Versioning in Hypertext Systems

E. James Whitehead, Jr.

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425
ejw@ics.uci.edu

December 7, 1999

Table of Contents

1 Introduction

In the absence of hypertext, our data is tightly packaged. Documents, spreadsheets, presentations, source code, and databases, all exist wrapped up in their snug, independent files, each datum an island. Or are they? In 1945, Vanevar Bush made the critical observation that documents do not exist in isolation, but instead participate in a set of relationships with other documents [7]. Some relationships are explicit, as with a reference or footnote in an academic article, while others are implicit, reflecting, for example, some similarity (or difference) in content, style, location, history, etc. Furthermore, Bush made the leap that a machine, the Memex, should capture these relationships so a reader could quickly follow them, nimbly hopping from one document to the next.

Though the proposed Memex machine employed microfilm, it was obvious to future researchers that the computer's ability to flexibly manipulate text made it the ideal medium for document interlinking, creating texts that are more than just text: *hyper*texts. A computer can store large volumes of information compactly, provide searching over the information, and, when combined with a network, can import and export this information. When following a relationship, it can highlight the related text, often by amending the original document to underline regions that are part of relationships. Relationships in computer storage can themselves be analyzed, displayed separate to their documents, and link trails can be exchanged among interested users. The computer also makes it easy to change documents and the network of relationships between them, simultaneously a blessing and a curse.

Hypertext captures the implicit and explicit relationships between individual files, making them real data objects that can be acted upon by the computer. But, this act of capturing the relationships breaks down the packaging of information into individual files. Since files are dynamic, changing, a tension develops between two views: the one presented by current tools where users have the impression of acting on files in isolation, and the hypertext view of files participating in rich networks of relationships where each modification can cause changes that propagate into the network.

Consider software engineering. A large software project consists of many thousands of files, comprising requirements and design documents, source code, test cases, build files, bug reports, memos, email, and Web pages. There are many relationships between these files, such as a source file that satisfies a requirement stated in another document, or a test case that examines whether the code does indeed meet that requirement. In fact, software project files have an enormous number of relationships between them, and hence the project is really more a *complex information artifact* than a mere collection of files [28].

Chimera [3] and DHM [34] are two examples of hypertext systems whose goal is to capture the relationships between software project files. Once these relationships are in the hypertext system, they allow for rapid navigation to related files, as well as visualization and analysis of the relationship network. The act of instantiating the relationships makes concrete the effect that changing a single software file can have on its network of relationships, since modifying a file can create new relationships, and can alter or destroy existing ones.

Software engineering is a domain where best common practice involves maintaining previous states of the project. The discipline of software configuration management has developed to address the difficult issues of how best to record and compose these previous states, allow teams of developers to work on them without clobbering each other's work, guide and audit the software development process, and produce statistics based on this historical data [14]. However, since software development projects are currently divided into files, configuration management systems typically provide their features for files. Once hypertext functionality is added to a software development project, there is an immediate tension between file-oriented configuration management, and network-oriented hypertext functionality.

Think of a collection of project files whose previous states have been saved. In the absence of hypertext, there are still relationships between these files, but they are not captured in a hypertext system. By picking and choosing individual revisions of each file, it is possible to create arbitrary compositions of file revisions, even though some compositions may have relationships that are inconsistent. However, once the relationships are made into an explicit hypertext, this is no longer the case. First, since the relationships change over time, they too must have their previous states recorded. The hypertext links furthermore make it obvious when picking a single file revision causes a hypertext relationship to become inconsistent.

In hypertext parlance, a *node* is a chunk of data that can be stored as a file, a database record, or even, as is the case with Web pages in embedded devices, a sequence of read-only memory. Dynamic nodes can be arbitrary computational processes, a common occurrence on the Web. Relationships between nodes are called hypertext *links*. Though the detailed specifics of links vary across systems, in general a hypertext link associates two or more nodes, providing a means to navigate quickly between the nodes. Links can be either to entire nodes, or to a specific region, called an *anchor*, within a node. At their most abstract, anchors and links can both be arbitrary computational processes, one example being an automatic link between any word in a document, and its entry in a dictionary. This document concentrates its discussion primarily on non-dynamic nodes, links, and anchors, reflecting both the fact that the preponderance of surveyed systems do not provide versioning operations to handle such dynamism (an exception being [44]), and that the problem is inherently complex.

A collection of nodes, links, and anchors comprises a *hypertext document*, more concisely known as a *hypertext*. *Hypertext versioning* is concerned with storing, retrieving, and navigating prior states of a hypertext, and with allowing new states of the hypertext to be created by groups of collaborating authors. A key concern of hypertext versioning is reconciling the tension between node-oriented and link-oriented viewpoints. For example, in the Web, the tension has a clear reconciliation: nodes are dominant, and links are subservient chunks of embedded markup within nodes, with no independent identity. The other systems in this survey achieve more of a balance, making links first class objects, just like nodes.

Hypertext versioning capability has a pervasive impact on a hypertext system. It affects the storage of nodes, anchors, and links, the way people collaborate using the system, how navigation occurs, the naming of objects, and can reduce system performance. Hypertext versioning is an expensive proposition, since adding this functionality directly increases the complexity of a system for both implementers and users. Given its scope of impact and complexity cost, it is reasonable to ask whether the functionality is worth the trouble. What, exactly, can hypertext versioning be used for? High-value use cases include:

- **Software engineering.** As mentioned above, capturing the evolution of development files, as well as the relationships between them provides significant advantages for software development. However, due to the prevalent use of versioning and configuration management in software development environments, in order to provide hypertext support, the hypertext structure must also be versioned. The fact that existing hypertext systems for software development do not version links is a significant factor preventing their wider use in this domain.

- **Document management.** Documents too have a wide range of relationships between them, and can benefit from using hypertext to make them explicit so they can be analyzed, visualized, and navigated. Document management systems today provide the ability to store important document states, thus supporting collaboration and backtrack. Hence, just as for software engineering, hypertext support for document management requires versioning structure in concert with the versioning of documents.

- **Legal.** Laws, regulations, and tax codes, are an important set of complex information artifacts, chock full of interrelationships. It is important to store and retrieve previous revisions of these artifacts because in legal systems that prevent ex-post-facto laws, the version of a law that affects a case is the one in effect at the time of an infraction. This is especially relevant for tax codes, which change frequently. Hypertext support can make it easy to navigate to related laws, precedents, regulations, and codes. In this domain as well, adding hypertext support requires hypertext versioning capability.

- **Archival**: Since the Web is an important cultural artifact, the development of the Web should be recorded as it evolves. Right now, though the Internet Archive group is archiving the Web [9, 48], there is no “way back” machine that allows a user to dial in a specific time, and then be able to navigate around as if they were interacting with the Web as of that day. Hypertext versioning functionality can fill this gap. Archiving fills legal needs too. Regulated firms, such as insurance or brokerages that advertise or sell their products via the Web, have an obligation to archive their web sites so they can recover previous states for use in lawsuits.

- **Reference permanence.** One solution to the common Web problem of dangling links ("404 Not Found") is to ensure the linked-to information is always intact, by recording its prior states. In this way, link endpoints will always be present.

Hypertext versioning research offers much to the long-term goals of building a better Web, and the convergence of collaborative repositories.

At present, linking on the Web consists only of tags embedded in HTML. This is changing. The Xlink proposal [21] provides for linking between XML [6] documents, and Xlinks can be stored external to the documents they reference. The ability of open hypertext systems to link between arbitrary data types, including legacy formats that support neither embedded HTML-style links, nor Xlinks, is another driver pushing the Web towards providing links that are stored separately from the data. Of course, these links will change over time, and will require version control. How best should this functionality be provided in the context of the Web?

There is a growing convergence of functionality across multiple types of repositories that support the development of complex information artifacts, such as document management systems, configuration management systems, and Web content management systems. This convergence is most evident in the Web, with the proven track record of HTTP [27] and WebDAV [31] to map to a wide range of repositories, and with Delta-V [81] extending this to configuration management repositories as well. These repositories today do not provide support for first-class linking, and it is an open question how they might provide the hypertext versioning combination of first class linking and version support in the future. Since there is no current research consensus on how best to provide this functionality, it is difficult to make a case for its standardization, since such standardization is probably premature.

By performing an exhaustive examination of all known systems that provide hypertext versioning capabilities, this document aims to aggregate the current state of the art and thus highlight both what is known, and not known, and to provide a springboard for future research and standardization. Though a survey, the paper makes original contributions by providing a taxonomy of hypertext versioning systems (Section 2), surveying the data models used across systems (Section 3), comparing composite-based systems by use of a common scenario (Section 3.5.6), and by collecting together and summarizing known goals of hypertext versioning systems (Section 4). Throughout the survey common terms and concepts are defined, especially valuable in this field where the same idea can have four or five different terms used to describe it.

Among hypertext researchers, there are two common views on what constitutes hypertext. One viewpoint is that hypertext is primarily about navigation between nodes, the Web being a good example. The second view holds that hypertext is a data model that can support, among other features, navigation between nodes. The research on hyperbase systems [64] exemplifies this view. Other viewpoints abound; these two hold the most significance for this survey.

This paper includes in its survey those systems that provide hypertext navigation of either versioned or unversioned links among versioned nodes, emphasizing the navigational view of hypertext in its selection criteria. By insisting on hypertext navigation features, systems that only provide some form of relationship, such as relational database systems, or software development environments, are excluded. However, do not be misled. Despite supporting hypertext navigation, the majority of systems surveyed emphasize the data model view of hypertext, to the extent that there is a significant lack of research on how best to visualize and create user interfaces for navigation through versioned hypertext structures.

There are some exceptions to this survey's selection criteria. Since an explicit design goal of Palimpsest [24] and VTML [78] is to provide a data structure supportive of collaborative work and reference permanence, they are included even though they provide no explicit link traversal capabilities. Another exception is the PIE system [32], which is included due to the significant influence its hypertext-like data model has had on several hypertext versioning systems, despite the fact it has no hypertext navigation, and makes no claim to be a hypertext system. These exceptions point out that as a discipline at the crossroads of hypertext, configuration management, and computer supported cooperative work, hypertext versioning research finds insight in many places, and suggests it is premature to draw too tight a noose around a single definition of hypertext versioning.

2 Trends in Hypertext Versioning

2.1 Versioning for Reference Permanence

Ted Nelson, describing the Xanadu system in Literary Machines [61], was the first to fully embrace the problems inherent in changing documents and links, recognizing that change, like a cancer, slowly eats away at the consistency of relationship structures. If an entire document is deleted, links to it dangle. When a document is moved, links break unless repaired. The same problems recur inside documents when they're edited, where a linked-to region may be deleted, moved, rearranged, or otherwise modified, with links playing catch-up to maintain the consistency of the original relationship.

> "But *if you are going to have links you really need historical backtrack and alternative versions.* Why? Because if you make some links to another author's document on Monday and its author goes on making changes, perhaps on Wednesday you'd like to follow those links into the present version. They'd better still be attached to the right parts, even though the parts may have moved." [61, p. 2/25]

The Xanadu solution is to remember everything, prohibiting moves and deletes, while storing every change made to every document. In this scheme, links never dangle because linked documents are always present in their original form. While maintaining alternate document versions, and the intercomparison of alternates and versions are also important, it is maintaining the stability of references that most drives the design of Xanadu's version support. It is this goal that forces every change to be stored, and leads to move and delete being forbidden operations. This goal also introduces a decidedly hypertext issue into the realm of versioning, making it a valid topic of study in the hypertext community. Several researchers followed Nelson in the exploration of this topic, notably Vitali, Maoili, and Sola in the Rhythm system [57], Durand in Palimpsest [24], Vitali and Durand in VTML [78], Davis in his dissertation on link consistency in open hypertext systems [15], and by Simonson, Berleant et alli in version augmenting URIs to achieve reference permanence on the Web [72].

2.2 Versioned Data, Unversioned Structure

The hypertext systems KMS [1], DIF [30], Hyperform [84], and the Online Design Journal proposal [50], added version support, but only for nodes. Versioning of links, and hence structure, is not supported, consistent with their (implicit) view that links are invariant, and hence do not require independent change tracking. Similarly, these systems have no configuration management support. Unlike Xanadu, where the emphasis was on persistently storing very fine-grain changes to preserve link consistency, these systems version content at the node level, and do not track fine grain changes. Thus, they persistently store important states of the nodes, without recording the sequence of changes between steps. This makes it more difficult to preserve link integrity in general, and impossible in cases where a linked region has been extensively altered. Despite their similarities in version control support, these systems differ significantly in emphasis, with KMS focused on creating a full environment for hypertext authoring and browsing, DIF interested in hypertext support for software development environments, and Hyperform concerned primarily with separation of concerns in a hyperbase architecture. It is due to these diverse other interests that these systems do not explore hypertext versioning issues in depth.

2.3 Composite-based Systems

In the Neptune system [19, 20], Norman Delisle and Mayer Schwartz grappled with hypertext versioning, but with a different motivation. Instead of preserving link consistency, or merely versioning individual nodes, for them the key problem is collaborative teams writing hypertext documents. In order to collect related nodes and links together, to provide isolated work areas for each collaborator, and to support selection of consistent groups of individual node and link versions, Neptune employs a key abstraction called a *composite*. (In fact, Neptune calls this abstraction a *context*, but we prefer the Dexter term, composite. Both can be used interchangeably.) A composite (context) is just a container object, and within a composite logically related items can be grouped, such as the sections of a paper, and links between those sections. When each collaborator works within a separate copy of the composite, it provides the appearance that each collaborator is working on their own individual paper. The drawback of this work isolation is that to create the final, complete paper, each individual's contributions must later be merged. Versioning is a

key support technology for this, since it allows each collaborator's changes to be tracked, and merges to be recorded, while requiring that each composite select a consistent set of versions and links within which a collaborator works.

Composites collect together into one abstraction three concerns that are typically separated: collections or compound documents, which appear in numerous hypertext systems [20, 36, 34, 63], and are typically used for collecting together related nodes and links; *workspaces* provide work isolation in many SCM systems; and *configurations* have as their primary concern the selection of consistent sets of nodes and containment relationships. However, composites do address a uniquely hypertext problem, that of how to consistently version links between a consistent set of nodes, and how to support evolution of this link structure, in conjunction with the evolution of the nodes. The CoVer [36, 38, 37], VerSE [40], and HyperPro [63] systems, along with versioning support for the Nested Composite Model [8] in HyperProp [74, 73], and Melly's versioning support for Microcosm [59] all share Neptune's goal of exploring how to support hypertext structure versioning, and team document authoring, using composites.

2.3.1 PIE, a Change-Oriented Composite-Based System

Two landmark events for establishing the legitimacy of hypertext versioning as a research topic are Frank Halasz's 1988 Communications of the ACM article, "Reflections on Notecards, Seven Issues for the Next Generation of Hypermedia Systems," [42] (based on a presentation at Hypertext'87) and his Hypertext '91 conference keynote, "Seven Issues, Revisited," [43], since both identify versioning as one of seven critical issues for future hypertext systems. Version control was subsequently noted as an important issue for hypertext databases in the 1992 NSF Workshop on Hyperbase Systems [54], and the Hypertext'93 Workshop on Hyperbase Systems [52], cementing its importance within the community. The Halasz issues and the two workshops both legitimized and motivated hypertext versioning, making it easier to publish papers solely on this topic.

In addition to identifying versioning as a key issue, Halasz also singled out a configuration management system, PIE [32], asseerting that, "the goal is to adopt and improve on the versioning mechanism that appeared in the PIE system" [43]. Though not originally designed as a hypertext system, PIE does allow arbitrary relationships to be defined between nodes, and provides some support for navigating across these relationships, thus giving it a hypertext-like quality. PIE is the first change-oriented configuration management system [11], and Halasz extolled PIE's emphasis on logical changes that could span multiple nodes and relationships, as opposed to versioning operations tailored to individual file operations (state-based). PIE stores a set of logical changes in a container object it calls a *layer*, which, because it can contain both nodes and links, can also be viewed as a composite. But, because PIE only keeps changes in its composites, rather than a consistent state of the hypertext under development, and since these changes can be arbitrarily composed to create new hypertexts, PIE differs from Neptune [20], HyperPro [63], and HyperProp [73]. However, all these systems share the use of composites to contain the hyperdocument and its changes.

PIE directly influenced the work of Prevelakis [68], who set out to reimplement PIE specifically for hypertext application, independent of its original Smalltalk environment. Unfortunately, this work was never completed. PIE's change orientation also influenced Anja Haake's work on melding change-oriented and state-based versioning styles within CoVer [36], which has both composites that capture the entire state of the hypertext under development, as well as composites that capture only the changes between these states.

At the end of his 1991 keynote, Halasz reflected on the lack of versioning research since 1987, and noted that, "whether, in fact, versioning is important or not is still, I think, an important issue." Ironically, at that time most work on the topic was just beginning.

2.4 Web Versioning

The Web's lack of standard features for either browsing or authoring versioned content has led several researchers to investigate versioning for the Web. Hypertext links on the Web are embedded within HTML resources, and hence hypertext structure is not separate from data. As a result, it is not possible to version structure separate from data, leading to a focus just on versioning data (though [55] does provide a proposal for separating unversioned links from embedded anchors in versioned SGML content).

