The Molhado Hypertext Versioning System

Tien N. Nguyen
Dept. of EECS
Univ. of Wisconsin-Milwaukee
tien@cs.uwm.edu

Ethan V. Munson
Dept. of EECS
Univ. of Wisconsin-Milwaukee
munson@cs.uwm.edu

John T. Boyland
Dept. of EECS
Univ. of Wisconsin-Milwaukee
boyland@cs.uwm.edu

ABSTRACT

This paper describes *Molhado*, a hypertext versioning and software configuration management system that is distinguished from previous systems by its flexible product versioning and structural configuration management model. The model enables a unified versioning framework for atomic and composite software artifacts, and hypermedia structures among them in a fine-grained manner at the logical level. Hypermedia structures are managed separately from documents' contents. Molhado explicitly represents hyperlinks, allowing them to be browsed, visualized, and systematically analyzed. Molhado not only versions complex hypermedia structures (e.g., multi links), but also supports versioning of individual hyperlinks. This paper focuses on Molhado's hypertext versioning and its use in the Software Concordance environment to manage the evolution of a software project and hypermedia structures.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement;
D.2.9 [**Software Engineering**]: Management;
H.5.4 [**Information Interfaces and Presentation**]: Hypertext / Hypermedia

General Terms

Management

Keywords

Hypertext Versioning, Software Configuration Management, Version Control, Software Engineering

1. INTRODUCTION

One of the most important tasks associated with software engineering is to improve the analysis, design, construction, verification, and management of software artifacts. The software artifacts produced during the development process are often logically related to each other. The logical relationships appear in many forms. They can be captured in the form of manually maintained cross-references, organizational indexes, or table of contents. While composition relationships are used to organize software components with respect to their granularity, build dependencies specify dependencies between source objects and derived objects in a build process. Semantic dependencies exist among source code, requirements, designs, and implementations. Documents under developments or maintenance are changed and updated to produce the next revision. Therefore, the set of active logical relationships among them can also change over time as a result.

Acknowledging the importance of relationship management in software engineering tools, many researchers have explored the hypertext approach [5]. As discussed, hypertext supports for software development must accommodate changes in both software artifacts and relationships. Here, *versioned hypermedia* comes into play as a natural means for representing and versioning software artifacts and their relationships. Versioned hypermedia (or *hypertext versioning*) is concerned with storing, retrieving, and navigating prior states of a hypertext, and with allowing groups of collaborating authors to develop new states over time [46]. Despite having many successes, existing hypermedia-based software engineering tools provided only limited support for the evolutionary aspects of a software project, especially for maintenance and evolution of document relationships. In the meantime, many existing hypertext versioning frameworks were not particularly designed for software development. Applying hypertext versioning technology in software engineering exposes certain requirements for which most existing hypermedia-based software development tools have yet to provide a complete solution.

First of all, the representation of hyperlinks must be explicit and facilitating systematic analysis, information retrieval, and visualization of document relationship networks. Implicit relationships cannot be browsed, navigated, queried, or systematically analyzed. Therefore, it hinders developers from having a full understanding of the system and from discovering important information. Secondly, hyperlinks must have variable arity since it is common for relationships to connect multiple documents. Document relationships can exist both between documents and within a single document. In software documents, which can be rather lengthy, it will be common that relationships will connect relatively small sections of material, such as functions, paragraphs, or subsections. So, a hypermedia system needs to support both coarse-grained linking (connecting entire documents) and fine-grained linking (connecting document fragments). In addition, hypertext versioning systems designed for software engineering should separate hypertext structures from document contents. This is particular important for program source code, whose lexical and syntactic rules hinder the use of embedded hyperlinks. The separation also gives developers

HT'04, August 9–13, 2004, Santa Cruz, California, USA.

more flexibility to have different relationship networks to complete particular tasks at hand without modifying software documents.

