorithm will not be efficient for both text and sound.

3.1 Structural issues

We divide the hypertext versioning issues into structural issues, related to the datamodel, and cognitive issues, related to the user interface. We will first address the structural issues, and turn to the cognitive issues in section 3.2

Immutability of versions

In hypertext it might be too simplistic to have versions of nodes be completely immutable. While it is obvious that the contents of a version (i.e. a frozen node) should be immutable, it is less clear how links and attributes should be treated.

During explorative development it ought to be possible to annotate nodes that have already been created, If new *annotation* links cannot be added to a frozen node, annotation of frozen versions is not possible - we believe it should be possible to annotate anything in a hypertext. However, other types of links (which we might call *substance links*) may be seen as carrying important semantics of the node, and when the node is frozen, it should be ensured that no changes are made to these links, and that no new substance links can be attached to the node. Thus, there is a need to freeze some types of links, while allowing others to remain unfrozen.

Adding new attributes to a node might also be attractive. Assume that after the node was created, a new tool was introduced into the system. That tool might want to store some information at the nodes.

This discussion cannot be kept completely independent of actual data models. The issue is whether or not anchors are kept as part of the node.

When anchors are part of the node, we must soften the requirement that nodes are completely frozen, as we will otherwise not be able to add annotation links, because new anchors cannot be added. Not all aspects of the node can be frozen, so we will call this to "jel" the node. Means must be found that allow us to specify which aspects are frozen and which are jelled.

When anchors are separate from the node we are faced with the opposite problem. Freezing the node does not prevent annotations, nor does it prevent addition and change of substance links. When the node is frozen this should radiate to some types of anchors also. We will call this "frost-radiation". Again means must be found that allow us to specify which aspects should be hit by frost-radiation when the node is frozen.

For both models methods for controlling change and addition of attributes must be found. Keeping anchors separate from nodes does not change the fundamental problem: that we must specify which aspects are jelled and which are frozen.

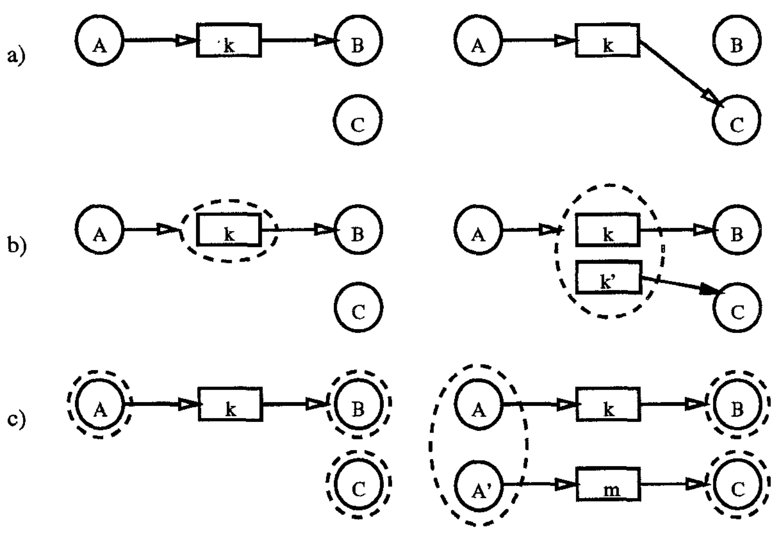

Figure 2: Problems with versions of links. Circles represent nodes, boxes links, and dashed lines represent version-groups. In a) there is no versioning, in c) links are versioned, and in c) nodes are versioned.

Versioning of links

So far only versioning for nodes has been addressed. One must also consider a separate versioning for links. If one changes the destination[3] of a link the old destination is preserved, and a new version of the link is created. While this symmetry between versioning on nodes and links seems intuitively appealing, it raises some difficult problems. When we want to follow a link from a node, we are faced immediately with the problem of which link to follow. One can, of course, have different selection criteria, e.g. the newest link or the one which was valid before a specific time.