Initial work concentrated only on browsing Web resources augmented with a version history. Typically, these systems append a version identifier to a URL, and augment a Web server to parse the version identifier, and retrieve the resource from a version store, such as an RCS [76] repository. Pettengill and Arango [67] adopt this approach to maintain different versions of materials in a digital library, as do Simonson and Berleant et alli [72], but to achieve reference permanence for all uses, not just digital libraries. Another common architecture for adding versioning services to the Web is the "form fill-in" style, exemplified by BCSW [4] and WWRC [69]. These systems share the approach of using HTML pages to create a user interface for a revision control system, and work within the existing Web infrastructure to add versioning services. The limitations of this approach have led some to employ a "Java helper app." approach, wherein a Java application is downloaded into the browser and acts as an intermediary between the remote versioned repository and the user's local environment. Examples of this type of system are WWCM [46], MKS WebIntegrity [60], and WebRC [69]. Characteristic of all these approaches is their sole focus on versioning content, with no support for configuration management, or for versioning structure. Delta-V [81], a current working group within the Internet Engineering Task Force, is developing a standard protocol for versioning and configuration management of Web content, but will not address full structure versioning since structure is embedded within HTML links.

The Web-based versioning and CM systems just described assume that the Web server is responsible for maintaining the predecessor and successor relationships between revisions. Research at NTT Labs resulted in a proposal [65] that shifts the relationship management to the client. In this approach, the HTTP LINK method (now deprecated) is used to create predecessor and successor relationships between resources in a version history, similar to [29]. This has the advantage that version histories can span multiple servers without requiring cooperation between these servers, but has the drawback that clients must be well behaved, as a single misbehaving client can corrupt a version history. A related approach is the non-Web-based NUCM [77] system, a client-server CM system in which a NUCM client interacts with a remote NUCM repository server. This interaction occurs using primitive operations (similar to those provided by HTTP [27] and WebDAV [27]) upon which are implemented higher-level CM styles. This is similar to the NTT Labs. work in that the client is responsible for maintaining the consistency of the relationships in the remote repository.

2.5 Versioning for Open Hypertext

Several systems have been concerned with how to provide hypertext versioning support in an open hypermedia environment. The hypermedia version control framework developed by Hicks et al. [44] provides the HURL data model, along with a conceptual architecture that, together, are used by an open hyperbase to provide structure and data versioning services. Unique among the systems surveyed, this model supports computed anchors and links, and permits structure to be versioned independently from data. The data model and conceptual architecture were instantiated in the HB3 hyperbase management system [53], also described in [44]. Hyperform [84] similarly provides versioning services in a hyperbase, but with fewer services. The proposal for adding versioning to the Chimera system [82] shares the goal of providing structure versioning for open hypertext, but does so for an open linkbase system, where the data is controlled by an external repository. As a result, a focus of this work is how to associate and synchronize the versioned structure with the externally versioned data. Melly's work on versioning in Microcosm [59] also addresses hypertext versioning for an open linkbase system, but does so by creating context-like structures called *applications* that contain references to an active set of documents and links, using the context to avoid the document and linkbase synchronization issues in [82].

3 Data Models for Hypertext Versioning

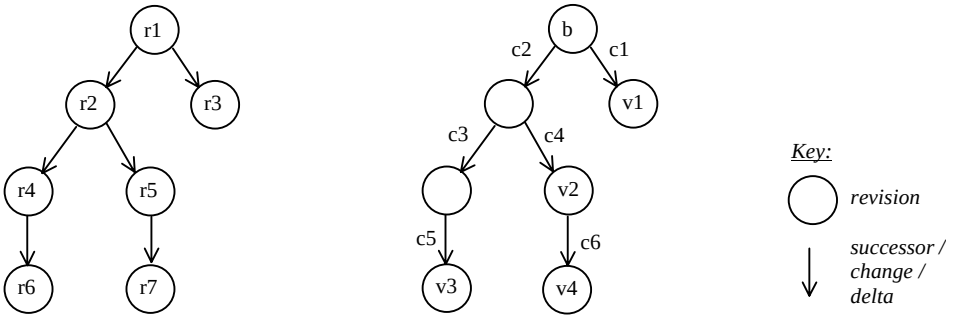
Just as the admonition in finance is "follow the money," for a deep understanding of hypertext versioning systems one must follow the data model. A system's hypertext data model directly affects how it is versioned, constraining possible features and determining likely problems. For example, if the data model embeds links in documents, as in HTML, it is not possible to version structure separate from data. Furthermore, since the language and terminology used to describe systems at times varies significantly within the hypertext versioning literature, examining the data model can tease out similarities where there appear to be only differences.

3.1 Versions, Variants, and Alternate Versions

The typical user of today's computers works with files that are not under revision control. For example, unless some additional version control facility is employed, all files in the Unix, Windows, and MacOS operating systems are unversioned. More precisely, an *unversioned item* is a data item where there is only one state of the item, the current state, and modifications overwrite it.

People use "version" to mean many things. A version can represent a modification to the content of an item, as in "the version from last Tuesday," and these modifications can include those made by the original author(s), or even by different authorities. For example, [56] mentions that, "a hypertext edition of *The Waste Land*, for instance, would enable comparison of T. S. Eliot's original draft, Pound's corrected copy, and the final published version" (p. 303). Version is also used to represent a mechanical change to an item without changing its meaning, as in the "PDF version of the document." In the context of source code development, version can also mean a change made to accommodate a different platform, operating environment, or feature, as in "the MacOS version" or the "debug version" of a source code file. Version can also mean a natural language translation, as in "the German version of the document." Finally, there is the notion of a version that captures intellectual precursors, even though the derived work is different enough to have a separate identity and history, as in the Jimi Hendrix version of the Star Spangled Banner, or the King James Version of the Bible.

Providing computer support for these different uses of the term version implies different features. For example, recording different versions of a data item involves different operations from mechanically deriving a different version of an existing item. In order to separate these different features, the configuration management literature has developed distinct terms for each sense of "version" [11].



a) State-oriented versioned item *b) Change-oriented versioned item*

Figure 1 – A versioned item shown with state and change orientation.

A *versioned item* is a data item for which the system persistently records its evolution. The kinds of data items that are versioned in hypertext versioning systems include anchors, links, documents/nodes, composites, and contexts. The evolution of these items can be viewed in two ways, as either a set of exact states of the data item, or as a set of changes between states of the data item. These are known as *state-based* and *change-based* versioning respectively [11]. Within state-based systems, a *revision* is a distinct state of a versioned item. RCS [76] and SCCS [70] are classic examples of state-based systems, while PIE [32] exemplifies the change-based orientation. Since most state-based systems persistently store the difference between successive states, called a *delta*, it is worth asking what is the fundamental difference between state-based and change-based systems. An essential difference is whether the states, or the changes are named, and hence first-class objects in the system. In state-based systems, the states are named with revision identifiers and the deltas are unnamed, while in change-based systems it is the changes that have identifiers, and intermediate states between versions are anonymous. Figure 1 highlights the difference between state-oriented and change-oriented versioning. In Figure 1a, all of the states are explicitly named revisions, while the changes have no separate identity. In Figure 1b, all changes are explicitly named, and several unused intermediate states are anonymous. The first state of Figure 1b is the initial, or baseline state, to which all changes are applied. Version v1 is constructed by applying change c1 to the baseline, and version v3 is constructed by applying changes c2, c3, and c5 in order to the baseline.

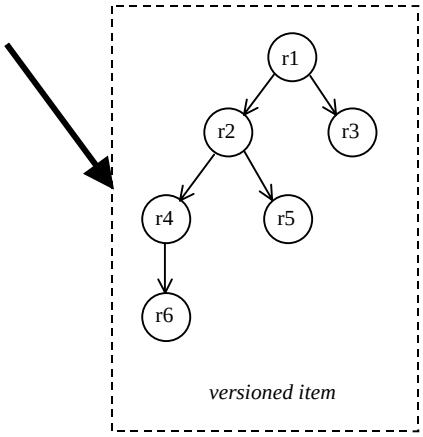
As a further nuance for state-based versioning, the Palimpsest [24] and VTML [78] delta formats are designed to record the changes between revisions in a state-based system, yet maintain a fine-grain history of the changes between states, recording down to the byte level exactly who made each change. Durand describes the set of operations that record these modifications as being *change complete* [24]. This is in contrast to the more typical deltas employed by, for example [46, 30, 20], which do not record the exact set of steps performed to go from one revision to the next, using a set of operations that have the property of being *version complete* [24]. As an example of the difference between the two styles, when two revisions from parallel branches are merged using VTML, the system explicitly records, with the MERGE tag, which individual changes belong in the merged revision, capturing exactly the changes, and their cause – a merge operation. In contrast, traditional deltas only record insertions and deletions of whole lines, do not record semantic information, like merging, and make it difficult to reconstruct who made a particular change.

Versioned items are represented using a graph structure, where the nodes of the graph are revisions, and the arcs are predecessor and successor relationships. In the simplest case, known as a *linear version history*, the graph is a straight line, where no revision has more than one predecessor. When a revision may have more than one successor, but only one predecessor, then revisions form a *tree version history*. This is used when different branches of the tree represent variants, as in SCCS [70]. If it is possible to merge branches together, thereby allowing revisions to have more than a single predecessor, then the revisions form a *directed acyclic graph (DAG)* [12]. This occurs when branches are used to represent different developments by collaborators working simultaneously (in parallel), and the contributions of each collaborator are merged together. As Figure 1 highlights, the type of graph structure formed by the predecessor and successor relationships is orthogonal to whether the system is state-based or change-based.

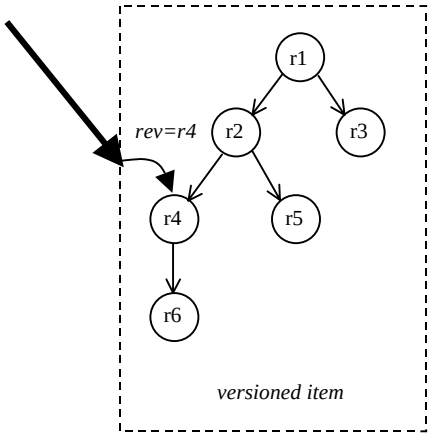
A *variant* is an alternate form of a data item. When a variant can be mechanically derived, it is termed a *rendition*. It is useful to distinguish between variants resulting from human modification that does not change the item's identity (e.g., a natural language translation, or an OS-specific change) and modification that does change the identity (e.g., King James Version of the Bible), however there are no established terms to capture this distinction. In part, this may be due to most systems being operated by a single institution, such as a development organization operating a configuration management system, and hence the kind of cross-organization change of ownership that typically accompanies changes in identity cannot be captured. Literary Machines [61] uses the term *versioning by descent* when the document owner creates a variant, and the term *versioning by inclusion* when another user creates a variant with distinct identity. However, these terms are limiting, since branches of a version tree are used not just for representing variants, but also for capturing distinct revisions that are used to isolate the work of simultaneous collaborators. The term "versioning by inclusion" isn't precisely right either, since it is reasonable to discuss creating a variant by performing a copy operation, and hence the word "inclusion" ties the term to a particular implementation strategy. We prefer to use the term *variant* for non-identity changing alternates, and *alternate version* for externally derived variants that do change the item's identity (e.g., the Jimi Hendrix Star Spangled Banner is an alternate version of the American national anthem.)

3.2 Version Selection

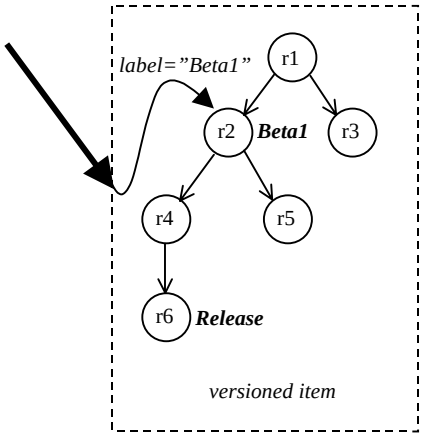
Version selection is a fundamental issue faced by any system that allows a relationship to point to a versioned item [63]. For example, in hypertext versioning, this occurs when a binary hypertext link has an endpoint at a versioned document, and in both configuration management [11] and hypertext versioning systems, containment relationships encounter this problem when the destination of the containment relationship ends at a versioned item. For example, the ClearCase [51] and Continuus/CM [12] systems both encounter this issue when a revision of a versioned object must be mapped into a single location in the filesystem namespace, selecting one revision from the versioned item that is contained by its parent directory.



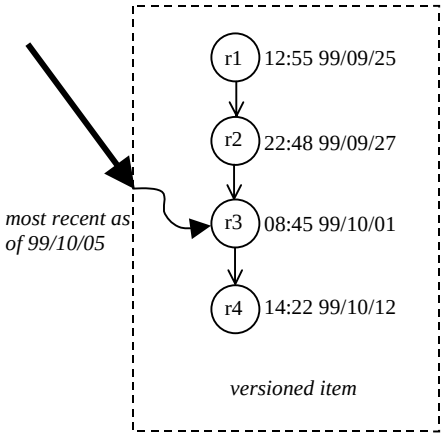
a) relationship endpoint with no version selection



b) relationship endpoint with explicit selection of a revision



c) relationship endpoint with selection based on label value



d) relationship endpoint with time-based selection

Figure 2 – Common mechanisms for version selection within a versioned item.

Figure 2 shows several common mechanisms for how the endpoint of a relationship can select a single revision within a versioned item. In Figure 2b, the revision is explicitly named by the relationship endpoint, and this endpoint will always select the specified revision, in this case revision r4. Figure 2c shows revision selection using a *label*, a human-readable short name attached to a revision. The relationship endpoint is selected by picking the revision with a matching label. A label can be moved from revision to revision over time, and hence a relationship endpoint that employs labels will select a different revision if the label is moved. In this way, labels provide a valuable form of indirect addressing, allowing selection based on an abstraction, such as a software release, which can float from revision to revision depending on which revision best matches the abstract description. In contrast, the direct addressing provided by explicitly naming a revision is more brittle; it is far easier to change a single label, and have relationships automatically track to the new revision, than to change every revision that directly addresses a revision.

In time-based endpoint selection, shown in Figure 2d, the revision endpoint is selected based on a time expression, typically the most recent revision as of a specified time. Another common mechanism is

choosing the most recent revision (as of the time of the request) within a versioned item. However, as noted in [63], choosing the latest revision in a version history tree will yield undesirable results, as the latest revision may switch from branch to branch in the tree over time. A common solution to this problem is to give each branch its own identity [51], and then select the latest revision on a specific branch.

Instead of offering a few specific mechanisms, such as label-based or time-based selection, some systems offer a full query-based mechanism. In both [44] and [36], there are attributes associated with every revision, and it is possible to construct complex predicates involving any of these attributes, such as "latest revision with status of released" [44]. However, Østerbye notes in [63] that this technique can increase the cognitive overhead for the user who must construct such complex queries versus using some system-defined default selection criteria.

3.3 Versioned Nodes, Unversioned Links

A straightforward approach for versioning in a hypertext system is to version the node contents, without providing any versioning of links. Reasons for adopting this approach vary. Hyperform [84], which focuses on providing extensible hypermedia services within a hyperbase, has no provision for versioning structure because "it is impossible (at this point in time) to provide general hyperbase support for versioning structure in hypertext without introducing a fixed data model" ([84], p. 256). Xanadu [61], a system where versioning is very important, does not version structure both because there appears to be an assumption that links capture invariant relationships between text spans, and also due to its emphasis on using versions to maintain reference permanence, thus requiring the linked-to version to always remain the same. After following a hypertext link to a specific document revision, it is possible to navigate along predecessor/successor and alternate version links to find other revisions and variants. DIF [30] and KMS [1] only allow links to entire nodes, providing no support for linking to sub-regions within a node. As a result, they do not share Xanadu's commitment to reference permanence. However, both do appear to share Xanadu's assumption that links capture invariant relationships, and hence do not require versioning; certainly neither DIF nor KMS version structure. Finally, since Web based versioning systems [69, 46, 81] operate on HTML where the links are embedded in the documents, data and structure are inseparable and hence versioning of data records versions of links as a side effect.

Looking at their data models, the most significant differentiator among this class of systems concerns the treatment of predecessor and successor relationships: are they first class links, or are they hidden within the version repository? In Xanadu and KMS, predecessor and successor relationships are treated the same as any other hypertext link in all aspects, including persistent storage. In the NTT Labs. Web versioning proposal [65], predecessor and successor relationships are represented using links made with the HTTP LINK method (now deprecated), non-HTML links that are stored external to a resource body, and that have as their destination an entire resource. Thus, while this system does use a different link mechanism for capturing predecessor and successor relationships than for navigational links, they are still stored using system link services. The advantage of representing predecessor and successor relationships using hypertext links is that it encourages distribution of the version graph across multiple servers. This permits the representation of alternate versions of items where multiple people from different institutions work on potentially many variations of the same item. However, it has the drawback of making operations that span the version history expensive. For example, since labels are unique across a version graph, setting a label is expensive because it requires a full revision graph traversal to guarantee uniqueness [81]. Furthermore, if configuration management capability is desired, efficient implementation requires efficient revision graph operations. There is also a simplicity argument: by using hypertext links for all relationships, the system need only support one kind of relationship, and one way of manipulating and navigating them.