In hypertext authoring applications, versioning for links might not be crucial. However, for software development, the ability to track changes for a particular logical relationship is very useful, for example, in consistency management tasks. The fact that existing hypermedia systems for software development do not version links is a significant factor preventing their wider use in the software engineering domain [46]. In order to correctly maintain fine-grained logical links, hypertext versioning systems must also version software artifacts in a fine-grained manner at the logical level. This ability is particularly useful in fine-grained traceability and change tracking tools since file is a much larger piece than an information unit affected by a single change. In addition, the history of hypertext structures also needs to be recorded since it would help engineers to understand better the development of software documents and logical relationships among them over time. An important user interface requirement is that versioning for hypertext structures should occur as transparently to users as possible. The versioning system should not constantly prompt users for checking in and out the member entities of a hypertext structure every time a new version of the structure is created. Beside these requirements, general requirements for a hypertext versioning system such as the ones listed by Haake [20] and by Whitehead [46] are also very important to the success of software relationship management.

To improve the relationship management in software engineering tools, our research has built a hypertext versioning and configuration management (SCM) system, named *Molhado*. Molhado distinguishes itself from existing systems by its hypertext versioning framework in which hypermedia infrastructures are built based on a *product versioning* and *structural* SCM model. Structure versioning in Molhado allows for fine-grained version control of logical structure of software components, hypermedia structures among them, and their logical organization within a software project. This paper focuses only on Molhado's versioned hypermedia and its use in the Software Concordance (SC) to manage versions of software artifacts and hypermedia structures.

2. RELATED WORK

Several researchers have applied hypertext technology to improve software document and relationship management. DynamicDesign [7] defines a fixed set of information units and relationships in an integrated development environment (IDE) for C language. ChyPro [2] is a hypermedia-based IDE for SmallTalk-80, which allows for linking distant pieces of code or documentation together. Østerbye supports literate programming [37] by modeling a SmallTalk system as a hypertext. The Chimera open hypermedia system [3] improves relationship management by providing hypermedia services across multiple document types maintained by different applications. None of these hypertext-based systems addresses the issue of relationship evolution.

Hypertext versioning offers an appealing approach to representing the evolution of software documents and relationships. However to date, IDEs based on versioned hypermedia have only provided simple versioning of objects, with limited versioning of links. Also, none supports interactive program analysis. Systems providing versioning only for data include Xanadu [33], KMS [1], DIF [17], and Hyperform [48]. In both RCS-based HyperWeb [16] and HyperCASE [12], the smallest versionable information unit is a file, which is too large to be a suitable basis for document relationship management. HyperPro [36] supports versioning of objects as small as procedures. NUCM [42] follows client-server SCM model in which a NUCM client interacts with a NUCM repository server. Palimpsest's change-oriented model addresses concurrency control in collaborative editing [15]. Vitali and Durand propose VTML (versioned text markup language) [6] to express change operations in editing structured documents including XML and HTML. To support distributed authoring for Web contents, WebDAV [47] extends HTTP protocol to include operations for overwrite prevention, properties, and namespace management, while DeltaV [47] builds upon WebDAV to offer versioning, workspaces, activities, and configuration management.

Prominent approaches in version control for hypertext structures are *composition model*, *total versioning model*, and *product versioning model*. In *composition model*, versions of atomic components are maintained and assembled into composite components. Each of atomic components has a version history. The relationships between these version histories are defined by revision selection rules (RSRs) for composing a version of a composite component or a hypertext structure. Although the use of RSRs provides flexibility, it creates some problems [4]. Efficiently maintaining and evaluating rules increase the complexity of structure versioning and make user interfaces complicated [46]. The *indirect representation* of a hypertext structure makes it impossible to find out elements of the structure, unless the rules are actually applied and executed. Differences between a structure's revisions can not be deduced from comparing RSR specifications. Therefore, change management would not be efficient with this approach. To avoid the use of rules, some systems maintain a notion of *current context* [14, 20, 21]. A context could be some kind of coherent structure, for example, a document, a document collection [36], or a partition of a hypertext [14]. However, the complexity of maintaining the current context becomes significant if composite components have many nested levels and hyperlinks among them.

The composition model is the core of many existing composite-based hypertext versioning systems. Composite-based versioning systems comprise an important class of hypertext versioning systems, and are characterized by the use of a container object to contain documents and hypertext networks [46]. In Neptune [13], the composite is used to represent the isolated work area of each collaborator. HyperPro [36] records changes for links by placing links inside a "version group" (i.e., a composite) that is versioned. It has both links to a specific version of a node and links to a node in general. Similar to HyperPro, HyperProp [39] does not record history of links individually. In HyperPro's and HyperProp's structure versioning, RSR is stored on the structure, affecting all link endpoints, and provides selection of specific document revision from a versioned document. With this RSR approach, it is inefficient to evaluate rules across the revisions of a specific link [46].