As we see it, the real issue is a choice between models. Is changing a link considered changing the source/destination-node of that link? Or symmetrically, is changing a node considered a change to the links that start in that node?

This problem is illustrated in figure 2. In a) there is no version control, so when the destination of a link is changed, the link k just points to D, rather than B.

If we turn to b), we have introduced versioning on links. If the link k is changed, a new version of that link is created, and it has become ambiguous whether k or k' should be used.

In c), we have version control on nodes only. One aspect of freezing a node is that the outgoing links cannot be changed. Thus if we want to redirect the link to point to C, is is necessary to create a new version of A, and make an entirely new link m which points to C.

[3]The discussion is symmetric for the source of the links.

Sometimes a specific type of link carries important semantics of the product we are developing. In that case changing the link can be seen as changing the nodes it connects. Versioning on links, as in b), causes a problem with respect to frozen nodes. Assume that the node A is frozen. Then by allowing links to be versioned, there is a way to change the k link, even though it should not have been possible. The point is that the versioning of links is sometimes tied strongly to the versioning of their source node.

A fourth possibility is to provide versions of both nodes and links at the same time. The general problem in doing this is how to maintain consistency between different nodes and links. It is worth noting that, as can be seen in figure 2 b) and c), versions of links can be simulated by versions of nodes. The opposite is also the case.

Another problem with versioning of links is whether links should be considered as objects or as values of a relation. This problem is illustrated in figure 3.

At time t_0 node A is connected to node B through the link k of link-type R. This is the situation in version 1 of the network. At time t_1 and t_2 we are experimenting with the hypertext. The

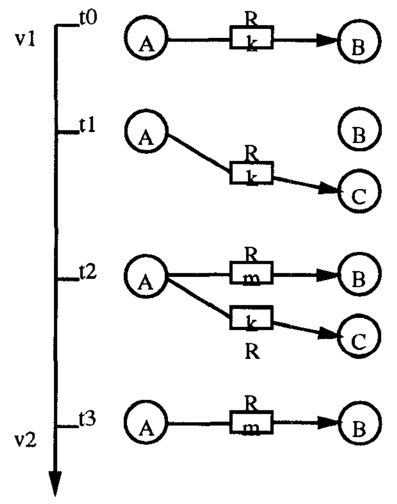

Figure 3: Links as objects or links as relation values: Are v1 and v2 equivalent though a different link is used?

link k has been redirected, and points to C. At t_2 we create a link m of type R, connecting A to B as in the beginning. Finally at t_3 we delete the link k, leaving us in a situation where A is connected to B through the link m of type R. This becomes the new version of the hypertext (assuming that we have worked elsewhere in the hypertext too, the situation is not unrealistic). The question is now, do we consider the final situation (t_3) to be different from the first (t_0)? It can be argued that the interesting thing is not the actual links, but the relation (R in this case) they represent. Tracing the versions of the individual link objects does not properly reflect changes in the relation.

Though there are numerous problems regarding versions of links, versioning of the individual links is an obvious way to capture fine grained changes to the structure of a hypertext as discussed by Hicks et. al in [11].

Versions of structure

Just as one wants to be able to have different versions of a single entity, 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.

The extended version of HAM described in [4] supports a time based versioning mechanism. To return to a previous structure therefore means to find all the entities that were the newest at some point in time. If we are not satisfied with the simple time oriented versioning, but want a revision tree based versioning, we can no longer depend only on the time when an entity was created, because the versions are organized in a tree where there is no defined order among the branches of the tree.

In systems with a datamodel supporting composites, these can serve as a base for versioned structure. This will be elaborated on in section 4.

