

Version Control in Hypertext Systems

David L. Hicks
John J. Leggett
John L. Schnase

Hypermedia Research Lab

TAMU HRL-91-004

JULY 1991

Table of Contents

1. INTRODUCTION	1
2. GENERAL VERSION CONTROL CONCEPTS	3
2.1 Revisions and Variants	3
2.2 Data Model	4
2.2.1 Tree Data Model	5
2.2.2 Layered Network Data Model	5
2.3 Storage Mechanism	7
2.3.1 Separate Delta Files	7
2.3.2 Conditional Compilation	8
2.4 Support for Collaboration	9
2.5 Configuration Management	10
3. VERSION CONTROL IN HYPERMEDIA SYSTEMS	12
3.1 Versioning Requirements of Hypertext Systems	12
3.2 Previous Work	14
3.3 Issues of Versioning in Hypertext Systems	15
3.3.1 Versioning Data	15
3.3.1.1 Data model	15
3.3.1.2 Storage Management	16
3.3.2 Versioning Structure	18
3.3.2.1 Definitions	18
3.3.2.2 Implementation of Anchors	19
3.3.2.2.1 Inter-Node Level	19
3.3.2.2.2 Intra-Node Level	20
3.3.2.3 Versioning Links	20
4. FUTURE RESEARCH	23
5. SUMMARY	25
6. REFERENCES	27

Version Control in Hypertext Systems

DAVID L. HICKS
JOHN J. LEGGETT
JOHN L. SCHNASE

1. INTRODUCTION

The need for version control arises naturally in many applications. Application areas which support the process of evolutionary development are particularly likely to have version control requirements (Zdonik, 1986). Version control systems help manage the intermediate revisions characteristic of such environments and assist in maintaining the relationships between revisions. Support for version control is particularly important when the number of intermediate revisions grows large, or the relationships among them become complex.

Historically, much of the research and development activity related to versioning has been in the software engineering area and the software engineering field has benefited considerably from these efforts. Over the years many useful tools have been developed to assist software engineers in the task of managing large software systems consisting of numerous modules, each possibly having several revisions (Katz, 1987).

Recently, researchers from other domains have begun to examine the benefits that version control can offer their application areas. Several researchers have used existing work on software engineering version control systems as a starting point for their efforts. In many cases, researchers are finding that techniques used in existing version control systems can be customized to work in their environment (Zdonik, 1986). For example, existing research in software engineering has been extended to the design of specialized version control systems for areas such as computer aided design (Chou, 1986; Katz, 1987).

Hypertext is a discipline in which version control is critically important (Halasz, 1988). Hypertext systems are already being used for evolutionary development tasks in which users request version

support such as authoring and idea processing. Future hypertext systems will likely be employed to support tasks such as software configuration management and collaborative work which have even more extensive version control requirements. The absence of version control functionality will limit the potential applications of hypertext systems. It is therefore critically important that existing version control material be reviewed, built upon, and expanded so that effective version control mechanisms for hypertext systems can be developed.

This paper examines the major issues of version control in hypertext systems. The first section is devoted to general background material on version control systems. It covers several concepts and features that are common to all version control systems. The information in this section serves as a basis for the material covered in the following sections. The second section of the paper specifically addresses version control in hypertext systems. This is followed in the next section by an examination of previous work done in hypertext versioning. In the final section, the major issues of versioning in hypertext systems are identified and discussed and practical solutions suggested where possible.

2. GENERAL VERSION CONTROL CONCEPTS

Version control systems assist in the evolution of objects. In the development of an object many refinements and changes are usually involved. This can result in several intermediate revisions of the object. In some application areas, preserving intermediate revisions is of utmost importance. When the number of revisions is large, this can be a difficult task. Version control systems support this process by facilitating the storing, naming, retrieving, and management of relationships between successive revisions.

Different environments vary in the degree of version control they provide. An example of minimal version control functionality is represented by a simple scheme in which a text editor automatically saves a backup copy of the file currently being edited. In this case, access is provided only to the immediately preceding revision. Features characteristic of a more elaborate version control environment include: easy access to all revisions, substantial reductions in storage requirements, and support for parallel lines of development.

This section examines version control in general. The majority of research efforts to date in this area have been in the development of software configuration management systems. Consequently, most of the work referenced here is from that domain. However, the concepts themselves are general and not specific to software development. This section begins with a brief definition and discussion of revisions and variants. This has been included to avoid the confusion that can result from inconsistent usage of these terms. Next we present discussions of the important aspects of version control systems including the data model, storage strategy, support for collaboration, and configuration management.

2.1 Revisions and Variants

Within the terminology of version control, a *revision* refers to an object that is a refinement of an existing one. When an existing object is edited and changed, a revision results. It represents a discrete point along the path of an object's evolution. A simple example of a revision is the text file produced when an existing source code module is edited and debugged. The resulting file is a refinement of its predecessor and represents a step in its evolution.

The term *variant* refers to a related but different concept. Two objects are variants of one another if they are on parallel paths of development. Variants usually represent alternative designs or implementations and often are interchangeable at some level. Unlike revisions, one does not evolve directly from the other, but along side one another on parallel paths.

In order for two objects to be variants of each other they must be indistinguishable at some level of abstraction [Tichy, 1988]. An example of a variant is a software module that has the same interface and function as another, but is implemented with a different underlying algorithm. The two modules are indistinguishable at the interface and functional levels, but have different implementations.

Some types of variants are indefinitely maintained on parallel paths of development. In some cases variants only exist for a limited amount of time and may eventually be merged. Consider the situation that results when an error is found in a production software module. It is possible that development work has created new revisions of the module since its production release. The error should be corrected, but it is desirable to do so without the influence of development that has occurred since the production release. This forces the maintainer to create a variant of the module as it existed at release time in order to fix the error. However, the changes required to create this variant and fix the error will likely be merged into the next production release [Leblang, 1984].

2.2 Data Model

The data model of a version control system describes the data entities allowed in the system and the relationships which can exist between them. It determines the paths of evolution allowed for objects within the system. Some of the older version control systems had rigid data models which only allowed a strict linear sequence of revisions to exist. Such limited data models can restrict a system's potential applications. Recent version control systems have more flexible data models that provide support for variants as well as revisions, and sometimes allow other nonlinear paths of development to exist.

2.2.1 Tree Data Model

The most popular data model is the *tree*. Variations of it have been used by many version control systems such as Shape (Mahler, 1988), Adele (Belkhatir, 1986), the Revision Control System (Tichy, 1985), and the Source Code Control System (Glasser, 1978). The tree data model supports both revisions and variants. As illustrated in Figure 1, a series of revisions is represented by a linear sequence of nodes in the tree, like the one extending from node 1.0 to 1.4. The creation of a variant is represented by a branch point in the tree, as occurs at node 1.1. The parallel development of the variant corresponds to the series of revisions represented by the linear sequence of nodes beginning at node 2.1.

Many version control systems have used the tree data model. Systems such as the Source Code Control System (SCCS) have successfully used this model to support software engineering. This model supports the process of creating and editing source code modules as they evolve through the traditional software life cycle. It also allows variants to be created when alternative designs or implementations are needed.

Much of the popularity of the tree data model can be attributed to the adequate support for traditional software engineering methods. However, as version control systems are designed and used in new areas such as hypertext systems or computer aided design, new data models will likely be needed. These new application areas are likely to bring requirements which will not necessarily map well to the tree data model.

2.2.2 Layered Network Data Model

An interesting alternative to the tree data model is the *layered network data model*. This data model was used in the Personal Information Environment System [Goldstein, 1984]. In this model objects correspond to nodes in a network. Arcs between nodes represent relationships between objects. A set of related arcs can be grouped together to form a layer and related layers can be grouped together to form contexts.

Figure 2 illustrates a software module and its associated documentation as represented by a layered network. The node corresponding to module A has arcs extending from it, linking it to related

components in the network. As the figure illustrates, release 1.1 of module A uses the quicksort algorithm and its documentation consists of three text nodes. These objects and the links between them comprise layer A, the single layer in this layered network.

Figure 3 illustrates a context that contains two layers, A and B. It is similar to the context represented in Figure 2, but includes a few changes. Layer A serves as the base of this context. Initially all of the nodes and arcs from layer A are included and then layer B is applied. It changes the context by adding new nodes and arcs and modifying existing ones. In this example, normal lines represent arcs from layer A which were not modified by layer B. Dashed lines represent arcs from a node in layer A to a node which was added by layer B. Dotted lines represent arcs from layer A that are dominated or deleted by layer B. As Figure 3 illustrates, the addition of layer B has changed the sort routine from a quicksort to a heapsort. It has also changed the associated documentation. This figure represents the network that corresponds to a context which contains layers A and B.