In contrast, WWRC [69], WWCM [46], BSCW [4], MKS WebIntegrity [60], and Delta-V [81] do not represent predecessor and successor relationships using hypertext links, instead delegating this to the facilities of the version store, often RCS [76], or the Revision Control Engine (RCE) [47], an RCS descendent offering a programmatic interface for RCS operations. While these systems all support a mechanism for retrieving a revision history listing, frequently a human-displayable hypertext document, there is a distinction between being able to produce a synthetic document containing hypertext links to all the revisions, and actually using hypertext links to persistently represent and store the predecessor and successor relationships. Due to the version store having complete control over all predecessor and successor relationships, it can offer the advantage of storing only the changes between revisions instead of

the entire revision contents, and can provide better consistency maintenance of the version graph than a distributed approach.

Directly related to the treatment of predecessor and successor relationships is whether systems offer any object to represent a versioned item. This affects the kind of version selection service the system provides. For example, neither KMS, Xanadu, nor the NTT Labs. versioning proposal have any object which represents a versioned item. As far as these systems are concerned, there are only objects and links. However WWRC, WWCM and Delta-V all do have an abstraction to represent versioned items, permitting operations like setting a label. The Delta-V protocol, along with [67] and [72], all provide a distinct URL for each revision and have a versioned item abstraction. This provides a challenge on the Web, since to avoid disruptions to relative URLs, the URL a resource had before it was placed under version control must also be the URL it has afterwards. Or, restated, the URL of the resource does double duty both as a versioned item identifier, and the identifier for a specific versioned resource. The same problem occurs when mapping a versioned item into a single name in a file system. [67] and [72] allow the URL to be decorated with a revision identifier (or a timestamp in [72]), a technique that only works for resources that aren't collections, that is, it only works for leaf nodes. Once configuration management is considered, both the final resource name and all internal collections need to have revision identifiers. The Delta-V protocol, which does address configuration management, uses a workspace approach in which a revision selection predicate is associated with the workspace. The workspace is then used as a filter, in conjunction with undecorated URLs, retrieving only the revision that matches the revision selection predicate.

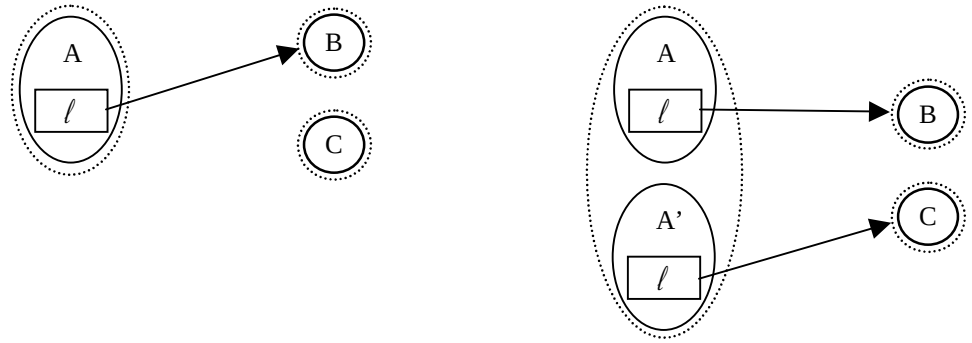
3.4 Versioned Links and Structure

A frequent goal of hypermedia systems is the ability to separate link structure from the data being linked. In systems that support this, it is natural to want to version this structure independent of the data being versioned. To highlight challenges inherent in representing versioned links, Figure 3 below (adapted from Figure 2 in [63]) presents a simple scenario of moving the destination of a link from node B to C, showing cases in which the nodes are versioned, but the link isn't, and vice-versa.

Figure 3a is representative of embedded link systems like the Web, where moving a link from B to C requires a change to the document containing the link, resulting in a new revision. In Figure 3b, links are first class entities external to nodes, but are unversioned. In order to change the endpoint of the link while preserving the information that A initially had a link pointing to B, a new revision, A', must be created. If a new revision of A were not made, A would then have two links, and it would be impossible to tell that it initially had only one link pointing to B. In Figure 3c, links are versioned, and are external to the unversioned nodes. Here changing the endpoint results in a new link revision, but causes link traversals to be ambiguous, since it is not clear which link version should be used for the traversal.

None of these approaches is entirely satisfactory, since they do not achieve the goal of versioning both nodes and links, while keeping the link structure separate from the linked data. Figure 3c, in which the links are versioned and separate from the data, comes closest to meeting these goals, but has the drawback that the nodes are unversioned. The remedy is obvious: version both nodes and links. This approach has been followed in the Hypermedia Version Control Framework [44], CoVer [36], VerSE [40], and HyperProp [73].

The Hypermedia Version Control Framework provides excellent separation of versioned structure from versioned nodes data. In the framework, anchors and links can be static, or can be arbitrary computational processes. A separate container, the *association*, contains anchors and a link by reference. A super-container, the *association set*, contains a set of associations by reference. An association set does not contain any node data, and contains only structural information, thus separating structure from data. Since association sets are versioned, they achieve the goal of separating versioned structure from versioned data. Using association sets, it is possible to have multiple structures, all versioned, that can be applied to the same set of versioned data.



a) nodes versioned, links embedded in nodes

b) nodes versioned, links external to nodes

c) links versioned and external to nodes, nodes unversioned

Figure 3 - A scenario where the endpoint of a link is moved from node B to C.

3.5 Composites

The systems that provide the richest hypertext versioning services do so by employing version-aware *composites*, a set of hypertext objects. Composites fulfill many roles in hypertext systems, providing organization of large collections of nodes and links into smaller units, a form of modularization (exemplified in the hypertext versioning literature by [73, 20, 63, 44]), and information hiding via encapsulation [40, 73]. Composites can also be used to model compound documents, for example, the combination of some text and image objects to model a document containing figures; Dexter composites are ideal for this use [41]. It is this use in modeling compound documents that distinguishes a composite from a filesystem directory or collection, though the underlying notion of a set of objects is the same.

The ability to subdivide a hypertext into smaller pieces fills the same need as hierarchical directory structures in filesystems, subdividing a huge space of data objects into smaller, more manageable collections. Unlike a filesystem, however, the containment hierarchy does not define the name of a hypertext object, and while context-using hypertext versioning systems give each hypertext object a unique identifier, and sometimes give each object a name, there are no rules for concatenating the names of nested composites together into a pathname. With respect to naming, the literature appears to agree with Conklin's view that users are distracted if they are required to name every node and link that is created [10]. Systems

that have need of the containment order of an object must re-create an abstraction akin to a file path such as the “perspective” concept in [73].

Composites are linkless structures in hypertext systems, providing composition features without using links to represent the membership/containment relationship. Using only links to model the structure of a set of logically related nodes leads to several problems. First, the structure doesn’t have a clearly defined “head” node that can be used to link to, or act as a handle for the structure as a whole. Furthermore, linked structures offer little ability for reuse for repeated structural patterns. Composites address these problems by aggregating together related nodes using a linkless container [35].

Collaborative work on hypertexts yields another use for composites: keeping an entire hypertext consistent as it is developed. Due to the interlinked structure of hypertexts, changes to nodes can result in changes to links, and vice-versa. This is exacerbated when links are fixed to specific anchor locations within the nodes, when every edit to a node changes the link anchors. The tendency for changes to explicitly propagate into the network distinguishes hypertext authoring from single item authoring. As a result, a hypertext authoring system needs to allow work to take place on a hypertext network, and furthermore this hypertext should contain nodes and links that are consistent with one another. That is, if a specific revision of a node is part of hypertext network that is being edited, links to or from that node should be those that were active at the same time the node revision was active. Composites address this problem. By placing the hypertext into a composite, all changes to the hypertext network can be kept within the composite. When the composites themselves are versioned, each revision freezes an entire, consistent state of the hypertext, and hence if a prior state of the hypertext is desired, all that is needed is to revert to a prior state of the composite. The prior state will, of course, be consistent, since it was consistent when it was originally frozen. Configuration management systems face a similar problem, due to the implicit dependencies between source code files. As will be discussed below, they use a similar concept, the *workspace*, to allow work on a consistent set of files.

3.5.1 Containment: Inclusion or Referential

A difference among composites is exactly how they contain their hypertext objects: by value, or by reference [34, 35]. When a composite contains its objects by value, it creates an inclusion relationship where each object can only belong to a single composite. In contrast, containment by reference allows each object to participate in multiple composites. This choice affects the behavior of operations on composites. For example, deleting a composite that included its members by value would be expected to delete the members as well, while deleting a by reference composite would not affect its members. In the hypertext community, Halasz [42] first noted the distinction between inclusion and referential containment, although this difference has long been known, cropping up whenever containment structures are introduced. One early example is the need for hard links in the Unix filesystem. The sole hypertext versioning system whose composites include members by-value is Neptune [20], while HyperPro [63], HyperProp [73], CoVer [37], VerSE [40], and the Hypermedia Version Control Framework [44] all contain their objects by-reference.

In the change-oriented PIE system [32], a composite called a *layer* represents a set of changes from a baseline to multiple nodes and links, and a composite called a *context* represents the sum of several layers. Due to the influence of PIE, Neptune and HyperPro also use “context” to describe their version-aware composites. With respect to containment in composite-like structures, PIE is a hybrid, storing nodes by-value in a layer, but including layers by-reference in contexts (in fact, layers are linked to contexts).

3.5.2 Composites Modeling Versioned Items

Composites are frequently used to model versioned items in hypertext versioning systems. When a composite models versioned nodes, contained nodes are revisions, and links represent predecessor/successor relationships. In HyperPro, HyperProp, and the Hypermedia Version Control Framework, composites also model the revision history of versioned composites (i.e., a composite that contains a number of other composites), and in this case the contained composites are individual revisions of the versioned composite, with links modeling their predecessor/successor relationships. A good example of the lack of convergence of terminology in this literature is the fact that each system has a different name for the versioned item composite. CoVer calls them *mobs* (multi-state objects), HyperPro uses the term *VersionGroup*, HyperProp calls them *version contexts*, and the Hypermedia Version Control Framework

terms them *version set histories*. While Neptune does have the concept of a linear revision history, it does not use composites to model this history, and PIE, being change-oriented, does not have a notion of version history at all.

3.5.3 Contained Objects

An important aspect of composites is the kind of objects they include. All composite-based systems allow nodes and links to be included in composites, but vary in the exact semantics of node and link inclusion. For node inclusion, the surveyed systems show examples where the node is included by-value (Neptune), where the node is contained by including by reference the entire composite representing the node's versioned item (HyperPro), and the more typical approach where the node is included by a reference to a versioned item, with version selection criteria selecting one or more revisions (CoVer, VerSE, NCM, Hyper Media Version Control Framework). In CoVer and NCM, it is possible for a reference to return more than one revision that satisfies the version selection criteria, in this way modeling alternate revisions. VerSE, though it is based on CoVer, only allows a single revision to be returned, preventing alternates to reduce complexity.

For links, issues include whether a link must be wholly contained within a composite, or whether a link can connect nodes in two different composites, and, if so, which composite contains the link. Taking this to its logical conclusion, can a link belong to a different composite from the nodes it links [20], or must it belong in a composite at all? In Neptune, links are allowed to span contexts, but at least one node must be in the context containing the link. In contrast, the Hypermedia Version Control Framework [44] has a special composite called an *association set* that contains only associations, the structural element in the framework. The Hypermedia Version Control Framework also has a general-purpose composite that can contain nodes, and other composites. The HyperPro [63] system has links that only connect nodes in its context, the exception being revision links between successive revisions of a context. In PIE [32] a layer can only contain nodes and links (whose endpoints must reside within the layer), while a context can link to layers, as well as other nodes.

CoVer [36, 37] and VerSE [40] use the term *task* for their composites, based on the insight that each change to the hypertext contained by the composite is performed to fulfill some task. Task examples include, "Sketch Outline", "Elaborate Example", and "Review by Joerg" [36]. CoVer and VerSE both support tracking the derivation history of tasks, as well as hierarchical decomposition of tasks into subtasks. Tasks are implemented using SEPIA composites [75], and contain only nodes and links, where the link endpoints must reside in the composite, in this way acting to partition the overall hypertext space.

The Nested Composite Model [73] has nodes that can belong to one composite, itself contained within another composite, and this composite potentially contained within yet another composite, and so on to an arbitrary level of nesting. In this model, links are defined so they must belong to one of the string of nested composites that eventually contain the endpoint nodes. One benefit of this approach is the ability to define link visibility rules tied to the nested composition. For example, in a situation where composite A contains composite B that contains node N, if a link containing N is defined to be in composite A, then when the user interface is showing node N through composite B (ignoring composite A), the link will not be visible, since it is defined in composite A. The Hypermedia Version Control Framework and the HyperPro system also permit composites to contain other composites, but do not have similar link visibility rules, or link definitions that include the containing composite.

3.5.4 Composites Providing a Consistent Slice Through History

Hierarchical decomposition and information hiding are two benefits that composites bring to version aware and unaware hypertext systems alike. For hypertext versioning systems, composites provide an additional important feature: providing a consistent slice through the history of all contained objects and links. As noted by Conradi and Westfechtel in [11], when there are a number of versioned items, the number of potential compositions of these items increases combinatorially. However, only a few of the possible combinations are actually consistent, or useful. By capturing a consistent slice, or combination, of nodes, links, and other contained items, a composite allows consistent hypertext navigation through the state of a hypertext at either the present, or a previous time.

There are two issues involved in providing this slicing capability, *selection*, selecting specific versions of all contained versioned items, and *consistency*, ensuring that, together, these selected items make sense, or are useful together. The selection issue is handled by the version selection capabilities of composites, with the hypertext versioning literature containing several solutions. HyperPro provides a link type called a generic version link where the endpoint is a versioned item. When a link traversal occurs, a specific revision of the versioned item is selected using time-based version selection criteria stored on the composite that contains the link. Thus the composite's sole version selection criteria is used by all generic versioned links it contains. HyperPro also has a non-generic link that ignores the composite's selection criteria and always refers to a specific revision. The Hypermedia Version Control Framework has composites where each item is contained by either a static reference, or a dynamic reference. A static reference selects a specific revision, and the dynamic reference uses an attribute predicate to select its revision. CoVer similarly has static and dynamic references, using a query language for version selection by dynamic references. This is different from HyperPro, in that each reference in CoVer and the Hypermedia Version Control Framework individually chooses the object to select, while all HyperPro generic version links use their composite's revision selection criteria. HyperProp uses a different approach, storing version selection criteria on a reserved anchor of the "version context" composites that model versioned items. If the version selection anchor is undefined, a HyperProp link can alternately contain the version selection criteria. Since Neptune contains all nodes and links by value, it has no need for version selection criteria.

The major departure between composite-based hypertext versioning systems and software configuration management systems lies in their approach to solving the consistency issue. In hypertext versioning systems, composites are used both for maintaining isolation between collaborating authors, and for creating consistent slices of the hypertext by preserving the state of development at specific instances. That is, as collaborators work on a hypertext, the view they have of the hypertext is the only allowable view that can be returned to in the future. In SCM systems, there is typically a separation between the version selection criteria used when creating workspaces, and the selection criteria used when creating a consistent set of objects used for system building. This separation does not exist in hypertext versioning systems. One explanation might be that hypertext versioning systems do not need to "build" a system. SCM systems must be able to create a configuration that can be operated on by a system build tool like Make [26], which feeds source code into a compiler to create an executable program. Since hypertext versioning systems do not need to perform system builds, they do not need to create special configurations for the build operation, and hence do not need to separate work areas (workspaces) from system build areas (configurations). Furthermore, combining together workspaces and configurations has the advantage of simplicity for authors.

The hypertext versioning approach provides the advantage that the preserved hypertext states are always consistent, at the cost of precluding creation of any hypertext other than those explicitly preserved. Even though the version selection facilities of CoVer and the Hypermedia Version Control Framework are rich enough to make the construction of arbitrary hypertexts feasible, no system facilities support this beyond version selection on individual references. Minimally, a system should provide the ability to change the version selection criteria for all references in a composite with a single operation, thus supporting rapid, efficient configuration changes. Of the composite-based systems, HyperPro, which centralizes version selection criteria for all links in their composite, comes closest to supporting configurations. However, HyperPro only supports version selection on links, assuming that link traversal is the only mechanism ever used to access a node, and hence version selection on nodes is not needed.

3.5.5 Composites and Collaborative Work

Composites are a key data structure for supporting concurrency control among multiple authors working simultaneously on a hypertext. Many concurrency control schemes assume that no versioning facilities are available, and hence mediate access to the lone instance of an object. These include locking [83], exploiting the hierarchical structure of documents as in Duplex [66] and Alliance [71], and real-time per-operation update protocols as in Grove [25] and Prep [62]. These schemes explore how the goal of perfect isolation from changes by other collaborators is traded off against the goal of constant access to the object for editing. Locking provides complete isolation from collaborators, but only allows one collaborator at a time to work, while per-operation update protocols offer no isolation, with collaborator's changes reflected

almost immediately, but do allow multiple collaborators to work simultaneously. Versioning offers a way around this tradeoff, allowing both full isolation and availability. With versioning, collaborators can work in parallel, each collaborator storing their work in separate revisions, often as separate branches of the object's revision history. To combine each collaborator's contributions, a merge step is required where modifications are reconciled to create a single consistent object. Versioning also has the advantage of supporting collaborative work on all types of objects, while hierarchical decomposition of documents and per-operation updates require significant knowledge about an object's internals [83]. But, as noted by Grinter [33], merging can be a complex operation, and this complexity can lead to wariness of situations that require merging. Thus, while versioning appears to resolve the tradeoff between isolation and availability, it does not come for free, introducing instead a new tradeoff between avoiding the cognitive overhead of merging, and achieving full isolation and availability.