3.2 Cognitive issues

Version creation

In [2] Conklin points out that there is a problem in naming nodes. If the user is forced to come up with names for each and every node that is created, it will distract the author's attention from the actual subject. Similarly, if the user must explicitly create new versions of nodes all the time, and maintain some consistency among different versions, it will further distract the author's attention from the real work to be done. Following Conklin, this will be called *cognitive overhead.*

In [5] Garg and Scacchi distinguish between *exploratory* and *assembly line* (which we will call anticipated) development. In anticipated development there is a consistent system, which needs to be changed in some way. An element is selected from the version group, and a successor created. When the changes are done the element is frozen. In exploratory development a version is typically created because the author gets an idea which involves changes here and there, but it is not known exactly where. In that case the author wants to freeze the current document, so that state can later be recovered, should the experiment fail.

We believe that the typical situation in hypertext is the exploratory approach, but in cases where the document is somewhat stable, anticipated changes will occur too.

If we want to support exploratory development, the problem is how to freeze the state. Obviously, it is not sufficient to freeze just a single node, the entire structure the author is working with needs to be frozen. This should not be done manually entity by entity. In software engineering manual freezing is feasible because of the few but large chunks of information, while in hypertext several hundreds of nodes may be affected. It is therefore important that the author can freeze the entire structure with one command.

One way to avoid the cognitive overhead of version creation is to create new versions implicitly. Either at regular time intervals, or as a side effect of other commands. In Neptune [3] new versions are created at each editor save. The problem with both timed version creation and the Neptune approach is that a lot of versions, which do not represent a consistent step in the evolution of the hypertext, are created.

Element selection

A prominent problem, when introducing versions of nodes, is to determine which element in the version group the link points to. In HAM there are two possibilities. A link can point to a *specific* element, in which case the link always points to the same element. Or the link can point to the *current* element, meaning the newest element in the version group.

In a situation where the elements in the version group are organized in a tree, it is not clear which is the current element. In the proposal for versioning in hypertext put forward by Weber in [14] element selection is done through the use of a query language.

Element selection based on a query language is a powerful technique, but increases the cognitive overhead for the user, who has to specify a selection criteria each time a link is created. In some interfaces the author must specify the type of the link, and sometimes give a link name when a link is created. If a selection query must be specified each time a link is created, the author is discouraged from making links. Special care must therefore be taken at the interface level to simplify the use of selection links. Weber suggests the use of task related information as a means for doing this.

An argument against attaching the selection query to the links can be put forward. We may want the link to use different queries in different situations. Imagine that we are writing a paper; for some purpose we want the paper to be short, and the link should therefore select the shortest elements. In a different situation we may want to include all the details and other elements should therefore be selected.

4 Versioning in HyperPro

In this section the HyperPro [15] approach to the above questions will be examined. We will first look at the structural issues (datamodel), and then at the cognitive issues (interface).

4.1 Extended datamodel

HyperPro extends the basic datamodel with three new types of entities to take care of versioning as was shown in figure 1. The entities are organized in an inheritance hierarchy, so properties defined for a supertype also hold for a subtype. The types are as follows.

Entity. The general entity provides attributes, which are key/value pairs attached to the objects. All information is stored as attributes. Entities can be members of composites, and must be members of at least one Context, see below. *In our datamodel, it is up to the user to specialize the built-in types of the system to suit the specific needs of concrete applications. One aspect of specifying an entity type is to define the immutability aspect for the type. For an entity type X, we specify which attributes can be mutated after an entity has been frozen; for nodes we also specify which types of links we can attach to a frozen node.*

Node. The node is the basic entity for storing contents. We require all nodes to have an attribute for contents and a name. *Nodes are versioned.*

Link. The link is an entity that relates a source node to a destination node (or subtypes of nodes). *There is no versioning for links.* There are two reasons for this: first, we believe that versioning of composites will work as well, which is illustrated later in figure 5; second, we believe that links are mostly used for relations, in which case it is less interesting to track individual links (cf. section 3.1).