The representational capabilities of the network data model encompass those of the tree data model. Consider the tree illustrated in Figure 4. It could be represented by a layered network which contains a layer for each node in the tree. The layer corresponding to a particular node from the tree contains the changes that caused the creation of the node. A specific revision can then be represented by building a context in the layered network that consists of the base layer along with all layers that correspond to nodes on a path from the root to the desired revision. For example, Figure 5 illustrates a context which corresponds to revision 1.2.0 from the tree in Figure 4. It contains the base layer along with the layers representing nodes 1.1 and 1.2.0.

The previous example, while illustrating how a tree can be represented in the layered network model, also illustrates one of the tree model's major constraints – its confinement to a temporal sequence of evolution. Suppose the nodes in the tree of figure 4 represent independent self contained units of evolution of an object. Even though the nodes are independent, they would still be connected in a linear sequence under the tree model. This sequence represents their temporal relationship. In some applications temporal relationships are not important and their mandatory inclusion can become unnecessarily restrictive (Katz, 1990).

In the layered network model the representation of temporal relationships is not explicitly required. Independent units of evolution can be represented as individual layers each capable of being included in arbitrary contexts. As described in the previous example, contexts are capable of capturing temporal relationships when necessary. However, the layered network model allows temporal information to be disregarded when it is not important. This provides the layered network model with considerable more flexibility than the tree model. The layered network model deserves consideration as version control systems are designed for new application areas that require more flexibility and expressiveness.

2.3 Storage Mechanism

One of the major advantages offered by version control systems is a reduction in the amount of storage required to allow flexible access to previous revisions of objects. This is achieved through the use of efficient storage management techniques. To provide such flexible access within a conventional file system would require a copy of each revision to be saved in its entirety. This would clearly be undesirable since it would require excessive amounts of storage. It would also require users to meaningfully name objects in a way that identifies their lineage.

Almost all version control systems use a delta storage mechanism to achieve a reduction in storage requirements. This storage method basically involves storing only the differences between successive revisions. In practice, some delta storage schemes have performed quite well. For example, the one used in DSEE is capable of storing 50-100 revisions of a text object in the same amount of space required for two complete copies of the same object (Leblang, 1987). The major dimension along which delta storage mechanisms vary is the way deltas are actually stored. The two main strategies are separate delta files and conditional compilation.

2.3.1 Separate Delta Files

When *delta files* are used, the information needed to transform one revision into another is stored in separate files, one corresponding to each revision. These changes are usually recorded in the form of line based edit commands. Each edit performed to produce a new revision is represented as the

deletion of a line from the original revision, and the addition of a new line containing the changed text.

Systems which use delta files are categorized as either forward delta systems or backward delta systems. Forward delta systems keep the original revision of an object intact. They save within their delta files the changes needed to produce later revisions from previous ones. In contrast, backward delta systems keep the latest revision of an object intact. They save the changes needed to produce earlier revisions from later ones.

The most appropriate strategy to use, either forward or backward delta files, depends on the application. An analysis of typical access patterns allows the optimal strategy to be identified. When older revisions are accessed often, a forward delta file strategy is appropriate. A backward delta file strategy works best when recent revisions are accessed most frequently. For example, within the realm of software development backward delta files are preferable since access is frequently required to latest revisions. Many of the systems developed for software version control such as RCS (Tichy, 1985) use a backward delta storage mechanism. According to Tichy, version control systems for software development should impose as little time delay as possible on programmers and therefore backward delta schemes are preferable. A backward delta scheme also allows new revisions to be added more rapidly since it only requires the calculation of one delta file (Tichy, 1985).

2.3.2 Conditional Compilation

In contrast to delta file storage strategies, *conditional compilation* schemes store all of the information for each revision of an object in the same file. Reconstruction of a revision is achieved by processing the file and applying only the deltas relevant to the desired revision. Most conditional compilation systems store delta information in the form of line based edit commands similar to those used by delta file systems.

One of the main differences among the various conditional compilation systems is the algorithm that is used to reconstruct revisions from the data file. In order to avoid intolerable access times as the number of revisions increases, it is important that an efficient algorithm be used for the regeneration process. The Source Code Control System (SCCS) uses a parallel delta application

method which allows access to any revision to be achieved in an essentially equal amount of time (Rochkind, 1975). However, even with an efficient single pass algorithm the time required for regeneration is bound to increase as the number of revisions grows large, thus causing slower access times than are possible with schemes using separate delta files storing backward deltas (Tichy, 1988).

A relatively new extension to systems using conditional compilation is the ability to edit multiple revisions simultaneously. Systems such as P-EDIT provide this capability (Sarnak, 1988). Multiversion editing can be very convenient when it is necessary to make the same changes to several revisions of an object. For example, suppose an error was detected in a particular block of a program module. If the block had not changed in several revisions, multiversion editing would allow all the revisions to be edited and corrected simultaneously. Without multiversion editing, each revision of the module would have to be corrected individually.

2.4 Support for Collaboration

Version control systems also provide facilities to assist collaborative efforts. An important facility provided by many systems is the management of simultaneous update attempts. Version control systems used in multiuser collaborative environments must have policies to handle attempts made to simultaneously update the same object.

Many existing version control systems use a simple check-out/check-in process to handle concurrent update attempts. The process of changing or updating an object begins by locking it through a check-out process. This prevents anyone from updating the object except the user who performed the check-out. Changes can then be made without concern for generating conflicting updates. The update process can usually span an arbitrary period of time, possibly involving several edit sessions. Once the changes have been made, the object is checked back in. This creates a new revision for the object and unlocks it allowing others update access. Requiring an explicit check-in operation allows the user to control the time when changes become visible to others. It also allows control over the frequency with which revisions are created.

Shape is an example of a software version control system that uses the check-out/check-in process to handle concurrent update attempts (Mahler, 1988). To change a program module a user must first

reserve it from the project database. The module can then be edited and changed as required. When the changes are complete, the module is submitted back to the project database. After being reviewed and accepted, the new revision reflecting the updates is made public.

Most systems which use the check-out/check-in policy allow read access to locked objects. Technically, references to locked objects are resolved to the latest checked-in revision of that object. This protects users from any changes that might currently be in progress. Some systems also allow write access to locked objects. After clearly informing a user that an object has already been locked for updates such systems allow the user write access in order to accommodate special situations. For example, if an error is detected in a program module that was currently locked by a development team member, a maintenance programmer could perform a check-out to correct the error. The development team could then be informed of the changes that were made and could correct their revision when convenient.

In systems which allow read/write access to locked objects, it is important to clearly define in terms of the data model exactly what happens when a locked object is updated. Most systems create a variant at the point where a write conflict might occur. This splits development into two distinct lines preventing the changes made in one revision from overwriting those made in the other. It is very important for these systems to provide support for merging objects. This facilitates situations such as the one described above, when it is desirable to merge changes that caused the creation of a variant back into the main line of development.

2.5 Configuration Management

While version control addresses the issues pertaining to the development and management of individual objects within a system, *configuration management* addresses the issues involved in identifying and maintaining relationships between those objects. A configuration identifies both a set of objects within a system and the relationships between them. Examples of configurations include a set of software modules that form a software system or a collection of Postscript files which compose a document. Configuration managers provide facilities to identify the components of a configuration and the relationships between them.

Once defined, configurations usually require some sort of processing to be transformed into a usable entity. This is sometimes referred to as "building" the configuration. In the examples above, this corresponds to compiling the source code modules into object code or processing the Postscript files to produce a print file. Configuration managers usually automate this transformation process. For example, the configuration management tool Make can automatically compile an entire programming system given a description of its configuration (Feldman, 1988).

Configuration management systems use various methods for the specification of configurations. For example, the Make system uses a file called the Makefile to describe the configuration (Feldman, 1979). In GYPSY the user specifies a configuration template to describe the configuration (Cohen, 1988). Regardless of how it is specified, the configuration normally identifies all of the objects in the configuration, the relationships between those objects, and the steps required to build them. Since multiple revisions of objects might exist, it is important that the specification uniquely identify each of the objects in the configuration.

Some older configuration management tools only allow static specification of the objects in a configuration. Newer systems, particularly those which include some form of version control, allow the dynamic specification of objects in configurations. In such systems, attributes can be associated with objects. The user can specify a predicate over those attributes to identify an object within a configuration. Whenever the configuration is used or accessed, this predicate is evaluated and the reference is resolved to a specific object. For example, in the Shape configuration management tool, the "state" attribute is used to track a module's status. Within a configuration description, it allows specifications such as the "latest" revision of a module which has a status of "debugged" (Mahler, 1988).