Neptune [20], HyperPro [63], HyperProp [73], and VerSE [40] all support collaborative work by having each collaborator work in a separate composite, each composite containing the hypertext under development. Since each collaborator has a separate composite as their work area, the composites provide work isolation, and since each composite contains a complete hypertext that was derived from a pre-existing consistent state of the hypertext, it holds an internally consistent view of the hypertext. In SCM systems [11], the workspace fulfills the same role as composites in hypertext versioning systems, providing work isolation and a consistent view of the objects under development. The Delta-V protocol [81], influenced by SCM systems, provides a workspace concept to achieve work isolation for versioned Web resources.

3.5.6 A Scenario of Collaborative Work Across Systems

Due to the complexity of composite-based hypertext versioning systems, it is difficult to understand the differences among them just by describing individual aspects of their data models. To address this difficulty, and enhance understanding of composite-based hypertext versioning systems, the following sections present a common collaborative work scenario that is modeled using the concepts and abstractions of Neptune, HyperPro, CoVer, and HyperProp. By integrating multiple data model aspects, such as whether each system includes objects by-reference or by-value, whether links include within-node endpoints or are node to node, and whether links are versioned, the scenario permits a rapid examination of how these differences end up affecting composites over the course of the scenario.

Figure 4 provides an overview of the scenario, a small hypertext consisting of four documents, connected by five links. Two different authors, "author 1" and "author 2" simultaneously work on the hypertext for a period of time, and then combine their work once they're finished. The scenario involves cases where the same document (D) was modified by both authors, where one author deletes a link (ℓ_γ) and the other doesn't, where one author creates a link (ℓ_η) the other doesn't, and where both authors create the same relationship, but as separate links (ℓ_ζ and ℓ_θ). For each system, the scenario demonstrates parallel work, changes to hypertext nodes, and changes to the hypertext link structure. Where supported, the scenario also shows merging of the authors' parallel work sessions.

Neptune's handling of the scenario is shown in Figure 5. The most significant difference between Neptune and the other composite-based versioning systems is that Netpune contains its objects by value. Thus, when a new context is created, or a new revision is made for an existing context, all nodes and links within the context are duplicated. Thus, it should come as no surprise that Neptune uses more objects than any other system to represent the scenario. Though inclusion containment has the disadvantage of significant duplication of nodes and links, it has the advantage of eliminating the need for revision selection rules on links and containment relationships. Since every node and link is contained within a specific context, there is no need to select a specific revision from a pool of objects. While Neptune's object duplication appears on the surface to make per-item revision history recording difficult, in fact Neptune can record complex version histories. Neptune records a linear version history for each object, and each line of development in the scenario has its own linear history. For example, node A has a main context history, an author 1 history, and an author 2 history. These separate histories are reflected in the version numbering system in the scenario – in the main context, revisions are labeled "M,1" and "M,2", reflecting that these are revisions in the main context history, while in the author 1 context, revisions are labeled "A1,1", "A1, 2" to show that these are revisions in the author 1 context. Neptune also records the derivation relationship, so it is possible

to follow a revision history from one linear history to another, for example, from the author 1 history back to the main context history (this capability is shown in Figure 7 of [20]).

A quick comparison of HyperPro's handling of the scenario in Figure 6 with that of CoVer in Figure 7 and HyperProp/NCM in Figure 8 immediately highlights a difference in containment. HyperPro contains the entire versioned item (and hence all of its revisions) within its composites, unlike CoVer and HyperProp/NCM that contain only a single revision from within a versioned item. But, since HyperPro contains its objects by reference, this containment strategy does not lead to a proliferation of objects. The same versioned items, and individual revisions are simply contained in multiple contexts. In HyperPro it appears that all objects are accessed via a link traversal, and this handles the addressing difficulty caused by the availability of all revisions of a versioned item. That is, since all revisions are recursively contained within a composite, when accessing a versioned item, there is no rationale for choosing one of its revisions over another. This dilemma is solved by the revision selection rule associated with each context. Though the revision selection rule only affects link endpoints, since all objects are retrieved via a link traversal, it has the effect of selecting a revision for containment in the context.

By anchoring a link on a specific revision, and using the context's revision selection rule to pick the endpoint, HyperPro is able to avoid versioning links. As links are node-to-node, it is not a problem if there is a new revision of the endpoint node, since the revision selection rule can be modified to pick a new revision *without changing the link*. Similarly, since the start of a link is associated with a specific node revision, and since links have no anchor points within a node, once this starting revision has been selected, the link is impervious to further node modifications. By implication, if HyperPro links were associated with specific anchor endpoints within a node, this scheme would not work, as the link endpoint location would change from revision to revision at the target node. The drawback to this scheme is that when there is a new node revision, all links starting at that node must be duplicated, and, unlike Neptune, there is no revision tracking across these duplicated links. Thus, while the numbering convention in Figure 6 indicates that ℓ_{α}, $\ell_{\alpha\alpha}$, and $\ell_{\alpha\alpha\alpha}$ (shown in the topmost context at time t_5) are all revisions of the same link, HyperPro does not record this fact, and cannot display a revision history of this link. They are three separate links to HyperPro. However, despite not versioning links, HyperPro can version structure, since links are contained within contexts, and contexts themselves are versioned.

Focusing on the use of composites within the scenarios, having a separate work composite for each author, a feature supported by Neptune, CoVer, and NCM, provides a better separation of work areas than if the first author works on the main branch of a composite's version history, while all other authors work on side branches, as is the case in HyperPro. When there is a separate composite for each collaborator, it is clear where each author's work takes place. Furthermore, by having a separate branch for the state of the system prior to collaboration, it is clear which state of the system new collaborators should use as their starting point.

After viewing the complete scenario, CoVer's task view emerges as a valuable simplified view of the ongoing work, especially since CoVer provides a specialized user interface focused only on the task view. The simplified visualization is the most significant benefit of tasks, however. The laboriously constructed distinction between task and state based versioning in [37], in the end acts only to increase complexity, as there is not a significant difference between composite-based versioning in CoVer and composite-based versioning in HyperPro, HyperProp/NCM, and Neptune. Certainly CoVer is not change-based in the same way as PIE, where the abstractions manipulated by the end user are changes to a portion of a hypertext, as opposed to CoVer's composites that contain the entire hypertext being modified. While CoVer does persistently store the differences between two tasks in a separate task called a "change task" that is comparable to a PIE layer, CoVer does not provide any facilities, like PIE does, for arbitrarily composing several change tasks together so as to sum together the changes. Thus, despite its many assertions to the contrary, CoVer is much better characterized as a state-oriented composite-based versioning system than as a change-oriented one.

Scenario Overview

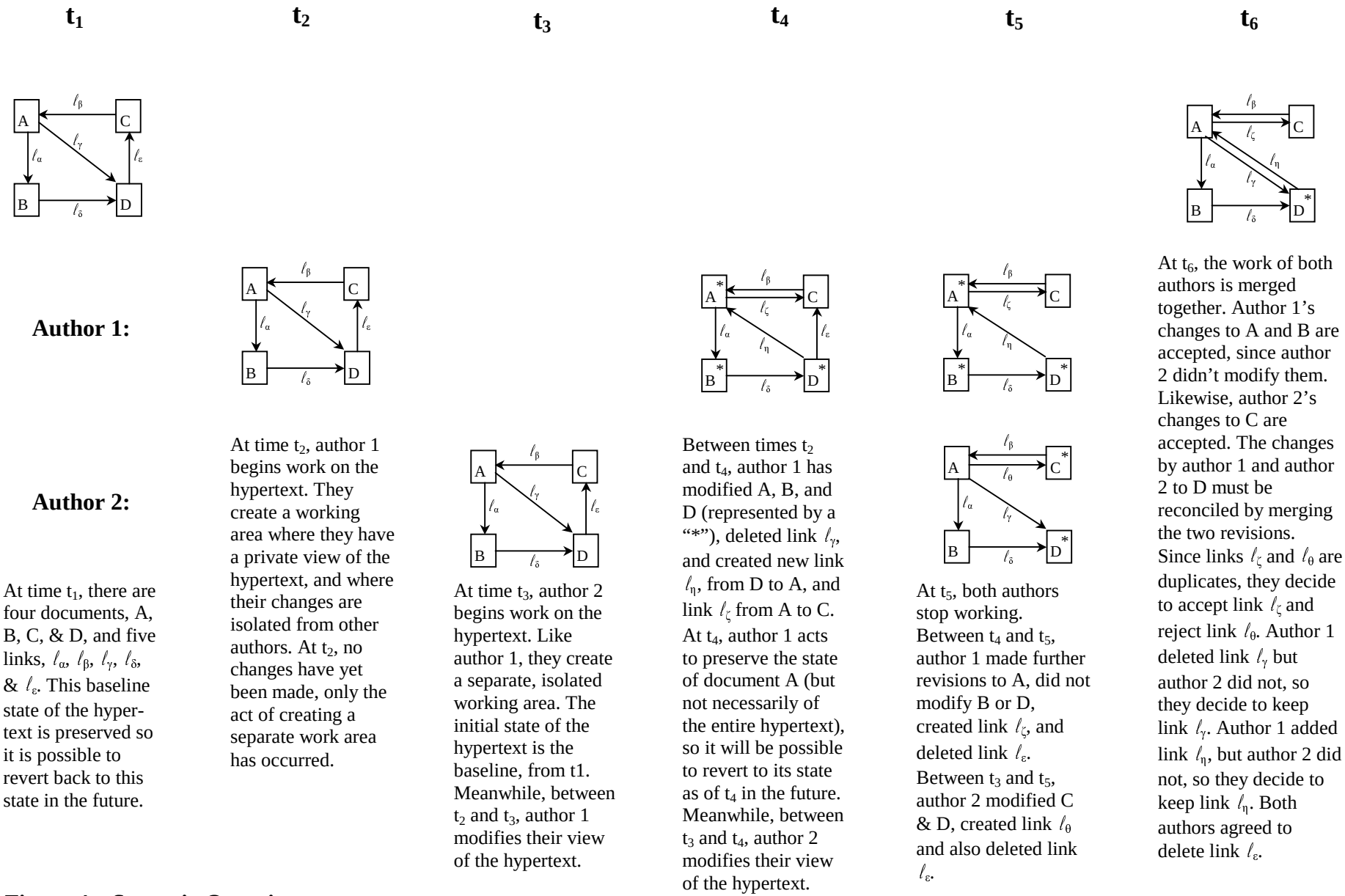


Figure 4 – Scenario Overview

Neptune

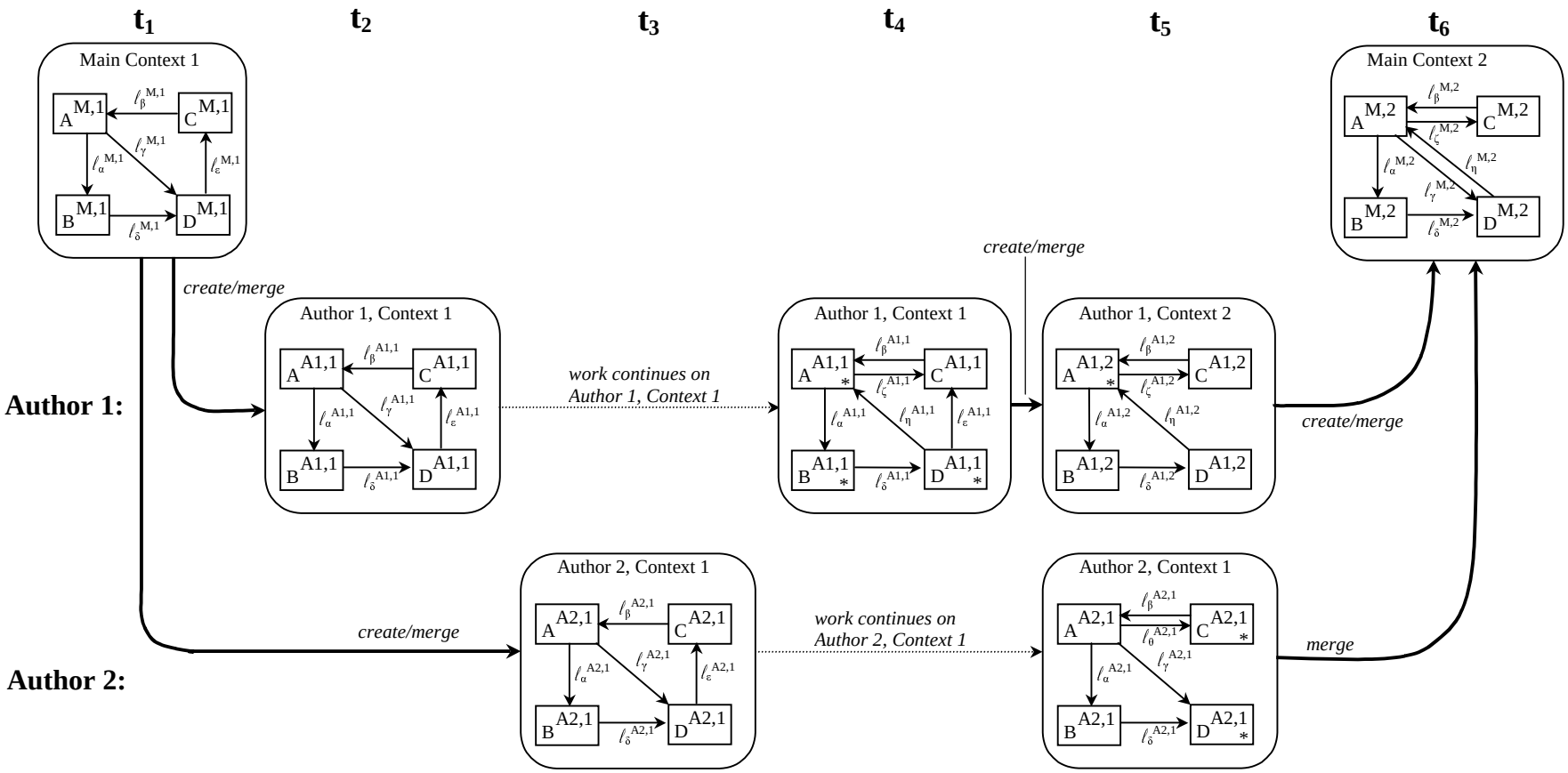
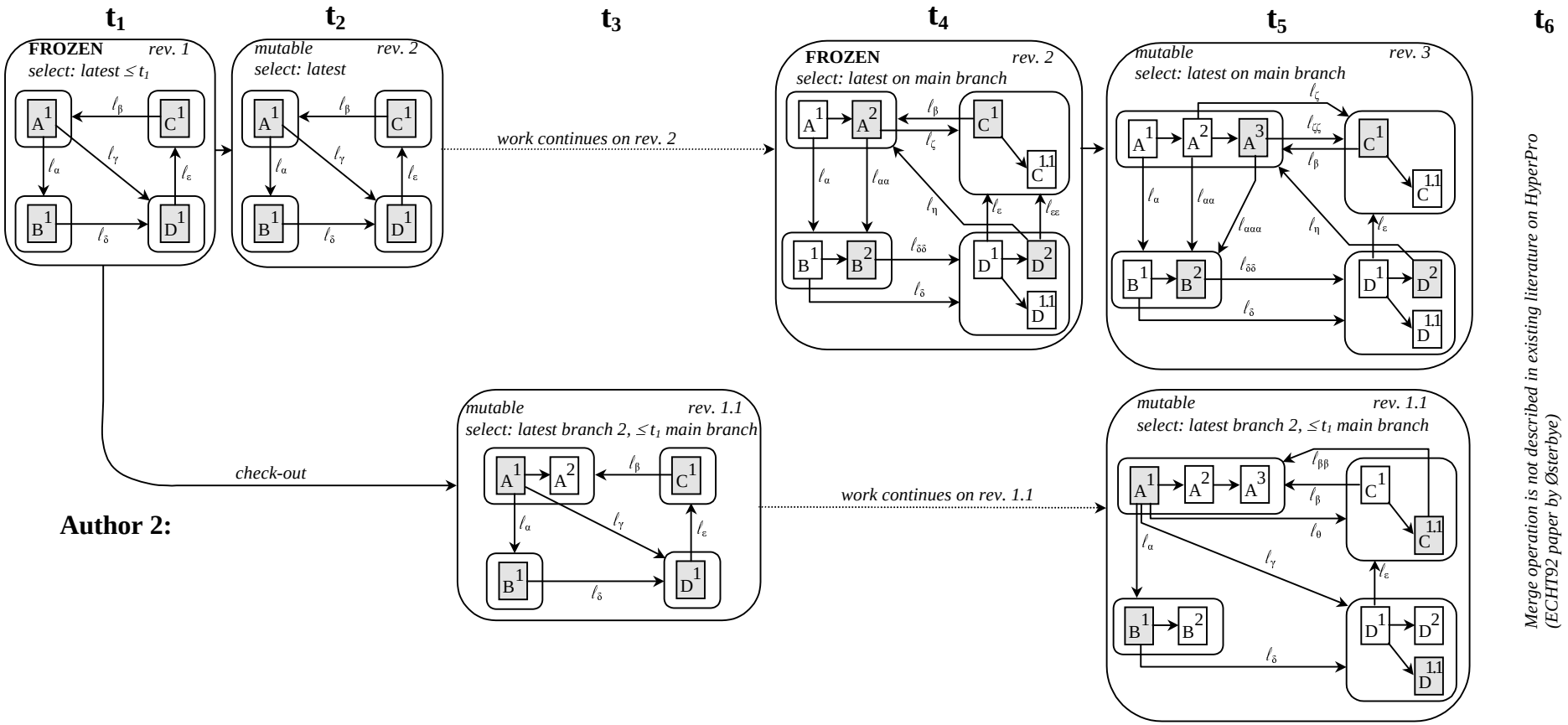