Composite. The contents of a composite node are a sequence of entities, repetitions allowed. The main purpose of a composite is to serve as a generalization of version group and context, and as a basis for new user defined types, say answers to queries. *Composites are versioned.*

VersionGroup. A version group is a special kind of composite, where all entities are considered versions of the same entity. The individual versions are related through revision links and organized in a tree-like manner (inspired by RCS [12]), in order to track the development history. *Version groups are not subject to versioning.*

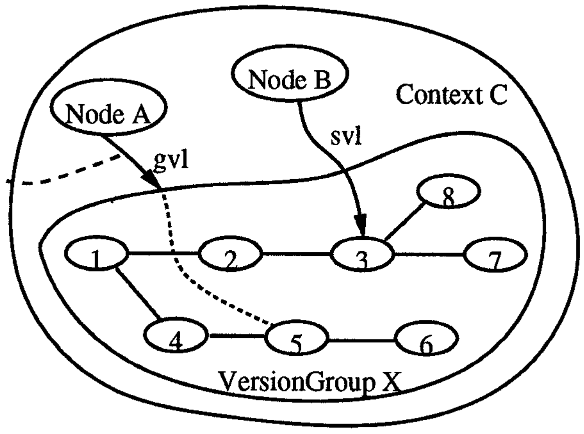

Figure 4: Versioning in HyperPro. Both specific version links (svl) and generic version links (gvl) are provided.

Context. Context is a composite whose purpose it is to provide the notion of hierarchical structure. All entities are members of at least one context[4]. A Context also maintains a selection criteria to be used by generic version links. The default selection criteria selects the newest version from the context, though not newer than the cut-off date that is set when a context is frozen. Thus the default selection is basically time based. *Contexts are versioned, which provides us with an explicit versioning of structure.*

GenericVersionLink. A generic version link is a specialization of a link, which computes its destination node by selecting a node from the version group it physically refers to. The selection is based on the selection criteria found in the current context of the link; the user interface of the system provides the notion of current context. Like other entities, generic version links can be members of several contexts.

Summary. In figure 4, we have illustrated our extended data model. We provide an explicit notion of a version group (X). The version group is a specialized composite containing the different versions, here 1 through 8. It is possible to create a *specific version link* (svl) linking to a specific version, in the figure from node B to version 3. It is also possible to create a *generic version link* (gvl) from a node (A) to a version group (X). The destination of a generic version link is found by applying the selection criteria in the context (C) to the version group. This gives the actual destination node (say 5).

4.2 Interface issues

At the interface level, we must make sure that we are not faced with the cognitive problems outlined earlier, as well as provide an interface that is convenient and where minimal concern is needed about versioning issues. The design principle is that if versioning is not used, no penalty should be payed. If one can do with just time based versioning as in HAM or KMS, that should be nearly transparent to the author.

Each element can be part of several contexts (but must be a member of at least one). In particular a generic version link can be a member of several contexts, which makes it ambiguous which selection criteria should be used.[5] One important task for the interface level is to maintain a notion of *current context*. A context will typically represent some kind of coherent structure, e.g. a paper. The generic version links will use the current context for their search criteria.

New nodes and links are created as members of the current context. When a link is created, the destination node is examined. If the node is versioned (is a member of a version group), the link will be created as a generic version link by default. The selection criteria to be used by the link is the one associated with the current context. We thus address the element selection criteria in a two step fashion. The actual selection is controlled by context, and the specific/generic is controlled by the user, with generic as default.

The problem of version creation is addressed using the current context too. While it is possible to explicitly create new versions of individual nodes, a new-version command is available at the context level. The command freezes all the context's members, then a successor to the current context is created (contexts are versioned), which becomes the new current context and which contains the same elements as the old context. The member list of the old context is frozen, making it possible to return to a previous state of the hypertext structure - in the form of a previous context[6]. All

[4]The only exception is the outermost composite in a hyperdocument.