The research addressing versioning in open hypermedia systems has also applied composition versioning approach. Microcosm [29] has a context-like structure called the "application", which is versioned. It uses RCS [41] to implement version control of nodes. RHYTHM [28] is a distributed hypertext system that tackled problems arising from distribution and versioning. In the versioning proposal for Chimera [45], a "configuration" is a named set of versions of Chimera hypermedia elements, representing the subset of a hypertext structure that might be affected by modifications to an externally stored object. RCS is assumed to be the external versioning system for objects. In the hypermedia version control framework (HURL) [22], a hypertext structure is represented by an "association", which is a collection of identifiers of links and anchors, and documents that defines a connection. HURL provides data versioning at document level. IAM group has investigated the Fundamental Open Hypermedia Model (FOHM) [31]'s contextual model as implemented within the contextual structure server, Auld

Linky [30], to see if the contextual model could replace the selection engine in a hypertext versioning server. They suggested to extend FOHM to include multiple connections and "contexts" on those connections [19]. The GAIA [23], a versioning framework for open hypermedia, has its "composite" included links, anchors, document, queries, other composites, graph objects, and a RSR.

In *total versioning model*, all items are versioned, including composites and atomic components. Each item still has its own version space. The relationships among items' version spaces are defined by RSRs or by versions of composites referring to versions of other components. For example, in ClearCase [24], a directory version refers to versions of other directories and files. The main principle is that when a component is modified to create a new version, new versions of all components that are its ancestors in compositional or relational hierarchy must be created [10]. In PCTE [44], a new version is created by recursively copying the whole composition hierarchy and establishing successor relationships between all components. In CoVer's [20] and VerSE's [21] structure versioning, RSR is stored on the containment arc between a container and its containees, providing selection of link revisions and document revisions from versioned links and versioned documents respectively. This structure versioning increases the work that must be performed to ensure that a structure container holds a consistent hypertext [46]. In CoVer and VerSE, a context is a description of the task being performed when the objects (involved in the task) are versioned. Such context information is used in assisting with the version selection process. SEPIA [40] has the notion of "composite node", which contains a partially ordered set of nodes and links and represents subgraphs of the hypermedia network. Its "container object" contains view information for each data object (link,node, composite node). The total versioning model also has the version proliferation problem [10], which will be discussed later.

In contrast to total versioning, *product versioning* establishes a total view of a software product, or even an entire database. This is done by arranging versions of all items in a uniform, global version space. Change-based product versioning systems include COV [25], Aide-de-camp [11], and PIE [18]. PIE stores logical changes in a composite called "layer", which can contain both objects and links. Voodoo [38], a state-based product versioning SCM system, also manages the evolution of the whole software project. A version in Voodoo is a global discrete point in a time baseline, rather than a point in a tree-structured discrete time abstraction as in Molhado. Wagner's product versioning [43] integrates version management with incremental program analyses.

Much research has been performed to provide fine-grained version control for software documents. Magnusson *et al* worked on tree-based versioning for programs and hierarchically structured documents to support collaborative editing in COOP/Orm [27]. The framework's principles include sharing of unchanged nodes among versions, change propagation, and change-oriented representation of differences. The framework works only on tree-based documents such as XML. POEM [26] uses total versioning to provide version control in terms of functions and classes. The Unified Extensional Versioning Model [4] supports versioning for a tree by composite, atomic, and link nodes. In Coven [9], the exact size of versioned fragment depends on the supported document: entire method, function, or field declaration for C++ and Java programs, or paragraph of text in LaTeX documents.