Figure 5 – The Scenario in Neptune

HyperPro



Version history of context at each time:

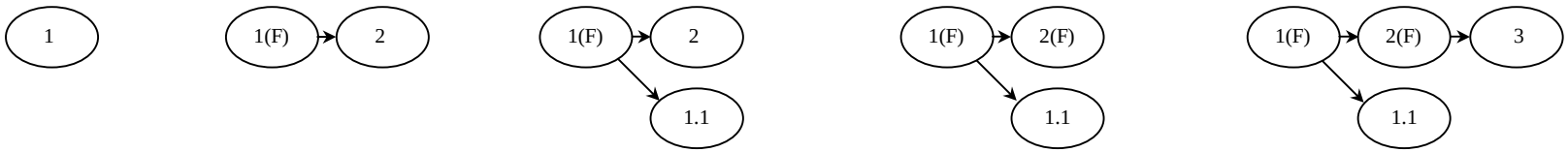


Figure 6 – The Scenario in HyperPro

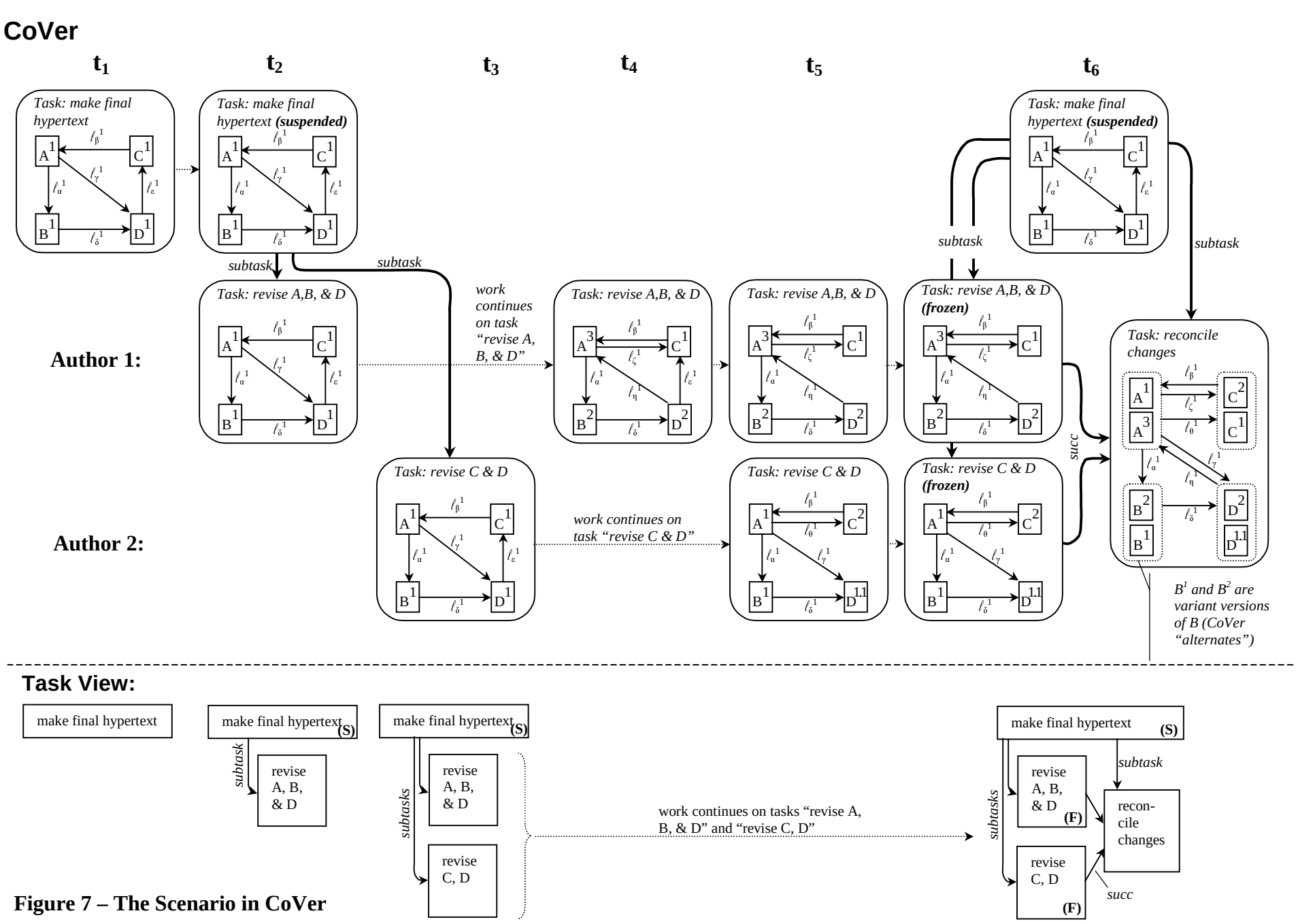


Figure 7 – The Scenario in CoVer

HyperProp / Nested Composite Model

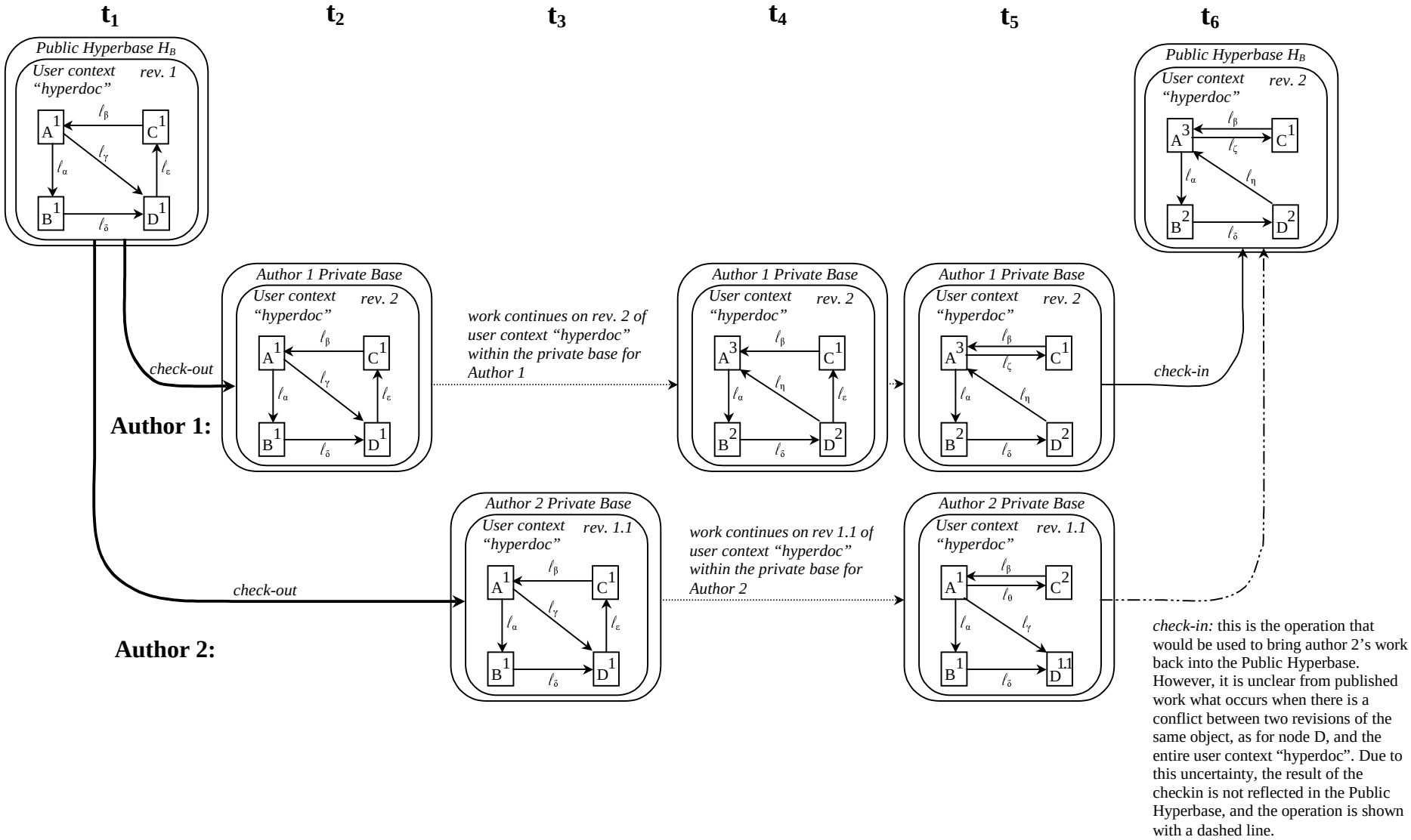


Figure 8 – The Scenario in HyperProp / Nested Composite Model

3.5.7 Summary of Composite Data Models

A summary of the containment characteristics and version selection capabilities of composite-based hypertext versioning systems is presented in Table 1 below. In Table 2, linking, versioning of composites, merging, and variant support is summarized.

System	Composites are called	Containable items	Containment type	Version selection
PIE [32]	layer, context	A *layer* contains a set of changes to nodes and links, a *context* is the sum of several layers.	Hybrid: nodes stored by value in layers, but layers stored by reference in contexts	none
Neptune [19, 18, 20]	context	nodes, links	by value	Supports links to the current version of a node, or to a specific version of a node. The Hypertext Abstract Machine (HAM) supports some time-based version selection.
HyperPro [63]	context, VersionGroup (both subclasses of composite type)	Contexts contain nodes, links, and VersionGroups. VersionGroups contain nodes, links, and contexts.	by reference	Specific version links select a specific revision of a node. Generic version links use time-based version selection to pick the newest version of a node (but not selecting a node newer than the time when the context was frozen).
CoVer [36, 38, 37]	composite, task, mob	nodes, links	by reference	Both links and containment relationships use an arbitrary query that is dynamically evaluated. If the query returns multiple versions, they are considered alternates with respect to the query.
VerSE [40]	composite, task (linear change task, parallel change task)	nodes, links	by reference	Both links and containment relationships use an arbitrary query that is dynamically evaluated. However, the query is required to only produce one result.
HyperProp / Nested Composite Model [74, 73]	composite node, private base, public hyperbase, version context, user context	content nodes, links, composite nodes	by reference	Version selection criteria for links are queries stored in an anchor (an attribute) on either a private base (private workspace) or a version context (or possibly both).
Hypermedia Version Control Framework [44]	Composite, association set	All system objects can be contained (i.e., components, anchors, links, associations).	by reference	Attributes on object revisions can be used by links and containment relationships to either statically refer to a single revision, or dynamically select a revision using a predicate operation over attribute values.

Table 1 - Containment and version selection in composite-based hypertext versioning systems.

System	Link definition	Links versioned?	Composites versioned?	Merging composites	Variant support
PIE [32]	Binary relationships are between whole nodes. Relationships do not support hypertext-style link traversal.	No	Layers and contexts are not individually versioned, but layers represent changes to a system, and hence store previous states.	Layers can be arbitrarily composed into contexts.	Yes, each layer can potentially represent a variant.
Neptune [19, 18, 20]	Binary links between nodes. Anchor information is contained within the link, and is a single position within the node.	No	Yes, though there is no system object that models a versioned composite. Operations to get parents and descendants of a context are supported in the HAM.	The merge context operation copies specified nodes and their attached links from the source context into the merge context. User must resolve duplicates in merge context.	Variants can be represented by having the same object developed in multiple, parallel contexts.
HyperPro [63]	Binary links between nodes. No anchors.	No	Contexts are versioned, VersionGroups are unversioned.	Unknown. Merge operation is not described in [63]	Variants can be represented by placing each variant on a different branch of a versioned node.
CoVer [36, 38, 37]	Binary unidirectional or bi-directional typed links between link anchor objects, which are associated to basic objects (node or composite).	Yes	Yes, though revisions of composites tend to appear as different tasks, rather than different revisions of the same composite.	Yes, a task can be made the successor of two or more other tasks. If there is a merge conflict in the same object, all revisions appear in the merged task as alternates.	Variants occur when a revision selection query returns more than one result. Causes multiple instances of the same object to appear within a task.
VerSE [40]	Binary unidirectional or bi-directional typed links between link anchor objects, which are associated to basic objects (node or composite).	Yes	Yes, though revisions of composites tend to appear as different tasks, rather than different revisions of the same composite.	Merging is supported for the parallel versioning style. It is unclear how merge conflicts are handled (some discussion appears in [39]).	Variants could be represented in a set of parallel change tasks. However, unlike CoVer, multiple variants cannot appear in the same task.

HyperProp / Nested Composite Model [74, 73]	Links are sets of endpoints contained in a context. Each endpoint is a perspective, an anchor that performs version selection to select a node, and an anchor that selects a region within the node	No	User contexts are versioned. The public hyperbase and private bases are not versioned.	Merging is supported with the shift operation that moves objects from a private base into the public hyperbase. It is unclear how merge conflicts are handled.	Variant contexts group together different renderings of the same objects (user context and content nodes).
Hypermedia Version Control Framework [44]	Anchors and links are computational processes capable of arbitrary actions. Associations represent structure, and contain a set of anchors and links.	Yes (links and associations are both versioned)	Yes. Versioned association sets represent versioned structure.	Merging is not discussed, and is considered an application-specific concern.	Variants can be represented by placing each variant on a different branch of a versioned object.

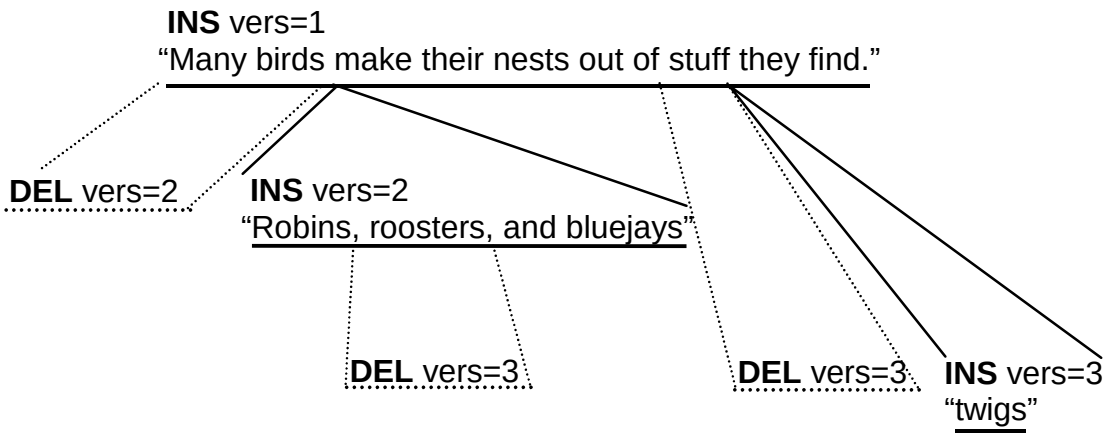
Table 2 - Linking, versioning of composites, merging, and variant support in composite-based hypertext versioning systems.

3.6 Within-node Versioning

Most hypertext versioning systems abstract away the details of exactly how changes between revisions are represented in their repository, generally using either a reverse-delta [76] or forward delta [70] encoding of the changes between revisions. As discussed in Section 3.1, Palimpsest [24] and VTML (Versioned Text Markup Language) [78] diverge from this pattern, choosing to focus almost exclusively on the representation of changes between revisions, for two primary reasons. First is to allow several collaborating authors simultaneous access to the resource, with all collaborator's changes reflected immediately. In this, they share the behavior of Grove [25] and Prep [62] that use per-operation update protocols. However, Palimpsest and VTML save the sequence of operations as a change history, while Grove and Prep do not. The second motivation is to preserve the integrity of link references, by recording exactly how anchored text regions move as a document is edited.

The basic data structure used to achieve these goals is a tree of fine-grain changes. An example of a VTML change tree is shown in Figure 9. A VTML change tree starts with the first text inserted into the document, which creates version 1. From this point forward, every inserted or deleted span of text is recorded, along with associated metadata. For every change, VTML always records its author, version, and date, and may also record a comment, and a tag indicating the importance of the change (to avoid figure clutter, only the version is shown in Figure 9). In many respects, this is similar to the change tracking functionality provided in recent versions of Microsoft Word (developed independently of the earlier work on Palimpsest, which directly led to VTML).

VTML can represent branches, and alternate versions of a document, and thus can represent all the revisions of an item within a version history tree. It also provides support for merging two or more revisions. Merging is performed by accepting or rejecting each modification in the merged revisions, with one predecessor revision chosen as the parent whose changes are accepted by default.



Version 1: Many birds make their nests out of stuff they find.
Version 2: Robins, roosters, and bluejays make their nests out of stuff they find.
Version 3: Robins and bluejays make their nests out of twigs they find.