[5]This could of course have been avoided, but it is quite useful to experiment with different contexts with different selection criteria.

[6]Actually this is not guaranteed, as a generic version

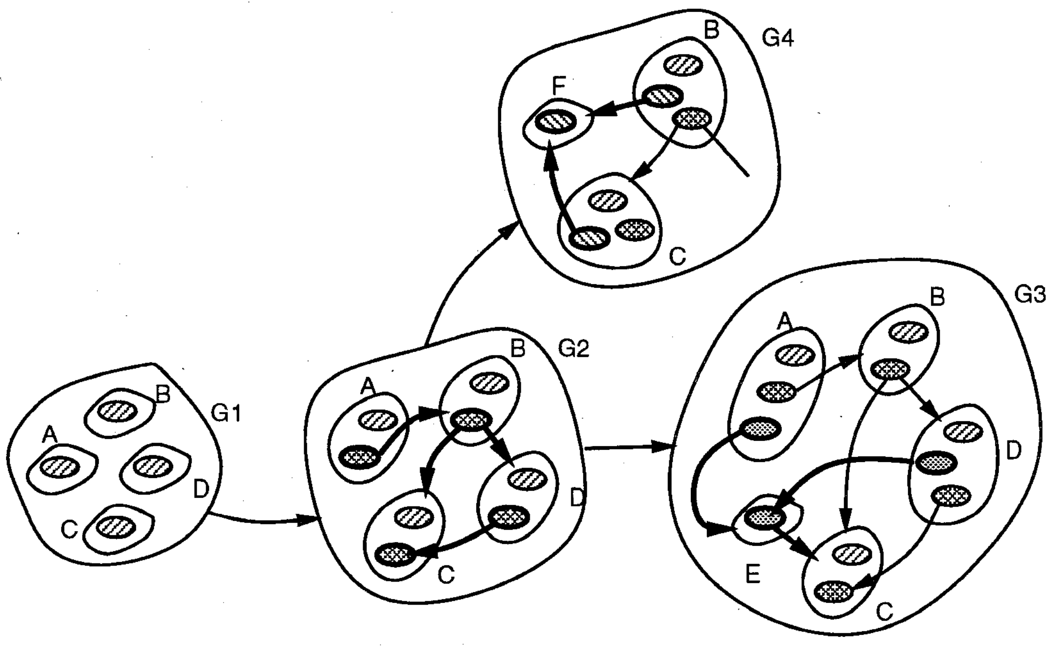

Figure 5: Versioning of contexts. Context versions can be used to simulate both PIE-layers and Intermedia Webs.

changes will now happen in the new context, including such actions as removing elements from the member list of the context.

We thus address the two cognitive problems through the notion of contexts *which is an important notion, both at the user interface level, and in the datamodel of our system.*

5 Related work

We will first relate our model to the PIE system [8] and Intermedia [7] through a specific example. In figure 5 we have outlined the development of a context G in four different versions. The initial version G_1 contains four version groups A, B, C, and D, and no links. This version could be the beginning of a paper, where some initial material has been copied/pasted into G_1. In G_2 we edit a bit in each element, and connect the various nodes. In G_3, a version group E has been added and connected to two new versions of A and D. G_4 is another successor to G2. Here the version groups A and D have been deleted, and a new one F, has been added. The link from B to D (in G_2) is also a member of G_4, but the destination of the link is not. This represents a link across contexts.

link can select nodes which are created later. Therefore a timestamp is included in the frozen context, making it possible to include a cut-off time in version selection.