Some configuration management systems facilitate configuration documentation by providing system defined variables which can be embedded into objects. When these variables are encountered during a configuration build, the appropriate values are substituted. This allows automatic documentation of certain aspects of systems that are produced from configurations. SCCS provides this capability by allowing the user to embed system defined variables for the date, time, etc. When a configuration is built, appropriate values are substituted thus documenting the generated system (Rochkind, 1975).

3. VERSION CONTROL IN HYPERMEDIA SYSTEMS

Similar to more traditional development environments, hypertext systems have very important version control requirements. Indeed, version control has been identified as one of the critical research areas in the hypertext field (Halasz, 1988). Certain of these version control requirements are due to new capabilities which hypertext systems provide, such as a more dynamic environment for defining structure over data. Others are caused by the tasks for which hypertext systems are being employed. For example, hypertext systems are increasingly being used to provide richer environments for tasks such as writing, software engineering, and other kinds of evolutionary development. Version control has always been important for such development tasks and hypertext systems inherit their version control requirements.

In spite of its importance to hypertext systems, version control has thus far been neglected by most researchers. Current hypertext systems have little or no facilities for version control. This is due to the relative newness of the field. However, it is critically important that the area of version control be addressed in hypertext systems. The lack of version control facilities restricts the potential applications for which a hypertext system can be used. For example, a hypertext system with little or no version control would not serve well as an environment for software development where this functionality is absolutely critical.

This section examines version control as it specifically relates to hypertext. We begin by identifying the version control requirements of hypertext systems. This is followed by a discussion of previous work that has been done in this area. Next is a discussion of the major issues of versioning in hypertext. Some of these issues are similar to those identified in the background section while others are specific to hypertext. Where possible, references are given to illustrate how existing systems have dealt with these issues. When references are not available, strategies which can be used to cope with the issues are suggested.

3.1 Versioning Requirements of Hypertext Systems

The requirements of versioning in hypertext systems can be categorized into two main areas — versioning *data* and versioning *structure*. Versioning data refers to the process of managing the

evolution or development of a data object. This is the type of functionality traditionally associated with version control systems and was described in the background section of this paper. Most version control systems that have been developed to date are primarily geared towards versioning data. Most of the version requirements that hypertext systems inherit from other application areas involve versioning data.

In addition to versioning data, hypertext systems must also provide support for versioning structure. Hypertext provides a very rich environment for defining and manipulating structure over data. This allows the structure to be more dynamic and subject to change than was possible with previous systems. This increases the importance of version support for structure. In more traditional applications, even those that provide some sort of version control, the ability to easily define and manipulate structure over data either does not exist or is much less dynamic, thus not requiring version support. The requirement to version structure is one of the main challenges which hypertext brings to the version control area.

Software configuration management systems provide a useful analogy which helps distinguish the concepts of versioning data and versioning structure. Versioning data is analogous to providing flexible access to arbitrary revisions of a software module. Almost all software configuration management tools provide at least minimal support for versioning data. Some provide quite elaborate support for this task.

The software configuration management analogue to versioning structure is in providing version support for configurations, since they represent the structure of a software system. However, most configuration management tools provide little or no functionality in this area. While almost all software configuration management tools support the definition of a configuration, few provide version support in this area. The PAPICS system is an exception (Demleitner, 1988). It provides version support for not only the textual but also the structural changes that occur in a software system. According to Demleitner, the PAPICS system unites the versioning of data and structure into a common concept and thus far is the only system to accomplish this.

3.2 Previous Work

As we stated earlier, version control has been largely neglected by hypertext system developers and most current hypertext systems have little or no facilities for version control. The two notable exceptions are Neptune and Intermedia. Neither of these systems offer an integrated system for versioning data and structure. However, Neptune has made progress in the area of versioning data in hypertext systems, and Intermedia provides some of the basic tools necessary for versioning structure in hypertext systems.

Neptune provides support for versioning hypertext data. The hypertext abstract machine (HAM) upon which Neptune is built provides complete version histories for each of the nodes in a hypertext. The HAM provides much of the functionality which is normally associated with versioning data such as efficient storage mechanisms and flexible access to previous revisions. Earlier releases of Neptune were limited in that support was only provided for a linear thread of development. Trees and other more flexible data models were not supported (Delisle, 1986). This has since been addressed by an extension to Neptune which allows multiple contexts to exist. A context represents a private workspace for making changes to a set of nodes and links. Once changes made within a private context are complete they can later be merged into existing contexts allowing them to become public (Delisle, 1987).

Intermedia provides part of the basic functionality needed for versioning structure. It allows separate webs to exist over a common set of hypertext nodes (Yankelovich, 1988). Webs are collections of links which provide part of the basic functionality needed to support versioning structure, namely the ability to group links. However, the intention in Intermedia was to allow multiple users to have their own private view of the structure over a set of nodes, not to support an evolving view of the structure. Webs are not treated as evolving objects and version control is not provided for them.

3.3 Issues of Versioning in Hypertext Systems

This section examines several of the major issues of versioning in hypertext systems. The discussion is divided into two subsections. The first covers issues related to versioning data. Most of the issues of versioning data in hypertext systems are similar to those faced by versioning systems in general. Methods of coping with these issues can be identified by examining previous work and identifying solutions which perform optimally for the specific requirements of hypertext systems. The second subsection examines issues pertaining to the versioning of structure. Most of these issues are new and have not been encountered by previous versioning systems.

3.3.1 Versioning Data

The data to be versioned in hypertext systems is contained in the nodes. Nodes in a hypertext often contain ascii text representing text or graphics. Therefore, much of the existing research on version control can apply directly to versioning data in hypertext. Similar to existing version control systems, two of the major issues of versioning data in hypertext are to determine what data model and storage management techniques are appropriate. This requires an analysis of the evolution paths and access requirements of hypertext nodes. Once the patterns of evolution and access have been determined, existing research can be consulted and the optimal strategies for hypertext can be identified.

3.3.1.1 Data model

Most current hypertext systems which support any type of version control are based on a simple versioning data model. These systems only support a strict linear sequence of evolution. This accurately reflects the lack of activity and need for further work in this area. Delisle, when commenting on the versioning data model in Neptune, noted that no existing hypertext systems have provisions for multiple version threads [Delisle, 1986]. Expanding the versioning data model supported by hypertext systems is important since a limited versioning data model may restrict a hypertext system's potential applications. Many types of development work require more functionality than that offered by a strict linear path of evolution.

The versioning data model selected for a hypertext system must be flexible enough to meet all of the versioning needs of its intended uses. At a minimum, hypertext systems should support the tree versioning data model. This will provide virtually all of the evolution paths allowed by the version control component of existing systems. Tasks such as writing and software development would not be any more restricted by a hypertext system with this type of versioning data model since it provides all of the versioning functionality found in systems currently used for those purposes.

Given the capabilities of hypertext, more elaborate data models are needed and should be explored. The layered network data model is one which should be considered. A network-based data model offers several advantages over the more traditional tree-based data model. As discussed previously, it eliminates the temporal evolution constraint with which systems based on other data models must contend. Layers can be created which contain any desired set of nodes or changes regardless of the order in which they were created. This could be an important factor in hypertext systems. It would allow the creation of customized layers that represent multiple simultaneous views of the data in a hypertext network with less duplication than required by other models. In general, the flexibility of this model will more closely accommodate the types of development likely to occur in hypertext systems.

3.3.1.2 Storage Management

Version control strategies for hypertext systems must utilize an effective storage mechanism. As with any system providing version control, a hypertext system might be required to handle a large number of revisions. This is especially true for those systems used in evolutionary development environments. The storage mechanism must accommodate this requirement by providing a proper balance between the often conflicting goals of minimal storage usage and rapid access time. While it is always desirable to minimize space requirements, this should not be done at the expense of causing prohibitive access times.

The two major modes of operation in hypertext systems, authoring and browsing, each have impacts on the selection of a storage management mechanism. The access requirements of both activities must be examined in order to determine a suitable storage strategy. In authoring mode the user is likely engaged in some development activity. This normally involves the editing of latest revision

nodes causing new revisions to be created. Also, access to older intermediate revisions may occasionally be required. These tasks require rapid access to the most recent revisions, the ability to quickly add new revisions, and flexible access to older revisions.

In browsing mode, no new revisions will be created. Instead, users will be traversing an existing set of nodes and will likely be less tolerant of long access time delays. Although many nodes which are visited while browsing will likely be latest revision ones, this is not necessarily always the case. Links to older revision nodes may sometimes be encountered. Therefore, browsing might require rapid access to any arbitrary revision of a node.

To be effective, a storage management strategy for versioning in hypertext must optimally address each of these requirements. The backward delta storage technique accommodates almost all of these types of access. It maintains delta files representing each revision along the path of an object's evolution, so flexible access to previous revisions is possible. Adding a new revision only requires computing a single delta file. This process can also be performed quickly. Since the latest revisions are stored intact, rapid access to them is also possible.