Figure 9 – An example VTML modification tree

VTML change trees can be linearized into plain text, a useful property for storing VTML documents in unstructured text files, or for sending VTML via email, or other network protocol. As an example, the change tree from Figure 9 is shown below, with bolding added to highlight the text as separate from the VTML markup. Note that VTML does contain a facility, called attribute lists, which reduce the repetition in attribute values visible below.

```
<!--{INS vers=1 author="Jim" date="Nov. 24, 1999"}-->
<!--{DEL vers=2 author="Fabio" date="Nov. 27, 1999"}-->
Many birds<!--{/DEL}--><!--{INS vers=2 author="Fabio" date="Nov.
27, 1999"}-->Robins<!--{DEL vers=3 author="David" date="Dec. 1,
1999"}-->, roosters,<!--{/DEL}--> and bluejays<!--{/INS}-->
make their nests out of<!--{DEL vers=3 author="David" date="Dec.
1, 1999"}-->stuff<!--{/DEL}--><!--{INS vers=3 author="David"
date="Dec. 1, 1999"}-->twigs<!--{/INS}-->they find.<!--{/INS}-->
```

While VTML is descended from earlier Palimpsest work, post-VTML Palimsest was enhanced in the process of developing David Durand's dissertation [24]. This later Palimpsest work took on the task of representing move and copy operations. This added significant difficulty because it increased the number and type of operation conflicts that can occur. There are several types of possible conflicts:

- **Point/point:** This occurs when two or more text insertion operations (insert, copy, move) have the same insertion point. Conflict resolution concerns ordering, determining which insertion is displayed first, second, etc.
- **Point/range:** This results when the point of an insertion operation is within the range of another operation (the region of a delete, or the source of a copy or move). Conflict resolution depends on the semantics of the range operation. Inserts into a deleted region are easy to resolve. Less simple are inserts into the source of a copy—should the inserted text appear at the destination of all copies (making the copy dynamic), or should it only affect the source region?
- **Range/range:** This results when there is an overlap of two ranges, such as deleting part of a copied region. As for point/range conflicts, range/range conflict resolution depends on the semantics of each operation, one difficult case being moving part of the source range of a copy.
- **Global:** These conflicts "involve cyclic relationships between the regions of effect of a set of changes, so that a conflict exists only when the entire set of changes is considered, but the removal of even one of the constituent changes would eliminate the conflict, by breaking the cycle" [24], p.

66. Two simple cases are a copy operation where the destination point of the copy lies within its source range, and a similar move, where the destination lies within the source region.

Developing policies for resolving these conflicts is a key challenge for applications that use Palimpsest.

Bearing a surface similarity to Palimpsest and VTML is the work on persistent data structures [23]. Both have as their goal maintaining multiple revisions of the state of a data structure: in VTML and Palimpsest, the data structure is a text string; in [23] it is a tree or graph. However, since a text string could be represented as a tree or graph, it seems feasible that persistent data structure techniques could be used to represent multiple revisions of a text. A key difference is that persistent data structures only capture the successive states of the versioned data structure, but do not capture the operations used to make the changes, as do VTML and Palimpsest. The difference between state and change orientation appears again.

4 Goals of Hypertext Versioning

A characteristic of hypertext versioning research is the lack of agreement on the facilities and goals to be provided by hypertext versioning systems. Research spans a range from the Palimpsest and VTML focus on document format, to systems like DIF, HyperWeb, Hyperform, KMS, and Delta-V that version only nodes, to context-oriented systems like Neptune, CoVer, HyperPro, and the HURL framework that version both nodes and link structure. These differences in capabilities embody differences in the system's underlying goals. To better understand the design space inhabited by this research, this section has collected and organized the explicit and implicit goals described in the hypertext versioning literature.

The papers that initially listed these goals almost always call them requirements, and for a particular system, it is reasonable to call them such since the system actually implements all of the listed features. However, after collecting all of these requirements together, it is clear that no single system will implement all of the individual requirements from several systems. Therefore, this section lists them as goals, and is not concerned that some goals may be in conflict with others. An individual system must resolve these tensions, while this survey merely catalogs them.

4.1 Data Versioning

4.1.1 The history of nodes must be persistently stored

This is a standard facility of all hypertext versioning systems, and differentiates systems that contain versioning features from those that do not. In essence, this requirement introduces time into the system, adding an extra dimension to nodes, composites, and user interfaces. Though a standard goal for all hypertext versioning systems, this requirement is explicitly noted in [74, 68, 15, 20, 42, 58, 61], and for the Web in [5, 81]. The literature lists many reasons for wanting to record the history of a node, including:

- **Exploration:** allowing for the exploration of the history of evolution of a particular hyperdocument. [15] (Section 10.1)
- **Comparison:** comparing two or more revisions to see what has changed between them [61], p. 2/25.
- **Backtracking:** making it possible to see any version of a hyperdocument back to its beginning. [20], p. 170. The motivation for backtracking is to maintain previous revisions in case mistakes or wrong decisions need to be undone, to reconsider old choices, or otherwise look at former states. [61], p. 2/13. [58] notes that auditability is important for some industries, that is, being able to recover, for legal purposes, prior states of a hyperdocument.
- **Safety:** assuring the safety of recent work against various kinds of accident. [61], p. 2/13.
- **Rationale capture:** since the reason for making a particular change soon fades from memory, versioning systems should allow a brief comment to be associated with each change to capture this rationale. Over time, these comments create a group memory for the node [81], p. 197.
- **Reuse:** by preserving a specific revision of a hyperdocument, the entire document, or parts of it, may be reused by others. [36], p. 44.

- **Exploratory development:** since authors can depend on the ability to revert to a known, "safe" state of the system, versioning supports exploratory changes, where the final impact is initially unknown [74], §3.1, [63], p. 37.

[74], quoting [42], writes:

"Maintenance of Document History – 'A good versioning mechanism will allow users to maintain and manipulate a history of changes to their network.'" However, it is clear that this combines together the notion of versioning a node and versioning structure. [68] also lists these two requirements together, stating that it is a requirement to "support multiple versions of nodes and multiple configurations (i.e., different link arrangements)" (p. 90), though his equating of the term "configuration" with link arrangements differs from current practice in the software configuration management community. However, [45] notes that configurations are the SCM analog to hypertext structure, since a configuration represents the structure of a software system. Perhaps the terms configuration and structure are not so far apart after all.

4.1.2 Immutable and mutable node revisions, and node metadata, must be supported

At its core, this requirement is concerned with just how much the past can be changed. As opposed to the true past, which can never be changed, the past in versioning systems is stored persistently by the computer on some writeable medium, and can be modified. The literature notes several reasons for wanting to change stored revisions:

- **Linking:** If anchors are stored within a node, and linking to/from frozen nodes is allowed, then some mechanism for temporarily making a node mutable is required to support linking. [63], p. 35.
- **Modifying system-specific metadata:** Since system-specific information, such as access control lists, are often stored as metadata, it is useful to be able to modify this metadata after its state has been frozen. However, some metadata specifically depends on the object's value (e.g., an attribute listing the size of the object), and should only be modified if the value is too. It might also be attractive to add new attributes to a node. [63], p. 35, [81], p. 196.
- **Making a small change:** If there is a minor, non-semantic change to a human-readable document, such as a spelling error, it may be reasonable to allow the error to be fixed rather than creating a new revision. [81], p. 193.
- **Preserving logical revision names:** In some cases it is important to maintain the logical name of a revision, especially in cases where some external authority controls revision names. For example, the Internet Engineering Task Force assigns revision identifiers to working drafts. If a spelling error is found in one of these drafts, it is preferable to fix the document without creating a new revision, since that would imply a new official revision had been made. [81], p. 193.

The literature is far from unanimous on this capability. Though [63] and [81] agree that some node metadata must remain mutable, while other metadata is frozen with the node contents, they differ on whether the node contents can be immutable. [63] states, "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." (p. 35). However, [81] has requirements stating both that, "some properties on revisions may be changed without creating a new revision" (p. 196), and that, "revisions may be mutable or immutable." (p. 192).

4.1.3 Versioned and non-versioned documents can coexist

In a hypertext, it is possible that a user may desire to version some, but not all of the data items. In this case, versioned and non-versioned items coexist in the same hypertext [36], p 44. Allowing this increases system complexity, since containers, (either composites, or directories in a hierarchically organized namespace), now have to handle the issue of whether a versioned container should record the membership of an unversioned item when its state is frozen. If it does record the membership, it is possible that, in the future, the unversioned items will be deleted, and the prior container state will now contain dangling membership links. If, on the other hand, the unversioned items are not recorded as part of the container's

state when it is frozen, containers are now complicated with two kinds of membership, versioned and unversioned. Furthermore, when a versioned container is reverted to a prior state, the system is faced with the choice of either removing all unversioned items from the container, or preserving the unversioned items and only reverting the versioned items, making it so that the exact prior state of a container can never be recovered. Due to these complexities, hypertext versioning systems that version containers adopt an all or nothing approach, requiring that all of a versioned container's items be versioned.

4.1.4 All content types must be versionable

Since hypertext systems allow for browsing, editing, and linking between all types of nodes, including text, images, movies, sound, etc., a hypertext versioning service must be capable of versioning these disparate information types [81], p 191. Furthermore, since many document and image types undergo constant evolution (e.g., the Word document type, or HTML), remaining independent of content type ensures the versioning facilities will be able to accommodate this evolution. In contrast, supporting only text versioning, or tailoring versioning facilities too tightly to one, or a small set of content types, would increase the brittleness, and reduce the applicability of the versioning support.

4.1.5 A mechanism must exist for giving a human readable name to a single revision.

Frequently, a specific revision of a versioned item has significance beyond just being an intermediate state of an evolving object. A revision may have been released to a customer, or submitted for external review. For these revisions, it is useful to assign a human readable name to that revision, such as "customer release" or "external review" [81], p. 198, [58], p. 21. These human readable names are often called *labels* in configuration management systems, and can be used in revision selection rules.

4.2 Stability of References

As noted in the section, "Versioning for Reference Permanence," a major goal for hypertext versioning is to preserve prior node states to ensure that links never dangle, a goal explicitly noted in [61], p. 2/25, [78]. The underlying assumption is the linked node state will always be available, since it is persistently stored, and hence the link endpoints will, likewise, always be present. However, both [61] and [78] assume that the owner of the information is responsible for storing, and incurring the costs of maintaining prior states. However, there are many valid reasons for the owner of a document to permanently remove it, and all its prior states. For example, in corporate settings, mergers and acquisitions can make a company name obsolete, product lines can be modified or terminated, in both cases making it desirable for the owner to remove all documents that reference the old company or product names. As a result, reference stability can only be achieved by incorporating third party versioned document stores, where the third party has no compunction about keeping around older information. That is, the implication of a third party caching old company or product names is different from the owner preserving this state. For the third party, no endorsement of the older states is implied, but for the owner of the information, the mere fact of making older information available increases its perceived value.

4.3 Change Aggregation Support

It must be possible to group a set of changes that together comprise a coordinated, logical change, a goal noted in [36], p. 45, [68], p. 91, [74], §3.1, and [81], p. 197. In state-based versioning systems, the set of revisions that together comprise a logical change is called an *activity* [81], a *task* [36], or a *version set* [74]. In change-based versioning systems, sets of changes are, appropriately enough, called *change sets*, while PIE [32] calls them *layers*. There are several reasons for aggregating changes:

- **Labeling a set of changes:** The ability to give a name to a set of changes [81], p. 197.
- **Tracing changes to revisions:** Since a logical change can span multiple versioned items, long after a change was made it can be difficult to reconstruct which revisions implement a change. By recording which revisions comprise a change, it is possible to trace a change to its revisions, and vice-versa [74], §3.1.

- **Combining version sets:** In change-based systems, changes are recorded so each logical change can be managed as a single entity. Changes can then be combined to create new states of the hyperdocument [68], p. 91.

4.4 Link and Structure Versioning

It may seem strange to separate versioning of links from versioning of structure. After all, one would expect that versioning of structure would be an emergent property of a set of versioned links. Because HyperPro [63] has demonstrated that it is possible to version structure without versioning links by placing the unversioned links inside a versioned composite, and since Hicks [44] goes to great lengths to separate structure versioning from node versioning, this paper intentionally separates the goal of versioning links from the goal of versioning structure.

The reasons for versioning links and structure are the same as for versioning nodes. That is, links and structure are versioned to support revision history exploration, backtracking, safety, rationale capture, reuse, and exploratory development.

4.4.1 It must be possible to version links.

Since links often have state, such as their endpoints, or additional metadata, it is possible for the value of a link to change over time, and hence it is desirable to record important points of its evolution. In particular, when a node is frozen, it is desirable to permanently record link endpoints so that when the node is reverted to this state at some point in the future, the links at that time will be available in their original form [20], p. 170.

4.4.2 It must be possible to version structure.

Several hypertext systems, including Microcosm [16], Chimera [2] and the Hypermedia Version Control Framework [44], allow multiple sets of links, or structures, to be applied to the same set of nodes. Due to the ability to apply multiple structures the same set of nodes, and the separation of versioning responsibility between node and link versioning in link server architectures, it is desirable to version structure independent of versioning nodes. The versioning proposal presented in [82] shows links versioned separately from data. Meeting this goal is very challenging, as Hicks notes when he states, “The requirement to version structure is one of the main challenges which hypertext brings to the version control area.” [45], p. 13.

When nodes and links are inextricably combined into composites, as is the case in Neptune [20], HyperPro [63], HyperProp/NCM [73], CoVer [36], and VerSE [40], it is still desirable to version structure, although it is no longer possible to separate versioning of structure from versioning of data.

4.4.3 It must be possible to link to a specific revision of a node.

Linking to specific node revisions makes it possible to construct documents that always link to a specific revision. For example, if a document is describing the evolution of a document, it is desirable to have links to the revisions in the text that describes each revision. This is a standard facility of hypertext versioning systems, and hence is not typically mentioned, although it is explicitly noted as a goal in [81], p. 196.

4.5 Variant Support

There are many types of node variants, including alternate revisions that were developed during parallel collaborative work, translations into different human languages, and renditions of the content as different content types (e.g., PDF and Postscript) [61, 74]. Structural variants are also possible, representing alternate structures for the same set of content [74]. A hypertext system should handle these different kinds of variants, by providing a data model and operations that can represent and manipulate variants.

Hypertext versioning systems also support literary use, representing multiple variants of the same work over a period of time. Machan writes, “hypertext enables editors to assemble in one edition all the versions of a given literary work and readers to access these versions, or parts thereof, in any number of ways.” [56], p. 303. Ideally, the facilities used to represent other kinds of variants will have the expressive power to model alternate versions of complex documents, such as the Canterbury Tales discussed by Machan.

Hypertext systems should also be capable of allowing authors to create new alternate versions, not just represent existing ones, even when this involves a change in ownership, authorial control, or media [61], p. 2/39. If an item does get developed into an alternate version, or is reused in a different context, it should be possible to trace this use, either from the new context to the original, or vice-versa [36], p. 45.

4.6 Collaboration Support

As discussed in Section 3.5.5, versioning makes it possible for two or more people to work on the same document or link simultaneously, by storing each person's work in a separate revision. As a result, versioning systems should support collaborative work by providing independent work partitions that allow concurrent authoring of the same information [74], while preventing one author from interfering in the work of other collaborators [20], p. 174-175. The ability to work in isolated work areas implies a need to merge together several changes once parallel work has ceased [39, 68], p. 91.

[39] notes two additional goals specifically related to merging hypertext networks:

- It should be possible to select a merge procedure, based on the hypertext data model and group work situation.
- Interactive merge tools will be required, since, in general, it will not be possible to automatically resolve merge conflicts. These tools should provide the ability to specify merge results interactively.

In addition to isolating changes, versioning supports collaboration by acting as an awareness mechanism. For example, [38] notes that in the absence of versioning, "… co-authors rejoining work e.g. after holidays found it difficult to find out what happened in the meantime." (p. 408). Basic versioning facilities also support awareness. Comparing revisions provides awareness of a collaborator's modifications. Examining an item's history shows who has worked on it, and hence who can answer questions about it, while stored comments provide awareness of change rationale.

4.7 Navigation in the Versioned Space

A hypertext versioning system should provide the ability to perform hypertext navigation both within a consistent time slice (i.e., across multiple nodes at the same time), as well as forward and backward in time [61], p. 2/26. Since link traversals across time are possible, a hypertext versioning system should provide information to the user so he knows what time each visible node represents, and what time lies at the destination of a link traversal. It should also be possible to select between navigation in a consistent time slice, and navigation always to the present revision. Generalizing, it would be desirable to be able to have the user select the version selection criteria used in link traversals. For example, navigation based on non-time based criteria should also be possible, such as queries on attribute values [36], p. 45.

4.8 Visualizing the Versioned Space

Nelson captured one facet of this goal when he wrote, "while the user of a customary editing or word processing system may scroll through an individual document, the user of {Xanadu} may scroll in time as well as space, watching the changes in a given passage as the system enacts its successive modifications." [61], p. 2/18. Unfortunately, this goal of dynamic, animated difference visualization has not been implemented by any hypertext versioning system to date. A more achievable goal is to visually display the differences between two textual revisions, a capability provided by Neptune [20] and CoVer [36], and strongly advocated for Xanadu [61], p. 2/20. Since hyperdocuments often contain non-textual elements, such as graphics, video, etc., techniques for visualizing differences between multiple revisions should also be provided [39]. Similarly, comparing the differences between two variants or alternate versions should also be possible.