Halasz has at two occasions [9, 10] argued that versioning in hypertext should take its starting point in the PIE system [8]. PIE uses *layers*, which are collected into contexts (we will here refer to PIE contexts as P-contexts, and our contexts as H-contexts to avoid confusion). A layer is a set of entities and their interrelations. Layers can be placed on top of each other, with the top ones dominating the lower layers. If a layer X has nodes A and B, and the layer Y has nodes B" and C, placing Y on top of X yields a combination containing A, B", and C, as B" dominates B (B" being a change to B). P-contexts are such combinations of layers. P-contexts and H-contexts differ at a few significant points. H-contexts are not represented as a set of layers; only the "sum" of the layers is represented. In figure 5, the "bolded" versions and links in G_2 are the ones that are added in G_2. The "bold" part can be seen as a PIE layer. We believe that memory saving techniques, such as representing only changes rather than wholes, is a task for the storage layer, and should not be part of the datamodel. Similarly, at the user interface level, we want tools with greater power to explore differences between versions than those provided by layers. The PIE notion of layers has two drawbacks seen from our point of view. First, it is possible to shuffle layers in arbitrary ways, which might give inconsistent combinations. Second, it is not clear how generic links can be provided; P-contexts are static in that they do not allow selection of nodes based on queries, only the order of the layers influence which nodes are selected.

Intermedia [7] stores nodes independent of the web of links. This makes it possible to develop different webs (structures) for the same set of nodes. Something similar is possible using versions of contexts, as illustrated in figure 5, where G_1 only contains nodes, and the links are added in G_2. Several other successors of G_1 could be created, representing different webs.

The notion of contexts as used in this paper closely resembles the notion of contexts in Neptune [4]. In Neptune time based versioning was chosen, which makes it easy to return to the previous state of a context, by just seeing which versions were the most current at some given time. Neptune provides two kinds of links, specific and current links. Thus there are no means to have links select on the basis of attributes or other information. It is not possible to track the derivation history; if a new version is created with the outset in an old one, it is not recorded anywhere. Just as in our approach, contexts are used to provide the notion of a *hyperdocument*. In our approach the contexts not only serve as a grouping mechanism, but also provide us with a place to attach the selection criteria used as default.

In [14] Anja Weber addresses versioning in hypertext systems. Her work differs from our in a couple of ways. First, the versioning under consideration is designed especially for publishing. This influences the way in which Weber attempts to deal with the cognitive overhead of version generation. With the outset in specific work routines (tasks) it is possible to let the system play a more active role in the generation of new versions than envisioned in this paper. In our opinion the idea of letting the generation of versions be guided by the tasks performed by the users can be transferred to any situation where the development of the hyperdocument follows strict organizational rules. Second, the proposal of Weber also differs in that she provides versions of links, which we rejected. However, it is not clear from the paper how versioned links appear to the user.

David Hicks et. al [11] also address versioning in hypertext systems. Their proposal differs from ours in the underlying datamodel, where anchors are separate entities independent of nodes. This leads to consistency problems when both nodes and links are versioned, as is the case in their proposal. Like us, they emphasize the importance of versioning composites, but they do not address the interface aspects of versioning.

The Document Integration Facility (DIF) described by Garg and Scacchi in [6] is a hypertext system intended for software engineering. Its main focus is on anticipated development, where there is a central manager who divides the overall structure of the hyperdocument. From our point of view there are a couple of interesting aspects in relation to this. Freezing does not happen at a node by node basis, but at the level of what corresponds somewhat to contexts (forms). Version selection criteria seem to be at the same level; that is, not individual nodes, but rather the same criteria are used for all entities in the form.

KMS [1] provides a limited sort of version control. Each node (frame in KMS) can exist in a number of versions. New versions are created by freezing a node. The next edit operation performed on the node automatically creates a new version of the node. KMS addresses the version creation problem in the same way as we do, by allowing for freezing a node which serves as a container for a number of other nodes. In KMS there are no composites, but the notion of hierarchical links gives the same effect. When a node is frozen so is all its descendants, giving the effect that a whole document was frozen at once. To our knowledge KMS does not support any notion of generic version links.

6 Conclusion

We have presented two major categories of problems that arise when we want to provide version control for hypertext systems. We have also shown that those problems can be solved by elaborating on the notion of configurations as known from software engineering environments. This solution is interesting because it combines configurations and composites, and therefore serves to further emphasize the need for composites in hypertext systems.