The last requirement identified above, rapid access to arbitrary intermediate revisions for browsing purposes, can be addressed by slightly augmenting the backward delta strategy. The augmentation involves allowing intermediate revisions of an object to be stored intact, thus providing faster access to selected revisions. Nodes which are not the latest revision, but that are traversed often while browsing, can be designated to be stored intact. This will provide faster access to such nodes than is possible by reconstruction from delta files.

None of the other storage strategies are likely to work well for hypertext. Forward delta strategies only provide rapid access to the oldest revisions of objects. Conditional compilation techniques require processing files which grow larger as the number of revisions increases. Neither of these approaches would provide the rapid access required while browsing; and backward delta strategies provide better support for authoring.

Another storage management consideration for hypertext systems is the types of data that are to be stored. In addition to textual data, several other types of data are possible in hypertext systems. These include graphics, video, and audio. This list will likely expand as new technologies are developed making new types of media possible in hypertext systems.

As hypertext systems are expanded to include these new types of media, the version control requirements involved must be addressed. In general, the storage format for such data will not be text based. Instead, it will likely be some binary format specific to a particular type of media. Versioning such nontextual data types presents several challenges. One of the most critical is the development of effective delta calculation mechanisms. The algorithms used in most current version control systems are specifically designed for textual data and are not applicable to arbitrary data types. Appropriate delta calculation algorithms must be found to keep storage requirements under control (Scacchi, 1988).

3.3.2 Versioning Structure

Hypertext provides a rich environment for manipulating structure over data. This allows the structure to become very dynamic. As the frequency with which the structure changes is increased, version control requirements begin to emerge. Although versioning structure will provide much needed functionality to hypertext systems, it will also impose new requirements that these systems must address. This section discusses several of the issues involved in versioning structure in hypertext systems. It begins with a few basic definitions. Next, the implications of versioning upon anchor specification are examined. The section is concluded by a discussion of versioning links.

3.3.2.1 Definitions

It is useful at this point to clarify a few terms which will be used often in this section, specifically anchor, link, and context. An anchor designates a node or set of nodes that serve as the source or destination of a link. A specific location within each of the member nodes of an anchor is usually required as part of the anchor definition. A link is a connection among anchors. Anchors are the only objects to which links can be attached (Leggett, 1988). Figure 6 illustrates two anchors with a link between them. Anchor 1 has two nodes associated with it. Anchor 2 has only a single node associated with it. In this case the link is a connection between anchor 1 and anchor 2. The term context will be used in this discussion to refer to a group of one or more links. Contexts provide a convenient mechanism for specifying subsets of links.

3.3.2.2 Implementation of Anchors

Anchors define the endpoints of links. When defining the endpoint of a link, two levels must be considered. First is the inter-node level. This is the level which identifies the nodes that compose an anchor. In Figure 6, the inter-node specification for anchor 1 designates nodes A and B as its component nodes. The second level of anchor specification is the intra-node level. At this level, locations within nodes are designated to be anchor positions. In Figure 6, intra-node specifications are indicated by the arrow heads pointing to specific locations within the boxes that represent nodes. Specification at each of these levels is necessary to completely define an anchor. This subsection addresses the impact that versioning has on the identification and maintenance of anchors at both of these levels.

3.3.2.2.1 Inter-Node Level

At the inter-node level, the component nodes of an anchor are identified. In hypertext systems which provide version support, this process involves more than simply identifying a set of nodes that make up the anchor. For each node, several revisions might exist. It is possible for links to be made to arbitrary revisions of nodes. Identification of a node will require specifying the revision as well as the node.

It is the responsibility of the anchor to determine which revisions are part of a link endpoint. Flexibility is important at this level. For example, in some situations it might be appropriate that an anchor always reference the latest revision of a node. In other situations it might be important for an anchor to permanently refer to a specific revision of a node, regardless of any newer revisions. In practice several other situations which require flexible inter-node specification are possible.

A mechanism for providing flexibility at the inter-node level is to allow nodes to have attributes, and allow inter-node specification to be done in terms of these attributes. This will accommodate each of the cases identified above as well as many others. For example, when an anchor must point to the latest revision of a node, the reference to it could include a specification for the revision with the largest (implying most recent) date attribute. A specific revision of a node could be referenced by supplying the specific value of its date attribute or a value for some other attribute which uniquely identifies it among all revisions of the node.

This concept is similar to software configuration management systems such as the Shape system that allow the dynamic specification of configurations. The Neptune hypertext system provides some of this capability. It allows nodes to have any number of attributes. It also allows links to point to either the latest revision of a node or a specific revision of a node. However, the reference to a node cannot be generally specified in terms of the attributes (Delisle, 1986).

3.3.2.2 Intra-Node Level

At the intra-node level, the specific location within a node referenced by an anchor is identified. Versioning introduces certain issues at this level which must be addressed. One of the most important is the issue of how locations within a node are identified. References can be relative or specific to an object within a node. When references are tied to a specific object within a node inconsistencies can arise. For example, consider an anchor which when created is defined to point to a specific object within the latest revision of a node. If the object were deleted from the node, the next time this anchor is encountered its intra-node reference would be undefined.

For text nodes, this situation can be avoided by implementing intra-node references as character offsets relative to the start of the node. This prevents references from becoming undefined. However, it is possible for the semantics of an intra-node reference to be lost if the text originally pointed to by the anchor is edited or moved. The Neptune system identifies intra-node references in this way.

3.3.2.3 Versioning Links

One of the major challenges of versioning links will be to provide an effective method of managing the large number of links that will result when structure is versioned. When multiple revisions of each link in a hypertext system are possible, the overall number of links will be greatly increased. This will impact operations such as initiating an authoring or browsing session. Since multiple revisions of each link will be possible, a mechanism must be provided to specify which revision of each link should be used at a particular time. A simplistic approach is to automatically use a default revision of each link, such as the latest one. Another simple strategy is to require the user to specify, on an individual link basis, which revision should be used. Neither of these approaches is practical.

The former lacks flexibility while the latter would quickly become overwhelming to the user for even a moderate number of links.

A grouping mechanism for links is clearly needed. Contexts can be used for this purpose. Contexts allow links to be grouped together and treated as a unit. This will enable a collection of specific revisions of links to be grouped and referred to by name. This provides a flexible solution to the problem identified above. The use of contexts reduces the problem of link revision specification to a simple context selection.

Another important issue in the versioning of links is deciding what level of granularity is appropriate for versioning structure. The use of a grouping mechanism for links as described above makes several levels possible. For example, when contexts are used it is possible to version structure at the individual link level, the context level, or a combination of both. The most appropriate level for a specific system will be determined by its intended applications.

When structure is versioned at the individual link level, the link becomes the base unit of versioning. The lineage of each individual link is preserved so that access to previous revisions of links is possible. Figure 7 illustrates versioning at the individual link level. In this example, individual links from the overall set of links have been grouped into a context named Callisto that is represented by the dashed ellipse. The context contains a specific revision of each link in the link database. Figure 8 illustrates the effect of an edit session. Links 1 and 4 have been updated producing new revisions of each. The Callisto context has been altered to include the new revisions of both of these links.

Structure can also be versioned at the context level. Version support at this level provides access to previous revisions of sets of links. In contrast to the link level, versioning at this level allows access to previous revisions of the structure as a whole. Figure 9 provides an illustration of versioning structure at the context level. The figure illustrates a context named Ganymede overlaying a hypertext node. The context consists of three links. Figure 10 illustrates the Ganymede context after an edit session. A new link to node 20 has been added and the destination of an existing link has been changed. These changes resulted in the creation of a new revision of the context.

Both the individual link level and the context level are useful for versioning structure in certain situations. Versioning at the link level allows a record of subtle changes to be maintained while

versioning at the context level allows the development of the structure as a whole to be tracked. When used individually, however, neither strategy completely captures the evolution of structure in a hypertext.

Versioning strictly at the link level does not preserve context information; it is not possible to access previous revisions of contexts. While the lineage of individual links is preserved, the relationships between separate links is not. For example, in Figure 7 after the Callisto context has been changed to include a new revision of link 1 and link 4, access to the previous context is not possible. Access to previous revisions of individual links is allowed but not to previous revisions of the structure as a whole.

Versioning structure exclusively at the context level fails to capture the evolution of individual links. In such a strategy there is no concept of revision for an individual link; all of the links in the system exist as independent entities. As illustrated in Figure 9, once changes have been made to the Ganymede context resulting in a new revision, access is only possible to a previous revision of the entire context. It is not possible to access previous revisions of individual links.