3. SYSTEM OVERVIEW

Our system is designed based on a layered architecture (see Figure 1). Each layer uses services of lower layers to provide the infrastructures for upper layers. The important tasks and notions

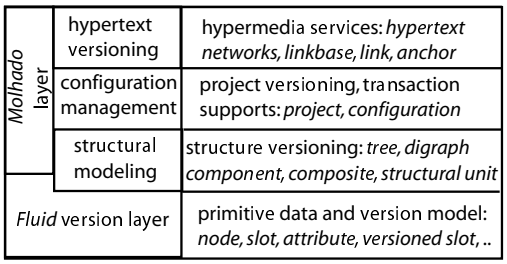


Figure 1: System overview

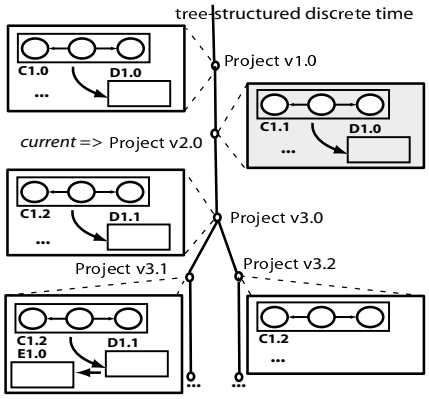


Figure 2: Product versioning

introduced at each layer are listed in Figure 1. The *Fluid* layer (Section 4) provides the product versioning engine for our system. It is built as part of the Fluid project [8]. It provides a combination of a version model and a primitive data model to allow for version control of many different data types. Its persistence model allows for the storage of versioned data. Since the Fluid model works at primitive data, it is necessary to construct high-level versioning infrastructures for different types of software artifacts and hypermedia structures. That is the task of the Molhado's structural modeling layer (Section 5). While the structural modeling layer takes care of components in individual, configuration management layer (Section 6) applies structural modeling infrastructures to the logical and compositional organizations among software artifacts. This layer's responsibilities include supports for versioning a project's structure and for users' operations. The hypertext versioning layer (Section 7) is built based on structural modeling infrastructures. Hypertext versioning services in Molhado have been integrated into the Software Concordance environment (Section 8).

4. PRODUCT VERSIONING MODEL

This section describes the Fluid *product versioning model*. Instead of focusing on individual components, Fluid/Molhado versions a software project as a whole (see Figure 2). All system objects including (atomic and composite) components and hypertext structures are versioned in a *uniform, global version space*. A *version* is global across the whole project and is a point in a *tree-structured discrete time* abstraction, rather than being a *particular state* of a system object as in total versioning and composition versioning approaches [10]. The state of the whole software system is captured at certain discrete time points and only these captured versions can be retrieved in later sessions. The *current version* is the version designating the current state of the project. When the

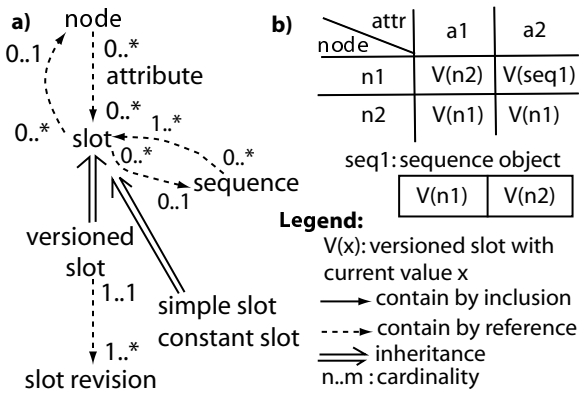


Figure 3: Node Slot Pattern

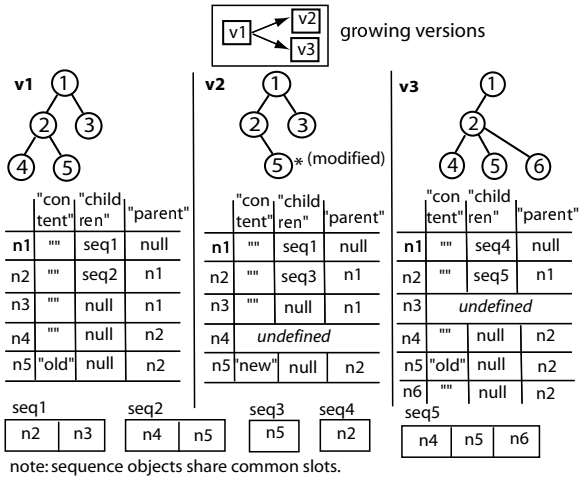


Figure 4: Tree versioning

current version is set to a captured version, the state of the whole project is set back to that version. Changes made to the project at the current version create a temporary version, *branching off* the current version. That temporary version will only be recorded if a user explicitly requests that it be captured. To record the history of an individual object, the whole project is captured. Capturing the whole project is quite efficient because the versioning system only records changes and works at a very small granularity.