We have made a controversial decision in our system: Links are not versioned, the main reasons being that we felt it was not necessary, and that we found it difficult to come up with the right user interface to support it. Whether individual versioning of links is needed after all is one of the topics for our future research.

At the moment the solution described in this paper is being implemented as part of the Aalborg Hyperstructure Programming Environment [15]. The current status is that the whole datamodel described here is operational. The system is, however, still an early prototype, so we have not yet had any experience in using it. As a preliminary example, we have stored this paper in its different versions, and are able to move back and forth between versions both at the individual node level, and at the context level.

Author's address: Department of Mathematics and Computer Science, Institute for Electronic Systems, Aalborg University, Fredrik Bajers Vej 7, DK-9220 Aalborg Ø, Denmark. Phone: +45 98158522, Fax: +45 98158129, Email: kasper@iesd.auc.dk.

References

[1] R. Akscyn, D. McCracken, and E. Yoder. KMS: A distributed hypermedia system for managing knowledge in organizations. *CACM*, 31(7):820–835, July 1988.

[2] Jeff Conklin. Hypertext: An introduction and survey. *IEEE Computer*, September 1987.

[3] Norman Delisle and Mayer Schwartz. Neptune: a hypertext system for CAD applications. In Carlo Zanilo, editor, *Proceedings of the International Conference on Management of Data (SIGMOD'86)*, pages 132–143. ACM, 1986.

[4] Norman Delisle and Mayer Schwartz. Contexts - a partitioning concept for hypertext. *ACM Transactions on Office Information Systems*, 5(2):168–186, April 1987.

[5] P. K. Garg and W. Scacchi. Composition of hypertext nodes. In *Online Information 88. 12th International Online Information Meeting, London 6-8 December, 1988, Proceedings Volume 1*, pages 63–73. Learned Information, Oxford and New Jersey, 1988.

[6] P. K. Garg and W. Scacchi. A software hypertext environment for configured software descriptions. In *Proceedings of the International Workshop on Software Version and Configuration Control*, January 1988.

[7] L. N. Garrett, K.E. Smith, and N. Meyrowitz. Intermedia: Issues, strategies, and tactics in the design of a hypermedia document system. In *Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW-86)*, pages 163–174, December 1986.

[8] Ira P. Goldstein and Daniel G. Bobrow. A layered approach to software design. Technical Report CSL-80-5, Xerox PARC, Xerox Corporation, Palo Alto Research Centers, 3333 Coyote Hill Road, Palo Alto, CA 94304, December 1980.

[9] Frank G. Halasz. Reflections on notecards: Seven issues for the next generation of hypermedia systems. *CACM*, 31(7):836–852, July 1988.

[10] Frank G. Halasz. "seven issues": Revisited. Slides from keynote talk at Hypertext'91, December 1991.

[11] David L. Hicks, John J. Legget, and John L. Schnase. Version control in hypermedia databases. Technical Report TAMU-HRL 91-004, Hypertext Research Lab. Texas A&M University, July 1991.

[12] Walter F. Tichy. RCS - a system for version control. *Software - Experience and Prctice*, 1985.

[13] Walter F. Tichy. Tools for software configuration management. In *Proceedings of the International Workshop on Software Version and Configuration Control*, 1988.

[14] Anja Weber. Versioning issues in hypermedia-publishing environments. Arbeitspapire 542, Gesellschaft für mathematik und dataverarbeitung MBH, Schloss Birlinghoven, Postfach 12 40, D-5205 Sankt Augustin 1, June 1991.

[15] Kasper Østerbye, Kurt Nørmark, and Hans Mejdahl Jeppesen. Hyperstructure programming environments. Presented at the fifth Nordic Workshop on Programming Environment Research. Obtainable through the author., January 1992.