In order to completely capture the evolution of structure, versioning must be done at both levels. This will allow access to previous revisions of individual links as well as contexts. A combination of the strategies discussed above can be used for this purpose. Version histories for individual links and contexts can be maintained just as described. However, the structure of contexts must be expanded to allow the denotation of specific revisions of links. Figure 11 illustrates versioning at both levels. As indicated in the figure, link references within contexts specify both a link identifier and a revision identifier; and the contexts themselves are also versioned. Whether or not the added overhead of versioning at both levels is necessary will be determined by the requirements of the intended application.

4. FUTURE RESEARCH

This paper has identified several basic issues which should be considered as version control functionality is added to hypertext systems. Many additional issues will likely be encountered. Since few existing hypertext systems provide support for version control, most of these issues have not yet been addressed. This situation leaves much open room for research in this area.

As described previously, many of the issues of versioning data in hypertext are similar to those faced by software configuration management systems, and existing research can be consulted to identify methods of addressing them. Other issues related to versioning data in hypertext will require new development or the extension of existing version control work to be effectively addressed. One such issue concerns the data model used for versioning data in hypertext systems. Even though the layered network model is certainly an improvement over previous ones, research should continue in this area to avoid imposing unnecessary constraints on hypertext systems simply because they are inherent in existing data models.

A significant amount of work is needed in the area of versioning structure in hypertext systems. This paper examined only those issues related to the basic low level building blocks of hypertext structure – anchors and links. Many issues remain that are related to higher level hypertext structures such as composites. The semantics of versioning such structures needs further research. Also, research is needed in determining appropriate levels for versioning structure. As explained earlier several levels are possible, but the appropriate levels for various applications need to be determined. Once suitable levels for versioning structure have been identified, additional issues are immediately introduced such as which data model and delta storage mechanisms are appropriate to use for versioning structural components. Work is also needed to determine the set of internode reference attributes that should be supported by hypertext systems, and the set of predicates that should be allowed over those attributes.

There are several issues at the system level related to versioning in hypertext which remain to be addressed. An important issue in this category is the partitioning of versioning functionality between the hypertext system itself and the backend database upon which it is built. As specialized hypertext databases (or hyperbases) are developed, it will become possible to assign more responsibility for version control to the backend. Research is needed to determine how much

Version Control in Hypertext Systems

version control responsibility beyond the simple storing and retrieving of objects should reside in the backend. A related question at this level is how to extend existing database systems to provide version support for hypertext. This issue should be examined for each of the various types of database systems that currently serve as hypertext system backends.

5. SUMMARY

The absence of version control in hypertext systems will become increasingly apparent as usage expands to include new areas where this functionality is essential. As research progresses on adding version control to hypertext systems, existing research in software configuration management systems should be examined. The majority of version control research to date is from this area, and many of the basic principles involved are applicable to version control in general. Specifically, research on the data model, storage component, support for collaboration, and configuration management of existing systems must be reviewed. This will serve as a good basis for researchers working on version control in hypermedia systems.

Although hypermedia researchers can benefit from existing work on version control, the unique environment that hypertext provides for manipulating structure over data will introduce new issues which have not been encountered by previous version control systems. The presence of multiple revisions of hypertext nodes creates several new issues related to the specification of anchors. When multiple revisions of links are possible, several additional issues are introduced. Effectively addressing issues relating to versioning structure constitutes the main challenge to researchers working on version control in hypertext.

Clearly, substantial research efforts will be required to thoroughly address the issues identified in this paper. Until the result of such research efforts become available, hypertext system developers can use a combination of existing research from the software configuration management area along with some of the suggestions from this paper to provide basic version control functionality. The following combination could be used to construct a basic version control component for a hypertext system:

- data model – layered network model
- storage mechanism – augmented backward delta strategy
- inter-node referencing – use attributed nodes with general predicates allowed over the attributes. Place revision selection functionality at the anchor level.
- intra-node referencing – character offset method
- structural versioning – support versioning at the link and context levels with a tree data model used for versioning structural components

Version Control in Hypertext Systems

A version control component based on these suggestions would provide effective basic versioning functionality. While this approach might not represent the optimal strategy for addressing each of the issues, it would provide substantially more version control functionality than found in any existing hypertext system.

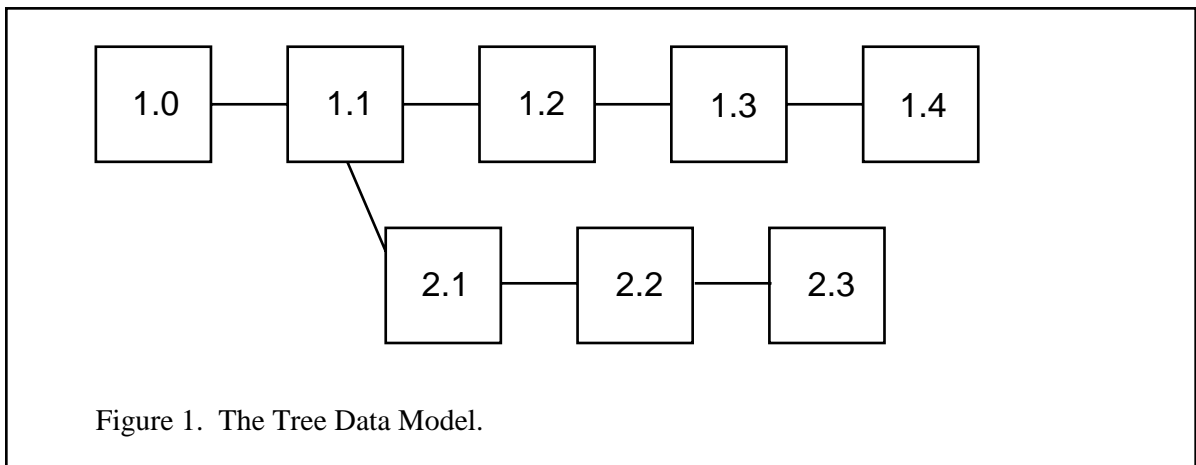
6. REFERENCES

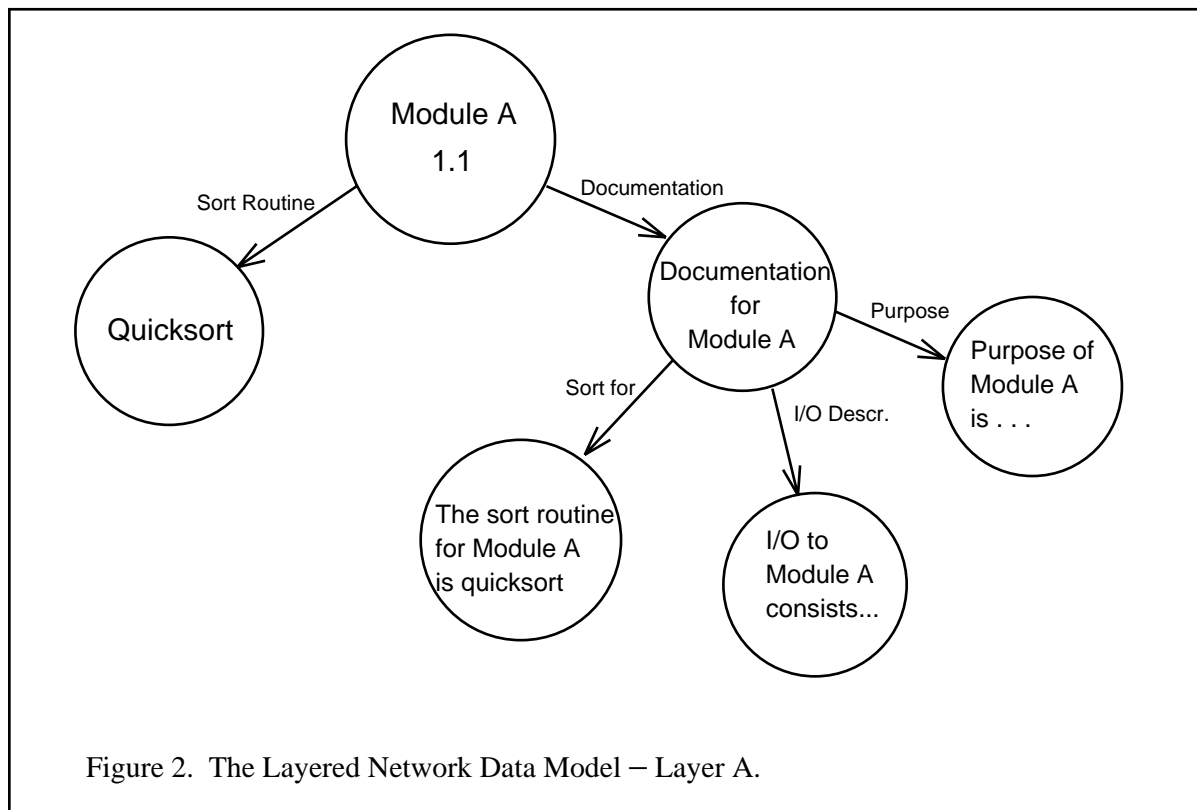
- Belkhatir, N., and Estublier, J. 1986. Protection and cooperation in a software engineering environment. *Proceedings: Advanced Programming Environments*, (Trondheim, Norway, June), *Lecture Notes in Computer Science*, 244, Springer-Verlag, pp. 221-229.
- Chou, Hong-Tai, and Kim, Won. 1986. A unifying framework for version control in a CAD environment. *Proceedings of the 12th International Conference on Very Large Data Bases*, (Kyoto, Japan, August). pp. 336-344.
- Cohen, Ellis S., Soni, D., Gluecker, R., Hasling, W., Schwanke, R., and Wagner, M. 1988. Version management in Gypsy. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (Boston, Mass., November), pp. 201-215.
- Delisle, Norman, and Schwartz, Mayer. 1986. Neptune: A hypertext system for CAD applications. *Proceedings of the SIGMOD '86 Conference*, (Washington, D.C., May), pp. 132-143.
- Delisle, N., and Schwartz, M. 1987. Contexts – A partitioning concept for hypertext. *ACM Trans. Off. Inf. Syst.*, 5, 2, (April), 168-186.
- Demleitner, Norbert. 1988. PAPICS: A practical approach to configuration management. *Proceedings of the International Workshop on Software Version and Configuration Control*, (Grassau, FRG, January), pp. 381-390.
- Feldman, Stuart I. 1979. Make – A program for maintaining computer programs. *Software -- Practice and Experience*, 9, (November), 255-265.
- Feldman, Stuart I. 1988. Evolution of Make. *Proceedings of the International Workshop on Software Version and Configuration Control*, (Grassau, FRG, January), pp. 413-416.
- Glasser, Alan L. 1978. The evolution of a source code control system. *Proceedings of the Software Quality and Assurance Workshop*, (San Diego, Calif., November), pp. 122-125.
- Goldstein, Ira P., and Bobrow, Daniel G. 1984. A layered approach to software design. In *Interactive Programming Environments*, D. R. Barstow, H. E. Shrobe, and E. Sandewall, Eds., McGraw-Hill, New York, pp. 387-413.
- Halasz, Frank G. 1988. Reflections on NoteCards: Seven issues for the next generation of hypermedia systems. *Commun. ACM*, 31, 7, (July), 836-852.