A hypertext versioning system should also provide visualizations of a versioned item, and of a versioned hypertext. Many configuration management systems have created graphical visualizations of versioned items, and within the hypertext versioning literature, CoVer [36] and VerSE [40] provide such a visualization, though, unfortunately, they reverse convention by having arrows pointing to predecessors, rather than successors. Thus, in their visualization, it is easy to mistake the youngest revision for the oldest,

and vice-versa. Furthermore, the visualization of a versioned item should display incoming and outgoing links, along with their revision selection criteria.

The task view in CoVer is a significant innovation, since it provides a visualization of just composite evolution, abstracting away their contents. As Figure 5Figure 8 highlight, showing both the composite and its contents results in space intensive, busy displays. Providing a visualization of only a composite as it evolves should be a goal of composite based hypertext versioning systems.

Merging together hypertext networks raises more needs for visualization. When merging hypertext networks, it is necessary to merge the hypertext structure in addition to the more pedestrian problem of merging nodes. In order to merge hypertext structure, a hypertext versioning system must provide a visualization of the difference between the two networks being merged, such as the Graph-Unification-Merger and Graph-Comparison-Merger proposals in [39].

4.9 Traceability

Versioning supports reuse of document parts in other documents. A hypertext versioning system should provide the ability to trace the reuse of the material from an original source to derivative works, and vice-versa, even when it spans document boundaries [36], p. 45. This addresses the problem of losing the object identity, identified in [38], that occurs when an object is copied into a new composite.

[36] identifies a different kind of traceability goal – the ability to trace a change request described in a document annotation to the modification that fulfills the request, thereby using annotations as a change request management (bug tracking) system. Instead of entering the change request into a separate system, the document (code) is annotated directly where the change needs to be made, providing the advantage that the change request is presented along with its context.

4.10 Goals for User Interaction

Most authors can cognitively handle the concept of multiple revisions of individual documents. However, since hypertext is a new phenomenon, people who create hypertext do not have years of authoring experience to draw upon when mentally visualizing a hypertext network as it changes over time. Furthermore, due to the presence of links, hypertexts are just plain more complex than individual documents, and having multiple states of the hypertext over time piles on more. As a result, many sources have stressed the importance of reducing the user-visible complexity of hypertext versioning systems [36, 63, 68, 82]. Østerbye is the most strident on this point, identifying naming of new nodes and links (echoing Conklin [10]), and version selection as two significant sources of cognitive overhead, addressed in HyperPro by the use of contexts [63]. Another source of complexity occurs when authors have to decide explicitly for every update whether it should result in a new revision [36].

The perceived complexity of a hypertext versioning system is dependent on its user interface, and its visualization of the versioned hypertext structure, and the operations used to manipulate it. To date, there has been little published research on the user interface of hypertext versioning systems, with CoVer [36] and VerSE [40] providing the only examples of published articles containing screen shots. Since perceived user complexity is such an important issue for the adoption of hypertext versioning systems, additional research that focuses on just the user interface aspects would be very valuable.

User interface research could address an open question: is hypertext versioning technology simple enough that it could one day be used in mass market software systems, achieving ubiquity similar to that of word processors today? Or is it the case that hypertext versioning will only be of interest to a highly skilled user base that is willing to invest significant time in learning the concepts and operation of the technology, because their problems are too big, or the pain of not versioning is too high, or there is some compelling regulatory or business driver. At present, there is insufficient data to settle the question, since no system that supports versioning of structure and data has ever been used outside of a laboratory setting. Some parallels can be drawn with configuration management systems, which address similar problems, and are in wide use today. Configuration management systems provide work isolation, merging, problem tracking, and the ability to revert to prior revisions, as do the composite-based hypertext versioning systems, so in many respects the systems are of equal complexity. While configuration management systems do not explicitly version link structures, and thus are less complex, they do provide the ability to construct

arbitrary configurations, a distinct increase in complexity over hypertext versioning systems that do not have this feature. This argues that there is a niche for hypertext versioning at least among highly skilled users.

But, what about less skilled use, such as in the home, or casual office use? The recurring pattern of document processing tools only providing linear versioning implies a tradeoff between data model complexity and breadth of users. Schedule pressure can also apply pressure to simplify even skilled use of configuration management technology [79]. This implies a need for hypertext versioning systems to provide functionality layers, with a simple layer providing a minimal set of high value operations, and one or more layers giving more complete, and hence complex functionality. Such a layered strategy is used by the DeltaV protocol [81].

De-emphasizing user naming of items carries with it some additional tradeoffs. When the system is primarily responsible for naming, the entire issue of item naming tends to be minimized. Yet, as the Web has ably demonstrated, object identifiers, such as the URLs commonly found in all manner of advertising, are not internal to a system. Item names have an important role in creating namespaces that are shared by other users of the system. As hypertext systems become more distributed, naming issues become more important, and, likewise, the ability of the system to completely hide naming issues from the user is reduced. Instead of trying to completely remove user control over naming, a better goal is to assist with naming, suggesting names that authors can change at will.

Difficult naming issues are raised when versioning data or structure. The first issue is which items should have distinct names. Certainly each revision should be individually referenceable, but typically it is the versioned item which is named, and each revision has a name that combines the versioned item name with a revision identifier, as in “report,v5”. Containment adds only difficulty. When containers are named and versioned, one mechanism for naming their contents is to list the container’s name and revision, followed by the contained item, as in “folder,v2/report,v5”. As it happens, URLs are syntactically incapable of associating revision identifiers with interior path segments, thus highlighting the need to design versioning support into names from the beginning, instead of retrofitting them later.

With nested containment, names with explicitly listed revision identifiers quickly become cumbersome, and brittle in the face of change. More durable names will contain version selection criteria, as in “folder/report;(selection rule=‘latest as of Nov. 5, 1996’)”. This immediately implies that names are no longer unique, as multiple revision selection rules can identify the same item. Furthermore, it is a very slippery slope from simple revision selection rules to full query language support, and the attendant challenge of encoding that query into a human-readable name.

4.11 Goals for Tool Interaction

For VerSE [40], the Hypermedia Version Control Framework [44], the Delta-V protocol [81], Hyperform [84], and the Chimera versioning proposal [82], a major goal is to provide a versioning-aware functionality layer that can be employed by multiple tools. However, these tools must either be coded from scratch to use the version-aware hypertext infrastructure, or existing third-party applications must be integrated, a well-known issue for open hypermedia systems [17, 80]. The Achilles heel of infrastructure development is that, in order to be relevant, the infrastructure must be used. This certainly applies to version-aware hypermedia infrastructures, which are useless unless applications employ their functionality. To address this concern, the goals for tool interaction espoused by these systems address how to make the infrastructure as attractive as possible so a tool integrator will mate their tool with the system.

Since application integration is often time-consuming, it creates non-trivial barrier preventing integration. Not surprisingly, reducing the infrastructure’s demands on third-party applications [82], and aiming to maximize the cost/benefit ratio of integrations [40] p. 226, are two goals explicitly named in the literature. Furthermore, since the versioning aware infrastructures extend pre-existing *unaware* systems and their respective integrated tools, it is a goal to ensure that pre-existing integrations must still work once versioning services are added, so as not to destroy the investment in the existing tool integrations [44, 81]. The version-aware infrastructure must maintain downward compatibility, and hence versioning aware and unaware applications must be able to interoperate.

However, reducing the barrier to entry for integrations doesn't address the issue of whether the infrastructure services are useful for a particular application. A common goal is that the versioning services should be applicable to as wide a range of tools, application domains, and versioning styles as possible [44, 40]. That is, ideally it should be impossible to reject a versioning-aware infrastructure due to lack of functionality, or a mismatch in applicability. Hicks, in [44], identifies three specific qualities to meet this goal:

- **Flexibility:** "to support a variety of different development methodologies, version control services should be flexible, unconstrained by any specific development paradigm." [44], p. 129.
- **Extensibility:** since the complete range of third-party applications is unknown in advance, the version-aware infrastructure should be able to accommodate new types of applications.
- **Scalability:** since applications will vary in their use of versioning services, the infrastructure should be capable of handling both heavy, and light users.

Several practical goals emerge once the Internet is the communications layer used to access the versioning infrastructure. Authentication is a concern, so that the versioning services can trust the accessing application, and vice-versa. Since it is desirable to remotely collaborate with individuals who should have access to a particular hyperdocument, but who should not have login privileges on a remote system, these should be decoupled [69]. That is, there is a need to give external collaborators write access, without giving them all the privileges of local users. Since, on the Internet, documents are being sent across a public network, if the contents of these documents must be kept private, some mechanism for encrypting their transmission must be used [81]. Furthermore, the Internet raises the problem that people from multiple countries, speaking a multitude of natural languages will be interacting with the versioning infrastructure. In this case, human-readable fields, such as attribute values, version labels, etc., will require internationalization [81]. Finally, since versioning requires the use of time, it is necessary to ensure that the clocks on interacting tools are synchronized [22].

4.12 Goals for Interactions with an External Repository

Most hypertext versioning systems provide versioning services internal to the system. The system itself contains the code used to store item revisions, retrieve older revisions, store version histories, etc. However, this duplicates the functionality available in existing versioning and configuration management systems. Furthermore, there are some use environments, such as existing software development projects, where all data is already under the control of an existing repository. In these situations, it is unlikely that the benefits offered by hypertext will overcome the cost of switching to a new repository, especially when, as is the case with current hypertext versioning systems, they do not offer the same functionality. In this case, in order to provide hypertext versioning functionality, the existing versioned repository must be used [13], p. 27. This scenario motivates the first goal in [82], which states that "objects stored external to the hypertext system and hypertext structures stored internal to the hypertext system must be capable of independent development." (p. 46). This raises the problem of synchronizing the versioned items in the external repository with the versioned structure within the link server.

4.13 Interactions with the Node Namespace

Hypertext versioning systems differ in whether the nodes being versioned are named, in addition to having some form of object identifier. For example, in the Chimera versioning proposal [82] nodes are files, and hence have filenames, while in the Delta-V protocol [81], Simonson and Berleant et alli [72], and Pettengill and Arango [67], nodes are named with URLs. In the context of the Web, the Delta-V protocol has identified several goals for adding versioning information to the URL namespace:

- **Each revision has its own URL.** This ensures that hypertext links can be made to every revision.
- **Relative URLs should not be disrupted.** If all of the revisions of an item are placed within a subcollection, for example, if all the revisions of "index.html" are placed within a collection with the same name, and hence the first revision would be "index.html/1", then relative URLs within this resource will no longer refer to the correct destination.

- **Requests on the URL for a versioned item should return a default revision.** If there is a URL for a versioned item, then read, write, and other requests on that resource should, via some revision selection rule, be redirected to a specific revision.

4.14 Missing Goals

One goal that is not present in any of the surveyed systems is the ability to revert just a single node within a hypertext, having all the links adjust. The ability to choose a pick a specific individual revision of an object independent of the other objects in a container is a common feature of software configuration management systems. Hypertext versioning systems require that reverting to a previous revision of a single object involves reverting to a previous revision a composite that contains the desired revision. This is undoubtedly more simple, since it provides a reasonable answer to how best to revert the links that begin or end at a single reverted node. But, the prevalence of this operation in configuration management systems suggests that hypertext versioning support for software development will need to support it too.

5 Conclusions

By surveying all known hypertext versioning systems, this survey collected and organized existing information on the subject, as well as identifying gaps in the knowledge. This paper has organized hypertext versioning systems into five categories: versioning for reference permanence, systems that version data, but not structure, composite-based, web versioning, and open hypertext versioning. It has also carried out a deep examination of data models, capturing in one place the details of how these systems organize their versioned data. This survey provides the first systematic cross-comparison of composite-based systems, cutting through the terminological thicket, and collecting relevant data model information from scattered locations throughout the literature, to assess the similarities and differences between them. Finally, the survey collects together all known goals of hypertext versioning systems. Together, this taxonomy of systems, and survey of data models and goals, will allow future researchers to better understand and position their systems among existing hypertext versioning work.

Several gaps in our knowledge of hypertext versioning are evident from this survey. One such is that existing work has almost completely focused on data modeling issues, and has ignored the user interface concerns of readers. Surveyed systems that do provide a user interface, orient this interface towards authors or developers, and ignore user interface issues for readers. Awareness of time is a primary problem in the reader interface: when navigating a link, the user should know the time of the current node, and the destination node. If multiple link versions can be followed, or links to multiple variants are available, it is an open question how these link traversal options should best be depicted.

The reader interface is a critical issue for composite-based systems, as discussed in Section 4.10. A concern voiced by some is these systems are just too complex. Certainly their data models are very intricate, and presumably tricky to implement. Hence, it is reasonable to conclude that the internals of these systems are complex. However, many usable systems have complex implementations. Production-quality databases are extremely complex, yet are used in many applications. Perhaps more tellingly, configuration management systems share many of the same functionalities, such as versioned objects and private workspaces, and also have large user communities. So, the mere existence of complexity does not, by itself, indicate that composite-based systems are unusable. More research is necessary to develop a better understanding of what constitutes a good reading user interface and authoring user interface for these systems. Further, these user interfaces should be tested with people who are typical of expected users, rather than just by the system's developers.

Surveyed systems have tended to make similar assumptions about control choices. Almost invariably, if there is a client/server split, versioning functionality goes into the server, thus permitting the client to be relatively simple. Yet, with the continuing increase in computing power, a more scalable approach is to push functionality to the edges of distributed systems, rather than to the center. This runs counter to the open hypermedia view that integrating third party clients should be made as simple as possible, and therefore few demands should be placed on applications. In the systems surveyed, NUCM [77] and the NTT Labs. versioning proposal [65] are two intriguing glimpses of how versioning is possible when the client has greater responsibilities.

Another common control choice is having all data stored in a centralized repository, a near-universal trait of surveyed systems. Yet, one of the key lessons of the Internet is that decentralized architectures scale better. Future research is needed to determine how best to support Internet-scale versioning across the control boundaries between versioning stores. Two main approaches suggest themselves. First, if versioning semantics are placed into the client, and removed from the server (as in the NTT Labs. proposal [65]), it is much easier to have versioning structures than span multiple servers under different control. So, one desirable outcome of client-oriented versioning research is scalability. A second approach is to develop interoperability protocols between cooperating versioning stores, such as the Back-End-Back-End (BEBE) protocol in Xanadu [61]. In this way, groups of servers in different control domains can cooperate to provide versioning services. A third approach is suggested by the increasing size and low cost of mass storage: a hybrid approach where a large local cache of versioned data from the network augments version information that is available online, but relatively more expensive to access.

While this survey provides a comprehensive view of hypertext versioning systems, there is still fertile ground for future survey work. More work can be done to tease out the similarities and differences between how configuration management systems manage their dependencies, and how hypertext versioning systems manage structure. In particular, since the change-oriented PIE system [32] has had significant influence, it would be worthwhile to examine later change-oriented systems to see if they too can contribute to hypertext versioning. Engineering database systems [49] are frequently cited as related work in the hypertext versioning literature, yet despite this awareness have had little impact. It would be worthwhile to more closely compare their data models with those of hypertext versioning.

Acknowledgements

The author gratefully acknowledges the help and feedback of Jörg Haake, Kapser Østerbye, Rohit Khare, Peyman Oreizy, Luiz Fernando Soares, and Julia Gaudinski. Richard Taylor receives my thanks for his feedback and ideas on this survey, and also for his patience (I first muttered something about doing a hypertext versioning survey five years ago). Special thanks go to David Hicks and Anja Haake for their detailed feedback on the composite-based scenarios. I'd also like to thank David Durand and Fabio Vitali for our discussions and their encouragement on hypertext versioning over the years.

References

[1] R. M. Akscyn, D. L. McCracken, and E. A. Yoder, "KMS: A Distributed Hypermedia System for Managing Knowledge in Organizations," *Communications of the ACM*, vol. 31, no. 7 (1988), pp. 820-835.

[2] K. M. Anderson, "Integrating Open Hypermedia Systems with the World Wide Web," *Proc. Eighth ACM Conference on Hypertext (Hypertext'97)*, Southampton, UK, April 6-11, 1997, pp. 157-166.

[3] K. M. Anderson, R. N. Taylor, and E. J. Whitehead, Jr., "Chimera: Hypertext for Heterogeneous Software Environments," *Proc. 1994 European Conference on Hypermedia Technology (ECHT'94)*, Edinburgh, Scotland, Sept. 18-23, 1994, pp. 94-107.

[4] R. Bentley, T. Horstmann, and J. Trevor, "The World Wide Web as Enabling Technology for CSCW: The Case of BSCW," *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, vol. 6, no. 2-3 (1997), pp. 111-134.

[5] T. Berners-Lee, "Versioning, A Web page that is part of the original design notes for the WWW," (1990). Web page, accessed Nov. 15, 1999. http://web1.w3.org/DesignIssues/Versioning.html.

[6] T. Bray, J. Paoli, and C. M. Sperberg-McQueen, "Extensible Markup Language (XML)," World Wide Web Consortium Recommendation REC-xml.

[7] V. Bush, "As We May Think," *Atlantic Monthly*, July, 1945.