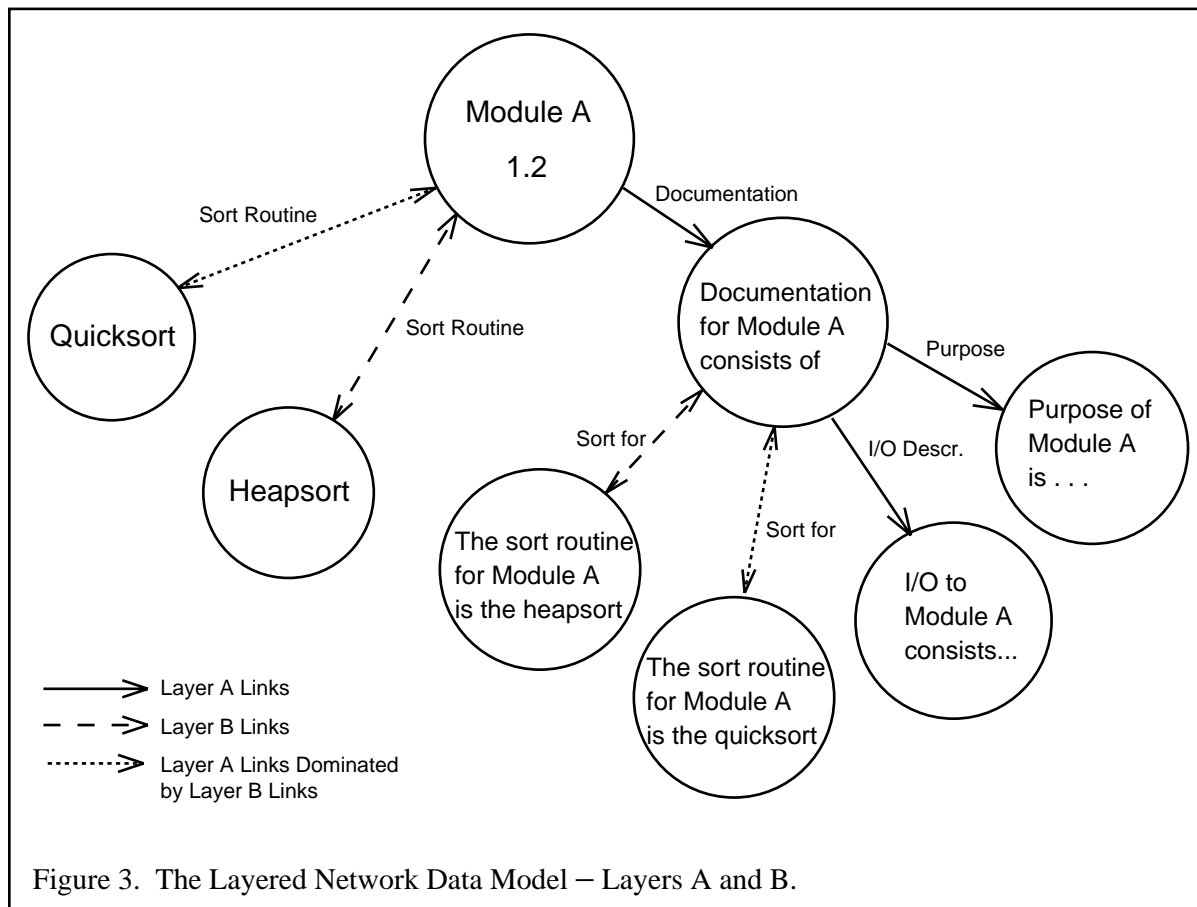
- Katz, Randy H., et al. 1987. Design version management. *IEEE Design & Test*, 4, 1, (February), 12-22.
- Katz, Randy H. 1990. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, 22, 4, (December), 375-408.
- Leblang, David B., and Chase, Robert P. 1984. Computer-aided software engineering in a distributed workstation environment. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (Pittsburgh, Penn., April), pp. 104-112.
- Leblang, David B., and Chase, Robert P. 1987. Parallel software configuration management in a network environment. *IEEE Software*, 4, 6, (November), 28-35.
- Leggett, J. J., Schnase, J. L., and Kacmar, C. J. 1988. Working definitions of hypertext. Department of Computer Science Technical Report No. TAMU 88-020, Texas A&M University, College Station, Texas.
- Mahler, Axel, and Lampen, Andreas. 1988. Shape – A software configuration management tool. *Proceedings of the International Workshop on Software Version and Configuration Control*, (Grassau, FRG, January), pp. 228-243.
- Rochkind, Marc J. 1975. The source code control system. *IEEE Trans. on Softw. Eng.*, SE-1, 4, (December), 364-370.
- Sarnak, N., Bernstein, R., and Kruskal, V. 1988. Creation and maintenance of multiple versions. *Proceedings of the International Workshop on Software Version and Configuration Control*, (Grassau, FRG, January), pp. 264-275.
- Scacchi, Walt. 1988. Summary of plenary discussion "CM for non-textual representation." *Proceedings of the International Workshop on Software Version and Configuration Control*, (Grassau, FRG, January), pp. 363-368.
- Tichy, Walter F. 1985. RCS – A system for version control. *Software -- Practice and Experience*, 15, 7, (July), 637-654.
- Tichy, Walter F. 1988. Tools for software configuration management. *Proceedings of the International Workshop on Software Version and Configuration Control*, (Grassau, FRG, January), pp. 1-20.

Yankelovich, N., Haan, B., Meyrowitz, N., and Drucker, S. 1988. Intermedia: The concept and the construction of a seamless information environment. *IEEE Computer*, 21, 1, (January), 81-96.

Zdonik, Stanley B. 1986. Version management in an object-oriented database. *Proceedings: Advanced Programming Environments*, (Trondheim, Norway, June), *Lecture Notes in Computer Science*, 244, Springer-Verlag, pp. 405-421.







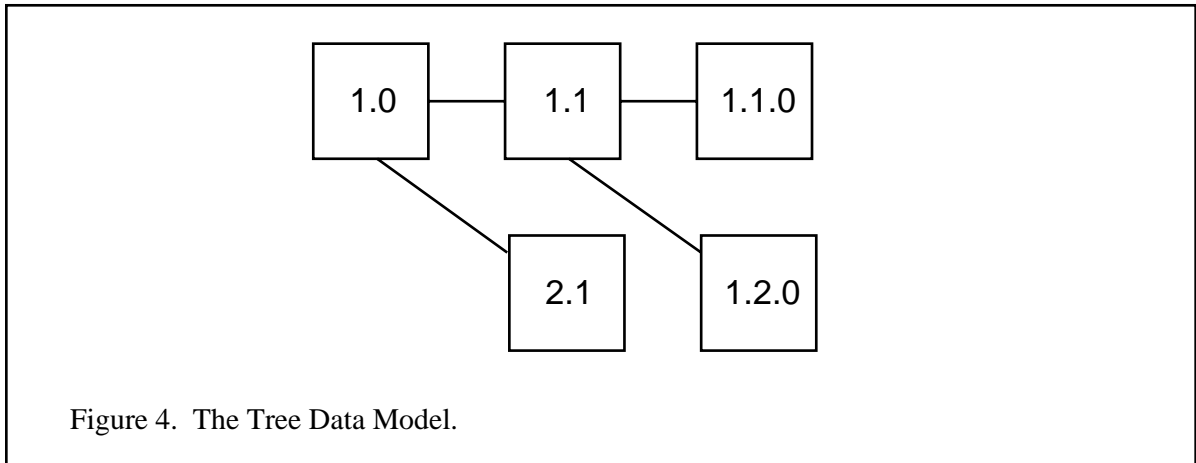
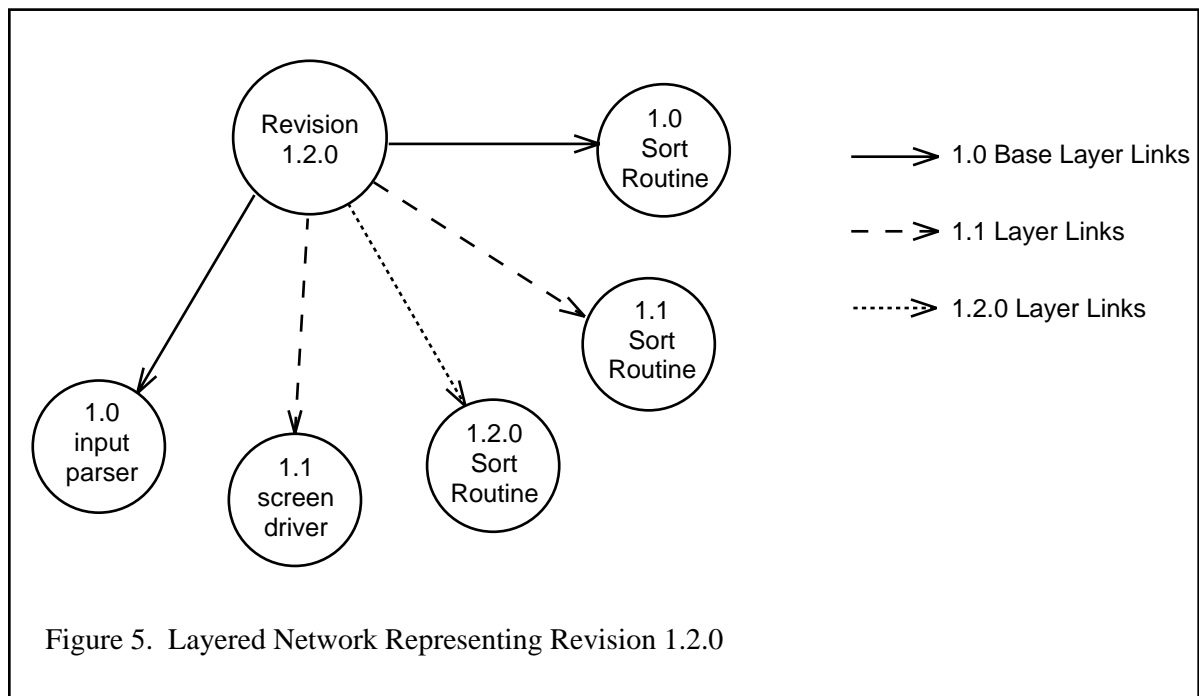
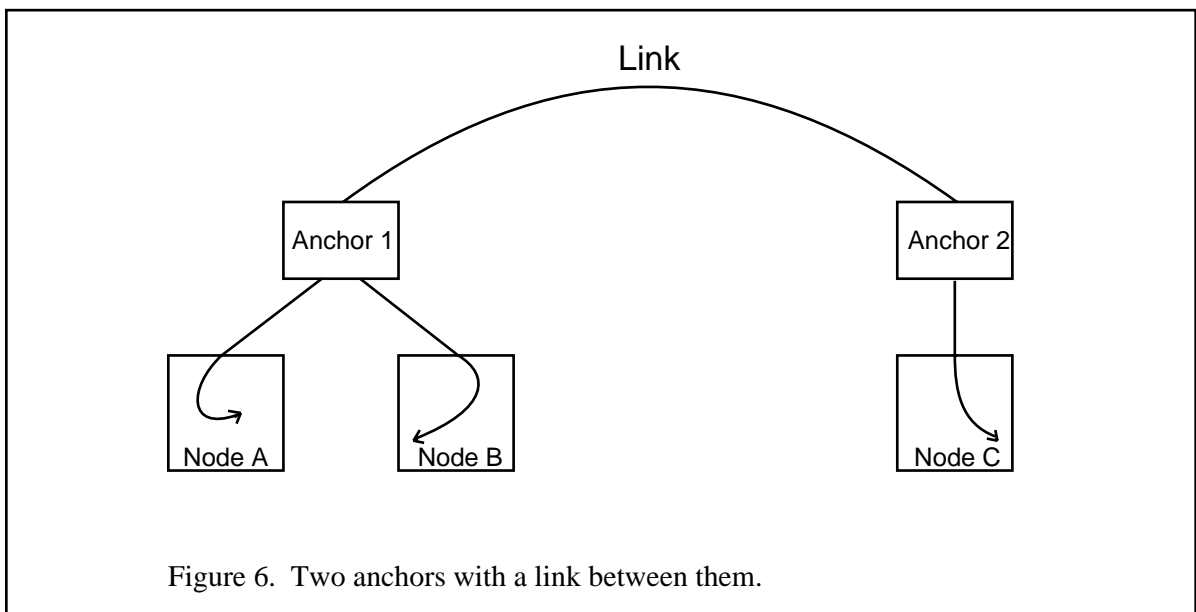


Figure 4. The Tree Data Model.





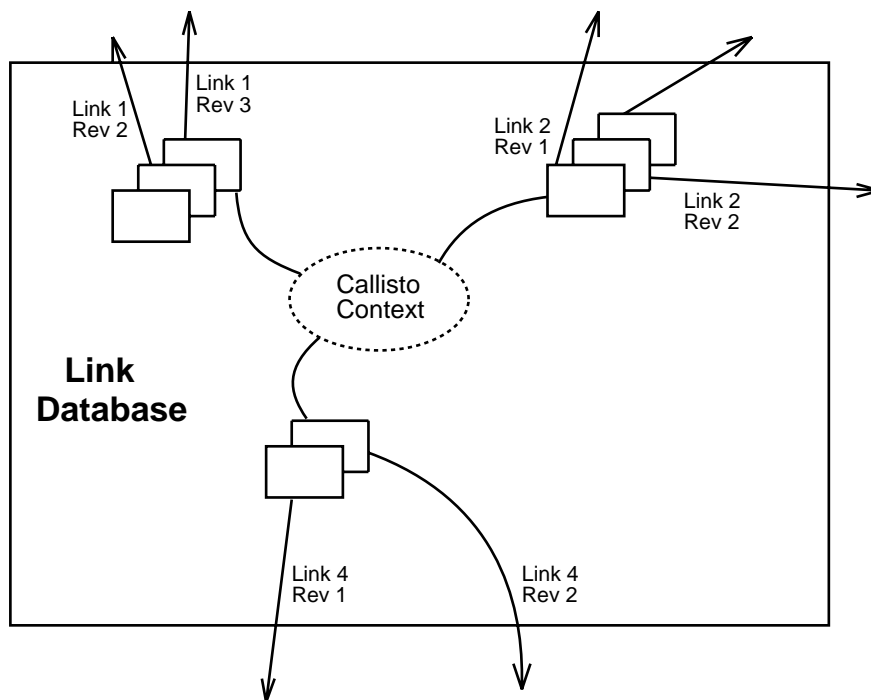


Figure 7. Versioning Structure at Link Level.