[8] M. A. Casanova, L. Tucherman, M. J. D. Lima, J. L. R. Netto, N. Rodriguez, and L. F. G. Soares, "The Nested Composite Model for Hyperdocuments," *Proc. Third ACM Conference on Hypertext (Hypertext'91)*, San Antonio, TX, Dec. 15-18, 1991, pp. 193-202.

[9] M. Colton, “Preserving the Web's Digital History,” *Brill's Content*, November, 1999, pp. 54-56.

[10] J. Conklin, “Hypertext: An Introduction and Survey,” *IEEE Computer*, vol. 20, no. 9 (1987), pp. 17-41.

[11] R. Conradi and B. Westfechtel, “Version Models for Software Configuration Management,” *ACM Computing Surveys*, vol. 30, no. 2 (1998), pp. 232-282.

[12] Continuus Software, “Continuus/CM Product Description,” (1999). Web page, accessed Nov., 1999, http://www.continuus.com/products/productsBB.html.

[13] M. L. Creech, D. F. Freeze, and M. L. Griss, “Using Hypertext In Selecting Reusable Software Components,” *Proc. Third ACM Conference on Hypertext (Hypertext'91)*, San Antonio, Texas, Dec. 15-18, 1991, pp. 25-38.

[14] S. Dart, “Concepts in Configuration Management Systems,” *Proc. Third Int'l Workshop on Software Configuration Management*, Trondheim, Norway, June 12-14, 1991, pp. 1-18.

[15] H. C. Davis, “Data Integrity Problems in an Open Hypermedia Link Service,” Dissertation. University of Southampton, Southampton, UK, 1995.

[16] H. C. Davis, W. Hall, I. Heath, G. Hill, and R. Wilkins, “Towards an Integrated Information Environment with Open Hypermedia Systems,” *Proc. Fourth ACM Conference on Hypertext (ECHT'92)*, Milano, Italy, Nov. 30-Dec. 4, 1992, pp. 181-190.

[17] H. C. Davis, S. Knight, and W. Hall, “Light Hypermedia Link Services: A Study of Third Party Application Integration,” *Proc. 1994 European Conference on Hypermedia Technology (ECHT'94)*, Edinburgh, Scotland, Sept. 18-23, 1994, pp. 41-50.

[18] N. Delisle and M. Schwartz, “Contexts -- A Partitioning Concept for Hypertext,” *Proc. 1986 Conference on Computer-Supported Cooperative Work (CSCW'86)*, Austin, TX, Dec. 3-5, 1986, pp. 147-152.

[19] N. Delisle and M. Schwartz, “Neptune: A Hypertext System for CAD Applications,” *Proc. Int'l Conference on the Management of Data (SIGMOD'86)*, Washington, DC, May 28-30, 1986, pp. 132-143.

[20] N. M. Delisle and M. D. Schwartz, “Contexts-A Partitioning Concept for Hypertext,” *ACM Transactions on Office Information Systems*, vol. 5, no. 2 (1987), pp. 168-186.

[21] S. DeRose, D. Orchard, and B. Trafford, “XML Linking Language (XLink),” World Wide Web Consortium Working Draft WD-xlink-19990726.

[22] S. Dreilinger, “CVS Version Control for Web Site Projects,” (1998). Web page, accessed Nov. 22, 1999. http://interactive.com/cvswebsites/.

[23] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, “Making Data Structures Persistent,” *Proc. Eighteenth Annual ACM Symposium on Theory of Computing (STOC)*, 1986, pp. 109-121.

[24] D. G. Durand, “Palimpsest: Change-Oriented Concurrency Control for the Support of Collaborative Applications,” Dissertation. Boston University, Boston, MA, 1999.

[25] C. A. Ellis and S. J. Gibbs, “Concurrency Control in Groupware Systems,” *Proc. ACM SIGMOD'89 Conference on the Management of Data*, Seattle, WA, May 2-4, 1989.

[26] S. I. Feldman, “Make - A Program for Maintaining Computer Programs,” *Software-Practice and Experience*, vol. 9, no. 4 (1979), pp. 255-265.

[27] R. Fielding, J. Gettys, J. Mogul, H. F. Nielsen, L. Masinter, P. Leach, and T. Berners-Lee, “Hypertext Transfer Protocol -- HTTP/1.1,” UC Irvine, Compaq, W3C, Xerox, Microsoft. Internet Draft Standard Request for Comments (RFC) 2616, June, 1999.

[28] R. T. Fielding, E. James Whitehead, Jr., K. M. Anderson, G. A. Bolcer, P. Oreizy, and R. N. Taylor, “Web-Based Development of Complex Information Products,” *Communications of the ACM*, vol. 41, no. 8 (1998), pp. 84-92.

[29] P. K. Garg, “Abstraction Mechanisms in Hypertext,” *Communications of the ACM*, vol. 31, no. 7 (1988), pp. 862-870.

[30] P. K. Garg and W. Scacchi, “A Software Hypertext Environment for Configured Software Descriptions,” *Proc. International Workshop on Software Version and Configuration Control*, Grassau, Germany, January, 1988, pp. 326-343.

[31] Y. Goland, E. J. Whitehead, Jr., A. Faizi, S. Carter, and D. Jensen, “HTTP Extensions for Distributed Authoring -- WEBDAV,” Microsoft, U.C. Irvine, Netscape, Novell. Internet Proposed Standard Request for Comments (RFC) 2518, Feburary, 1999.

[32] I. P. Goldstein and D. P. Bobrow, “A Layered Approach to Software Design,” in *Interactive Programming Environments*, D. R. Barstow, H. E. Shrobe, and E. Sandewall, Eds. New York, NY: McGraw-Hill, 1984, pp. 387-413.

[33] R. Grinter, “Supporting Articulation Work Using Software Configuration Management Systems,” *CSCW: The Journal of Collaborative Computing*, vol. 5, no. (1996), pp. 447-465.

[34] K. Grønbæk, “Composites in a Dexter-Based Hypermedia Framework,” *Proc. 1994 European Conference on Hypermedia Technology (ECHT'94)*, Edinburgh, Scotland, Sept. 18-23, 1994, pp. 59-69.

[35] K. Grønbæk and R. H. Trigg, *From Web to Workplace: Designing Open Hypermedia Systems*. Cambridge, MA: MIT Press, 1999.

[36] A. Haake, “CoVer: A Contextual Version Server for Hypertext Applications,” *Proc. Fourth ACM Conference on Hypertext (ECHT'92)*, Milano, Italy, Nov. 30-Dec. 4, 1992, pp. 43-52.

[37] A. Haake, “Under CoVer: The Implementation of a Contextual Version Server for Hypertext Applications,” *Proc. Sixth ACM Conference on Hypertext (ECHT'94)*, Edinburgh, Scotland, Sept. 18-23, 1994, pp. 81-93.

[38] A. Haake and J. Haake, “Take CoVer: Exploiting Version Support in Cooperative Systems,” *Proc. InterCHI'93 - Human Factors in Computer Systems*, Amsterdam, Netherlands, April, 1993, pp. 406-413.

[39] A. Haake, J. Haake, and D. Hicks, “On Merging Hypertext Networks,” *Proc. Workshop on the Role of Version Control in CSCW*, 1995.

[40] A. Haake and D. Hicks, “VerSE: Towards Hypertext Versioning Styles,” *Proc. Seventh ACM Conference on Hypertext (Hypertext '96)*, Washington, DC, March 16-20, 1996, pp. 224-234.

[41] F. Halasz and M. Schwartz, “The Dexter Hypertext Reference Model,” *Communications of the ACM*, vol. 37, no. 2 (1994), pp. 30-39.

[42] F. G. Halasz, “Reflections on Notecards: Seven Issues for the Next Generation of Hypermedia Systems,” *Communications of the ACM*, vol. 31, no. 7 (1988), pp. 836-852.

[43] F. G. Halasz, “"Seven Issues": Revisited, Hypertext'91 Closing Plenary,” *Proc. Third ACM Conference on Hypertext (Hypertext'91)*, San Antonio, TX, 1991.

[44] D. L. Hicks, J. J. Leggett, P. J. Nürnberg, and J. L. Schnase, “A Hypermedia Version Control Framework,” *ACM Transactions on Information Systems*, vol. 16, no. 2 (1998), pp. 127-160.

[45] D. L. Hicks, J. J. Leggett, and J. L. Schnase, “Version Control in Hypertext Systems,” Texas A&M University, Technical Report TAMU HRL-91-004, July 1991.

[46] J. J. Hunt, F. Lamers, J. Reuter, and W. F. Tichy, “Distributed Configuration Management via Java and the World Wide Web,” *Proc. 7th Workshop on Software Configuration Management (SCM-7)*, Boston, MA, May 18-19, 1997, pp. 161-174.

[47] J. J. Hunt and W. F. Tichy, *RCE API Introduction and Reference Manual*: Xcc Software, 1997.

[48] Internet Archive, “Internet Archive Home Page,” (1999). Web page, accessed Dec. 5, 1999. http://www.archive.org/.

[49] R. H. Katz, “Toward a Unified Framework for Version Modeling in Engineering Databases,” *ACM Computing Surveys*, vol. 22, no. 4 (1990), pp. 375-408.

[50] S. Kydd, A. Dyke, and D. Jenkins, “Hypermedia Version Support for the Online Design Journal,” *Proc. Workshop on Versioning in Hypertext Systems*, Edinburgh, Scotland (held with ECHT'94), Sept. 18-23, 1994, pp. 9-12.

[51] D. Leblang, “The CM Challenge: Configuration Management that Works,” in *Configuration Management*, W. F. Tichy, Ed. New York: Wiley, 1994, pp. 1-38.

[52] J. J. Leggett, “Report of the Workshop on Hyperbase Systems,” Dept. of Computer Science, Texas A&M University, College Station, TX, Technical Report TAMU-HRL 93-009, 1993.

[53] J. J. Leggett and J. L. Schnase, “Viewing Dexter with Open Eyes,” *Communications of the ACM*, vol. 37, no. 2 (1994), pp. 76-86.

[54] J. J. Leggett, J. L. Schnase, J. B. Smith, and E. A. Fox, “Final Report of the NSF Workshop on Hyperbase Systems,” Texas A&M University, College Station, TX, Technical Report TAMU-HRL 93-002, 1993.

[55] C. Y. Lo, “Comparison of Two Approaches of Managing Links in Multiple Versions of Documents,” *Proc. Workshop on Versioning in Hypertext Systems*, Edinburgh, Scotland (held with ECHT'94), Sept. 18-23, 1994, pp. 17-22.

[56] T. W. Machan, “Chaucer's Poetry, Versioning, and Hypertext,” *Philological Quarterly*, vol. 73, no. 3 (1994), pp. 299-316.

[57] C. Maioli, S. Sola, and F. Vitali, “Versioning for Distributed Hypertext Systems,” *Proc. Hypermedia'94*, Pretoria, South Africa, March, 1994.

[58] K. C. Malcolm, S. E. Poltrock, and D. Schuler, “Industrial Strength Hypermedia: Requirements for a Large Engineering Enterprise,” *Proc. Third ACM Conference on Hypertext (Hypertext'91)*, San Antonio, Texas, Dec. 15-18, 1991, pp. 13-24.

[59] M. Melly and W. Hall, “Version Control in Microcosm,” *Proc. Workshop on the Role of Version Control in CSCW (held with ECSCW'95)*, Stockholm, Sweden, September, 1995.

[60] Mortice Kern Systems, “Web Integrity,” (1999). Web page, http://www.mks.com/solution/wi/.

[61] T. H. Nelson, *Literary Machines*, 93.1 ed. Sausalito, CA: Mindful Press, 1981.

[62] C. M. Neuwirth, D. S. Kaufer, R. Chandhok, and J. H. Morris, “Computer Support for Distributed Collaborative Writing: Defining Parameters of Interaction,” *Proc. 1994 Conference on Computer Supported Cooperative Work (CSCW'94)*, Chapel Hill, NC, Oct. 22-26, 1994, pp. 145-152.

[63] K. Østerbye, “Structural and Cognitive Problems in Providing Version Control for Hypertext,” *Proc. Fourth ACM Conference on Hypertext (ECHT'92)*, Milano, Italy, Nov. 30-Dec. 4, 1992, pp. 33-42.

[64] K. Østerbye and U. K. Wiil, “The Flag Taxonomy of Open Hypermedia Systems,” *Proc. Seventh ACM Conference on Hypertext (Hypertext'96)*, Washington, DC, March 16-20, 1996, pp. 129-139.

[65] K. Ota, K. Takahashi, and K. Sekiya, “Version management with meta-level links via HTTP/1.1,” (1996). Internet-Draft (expired), accessed Nov., 1999, http://www.ics.uci.edu/pub/ietf/webdav/draft-ota-http-version-00.txt.

[66] F. Pacull, A. Sandoz, and A. Schiper, “Duplex: A Distributed Collaborative Editing Environment in Large Scale,” *Proc. 1994 Conference on Computer Supported Cooperative Work (CSCW'94)*, Chapel Hill, NC, Oct. 22-26, 1994, pp. 165-173.

[67] R. Pettengill and G. Arango, “Four Lessons Learned from Managing World Wide Web Digital Libraries,” *Proc. Second Annual Conference on the Theory and Practice of Digital Libraries*, Austin, TX, June 11-13, 1995.

[68] V. Prevelakis, “Versioning Issues for Hypertext Systems,” in *Object Management*, D. Tsichritzis, Ed. Geneva, Switzerland: University of Geneva, 1990, pp. 89-106.

[69] J. Reuter, S. U. Hänßgen, J. J. Hunt, and W. F. Tichy, “Distributed Revision Control Via the World Wide Web,” *Proc. Sixth Int'l Workshop on Software Configuration Management*, Berlin, Germany, March 25-26, 1996.

[70] M. J. Rochkind, “The Source Code Control System,” *IEEE Transactions on Software Engineering*, vol. 1, no. 4 (1975), pp. 364-370.

[71] M. R. Salcedo and D. Decouchant, “Structured Cooperative Authoring for the World Wide Web,” *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, vol. 6, no. 2-3 (1997), pp. 157-174.

[72] J. Simonson, D. Berleant, X. Zhang, M. Xie, and H. Vo, “Version Augmented URIs for Reference Permanence via an Apache Module Design,” *Proc. WWW7, Computer Networks and ISDN Systems*, Brisbane, Australia, April 14-18, 1998, pp. 337-345.

[73] L. F. G. Soares, G. L. d. S. Filho, R. F. Rodrigues, and D. Muchaluat, “Versioning Support in the HyperProp System,” *Multimedia Tools and Applications*, vol. 8, no. 3 (1999), pp. 325-339.

[74] L. F. G. Soares, N. L. R. Rodriguez, and M. A. Casanova, “Nested Composite Nodes and Version Control in an Open Hypermedia System,” *International Journal on Information Systems (Special issue on Multimedia Information Systems)*, vol. 20, no. 6 (1995), pp. 501-520.

[75] N. Streitz, “SEPIA: A Cooperative Hypermedia Authoring Environment,” *Proc. Fourth ACM Conference on Hypertext (ECHT'92)*, Milano, Italy, Nov. 30-Dec. 4, 1992, pp. 11-22.

[76] W. F. Tichy, “RCS - A System for Version Control,” *Software-Practice and Experience*, vol. 15, no. 7 (1985), pp. 637-654.

[77] A. van der Hoek, “A Generic Peer-to-Peer Repository for Distributed Configuration Management,” *Proc. 18th Int'l Conference on Software Engineering (ICSE-18)*, Berlin, Germany, March, 1996, pp. 308-317.

[78] F. Vitali and D. G. Durand, “Using Versioning to Support Collaboration on the WWW,” *Proc. Fourth International World Wide Web Conference*, Boston, MA, November, 1995, pp. 37-50.

[79] D. W. Weber, “CM Strategies for RAD,” *Proc. Ninth Int'l Symposium on System Configuration Management (SCM-9)*, Toulouse, France, Sept. 5-7, 1999, pp. 204-216.

[80] E. J. Whitehead, Jr., “An Architectural Model for Application Integration in Open Hypermedia Environments,” *Proc. The Eighth ACM Conference on Hypertext, Hypertext'97*, Southampton, UK, April 6-11, 1997, pp. 1-12.

[81] E. J. Whitehead, Jr., “Goals for a Configuration Management Network Protocol,” *Proc. 9th Int'l Symposium on System Configuration Management (SCM-9)*, Toulouse, France, Sept. 5-7, 1999, pp. 186-203.

[82] E. J. Whitehead, Jr., K. M. Anderson, and R. N. Taylor, “A Proposal for Versioning Support for the Chimera Systems,” *Proc. Workshop on Versioning in Hypertext Systems*, Edinburgh, Scotland (held with ECHT'94), Sept. 18-23, 1994, pp. 45-54.

[83] E. J. Whitehead, Jr. and Y. Y. Goland, “WebDAV: A Network Protocol for Remote Collaborative Authoring on the Web,” *Proc. Sixth European Conference on Computer Supported Cooperative Work*, Copenhagen, Denmark, Sept. 12-16, 1999, pp. 291-310.

[84] U. K. Wiil and J. J. Leggett, “Hyperform: Using Extensibility to Develop Dynamic, Open and Distributed Hypertext Systems,” *Proc. Fourth ACM Conference on Hypertext (ECHT'92)*, Milano, Italy, Nov. 30-Dec. 4, 1992, pp. 251-261.