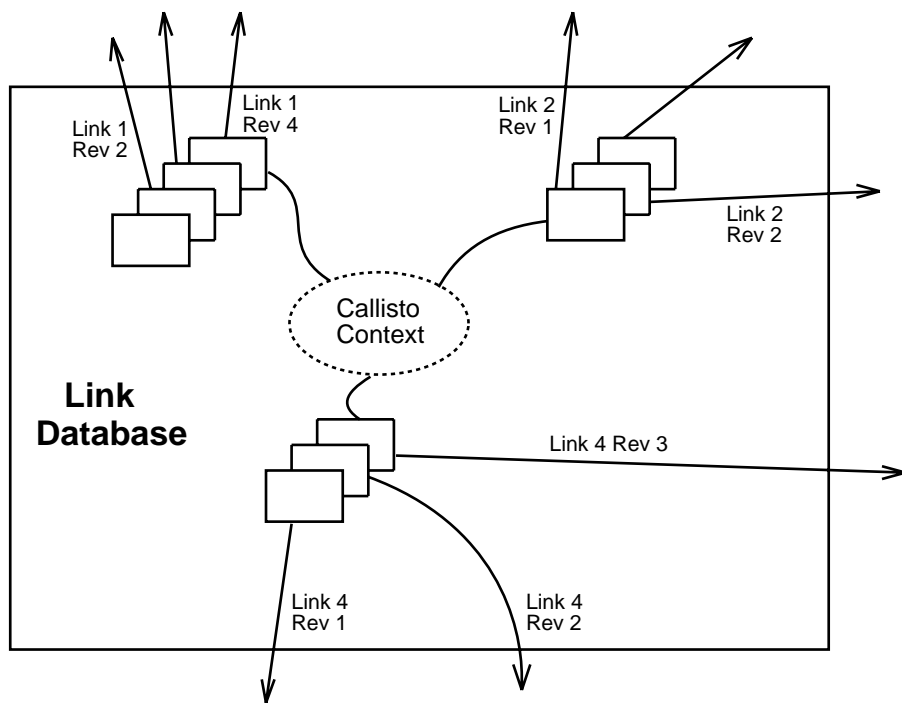
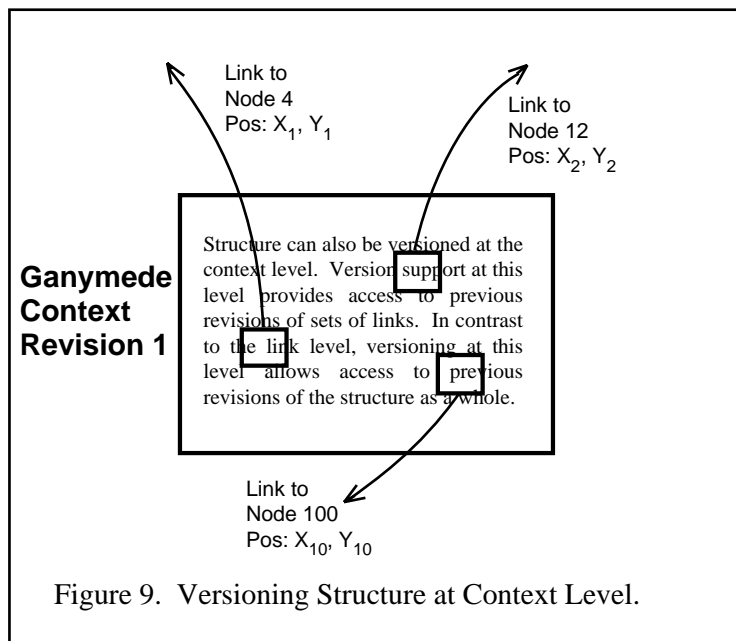
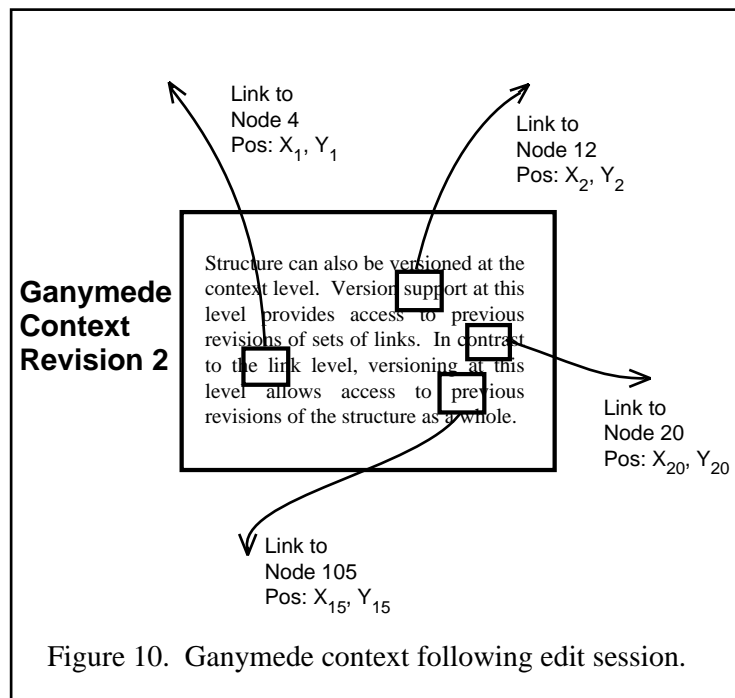


Figure 8. Callisto context following edit session





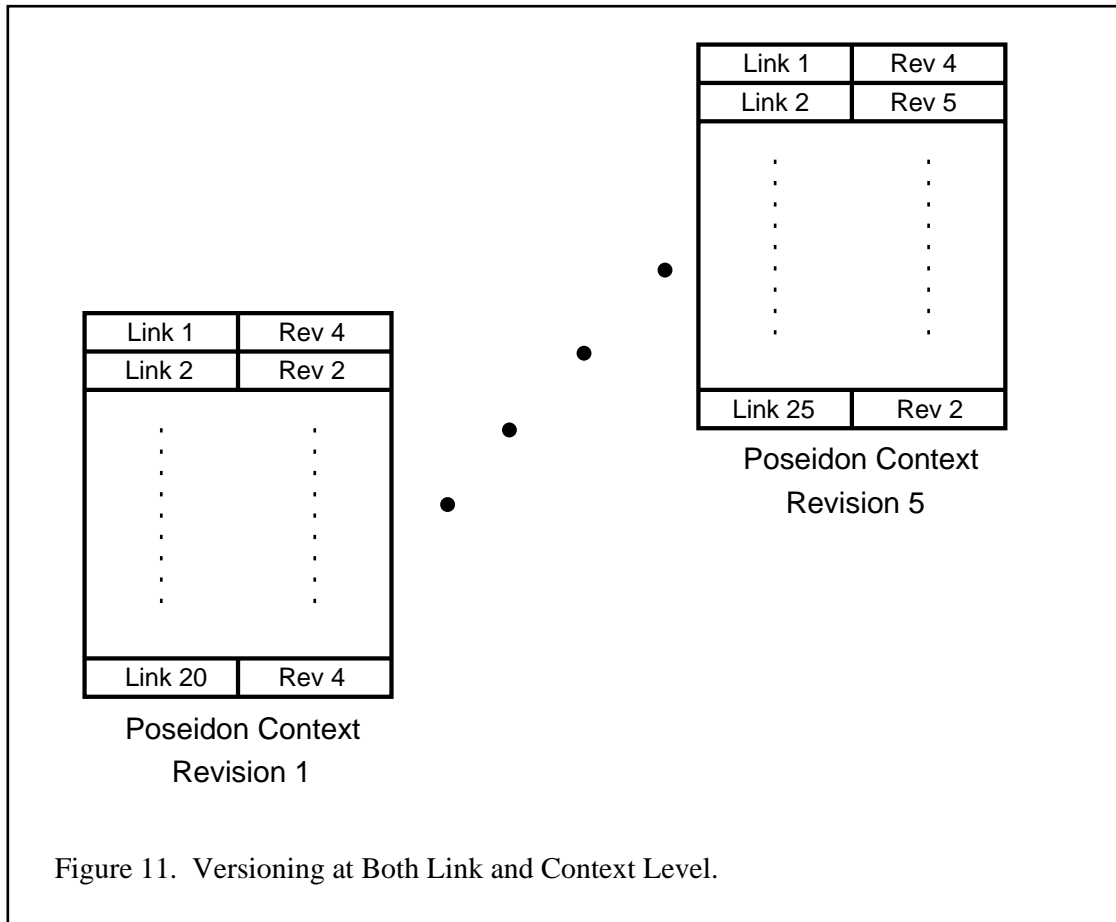


Figure 11. Versioning at Both Link and Context Level.