The primitive data model of Fluid product versioning engine is the *node-slot pattern* [8]. Figure 3a) summarizes the node-slot pattern using the notation of Whitehead's containment model [46]. A *node* is the basic unit of identity and is used to represent anything. A *slot* is a location that can store a value in any data type, possibly a reference to a node or a *sequence* (will be described later). A slot can exist in isolation but more typically slots are attached to nodes, using an *attribute*. An attribute is a mapping from nodes to slots. An attribute may have particular slots for some nodes and maps all other nodes to a default slot. The data model can thus be regarded as an attribute table whose rows correspond to nodes and columns correspond to attributes. The cells of the table are slots. Once we add versioning, the table gets a third dimension: the version. There are three kinds of slots. A *constant slot* is immutable; such a slot can only be given a value once, when it is defined. A *simple slot* may be assigned even after it has been defined. The third kind of slot is the *versioned slot*, which may have different values in different versions (*slot revisions*). A *sequence* is a container with slots of the same data type. It has a unique identifier. Sequences may be fixed or variable in size and share common slots together. Figure 3b) shows a simple example of an attribute table. The versioned slot associated with node "n1" via attribute "a1" currently holds a reference to the node "n2". Sequence "seq1" has two versioned slots referring to "n1"' and "n2" respectively.

5. STRUCTURE VERSIONING

To be able to control versions of software documents at a fine granularity and at the logical level, Molhado follows a *structure-oriented* approach where each document is considered to be logically structured into fine units, called *structural units* or *logical units*. This approach is often taken in *structured document* research, e.g. SGML and XML. In this approach, each software document is represented by a *document tree* or a *document graph* in which each node encodes a logical unit of the document. Since XML has become the standard structured document format and very successful in representing many different data types, it is very natural to use XML for representing non-program artifacts. Syntactical rules for a document and its structural units are defined by users in a specification such as a *document type definition* (DTD) or *XSchema* specification. For a program, an abstract syntax tree (AST) perfectly represents its logical structure. To achieve fine-grained versioning for documents, it is obviously that a versioning framework for tree and graph data structures is required.

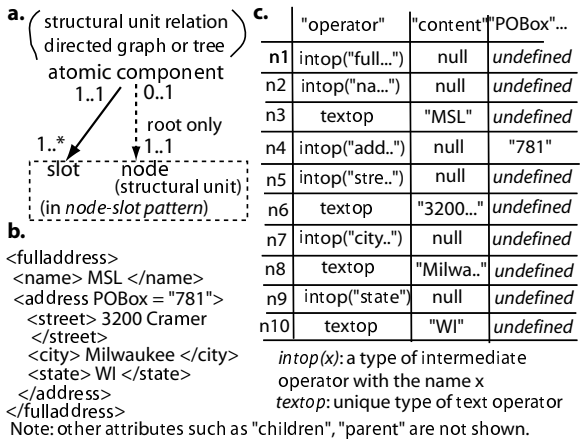
5.1 Tree-based versioning

This section describes the fine-grained version framework for the tree data structure, which is built as part of the Fluid project [8]. Trees are built from nodes, slots, and attributes. A tree is defined with two main attributes: 1) the "children" attribute maps each node to a sequence holding its children, and 2) the "parent" attribute maps each node to its parent. Figure 4 illustrates this via an example. In this example, a "content" attribute is also defined that holds a string value for some of the nodes. Note that other attributes can also be defined for nodes. Assume that there are three versions: *v1*, *v2*, and *v3*. Versions *v2* and *v3* branch off from version *v1*. The shape of the tree at each of the three versions is shown. Version *v2* has two differences from the version *v1*: node 4 was deleted and the content of node 5 was changed. Version *v3* has an inserted node (node 6) and node 3 was deleted. The values of versioned slots in the attribute table are changed to reflect modifications to the tree at these versions. For example, at the version *v2*, the "content" slot (i.e. the slot defined by the attribute "content") of node 5 contains a new value (the string "new"), and the "children" slot of node 2 contains a reference to a new sequence object (*seq3*). *Seq3* has only one slot, which contains a reference to node 5 since node 4 has been deleted. If there is a request for the values of slots associated with node 4 at *v2*, a run-time error will be reported. Versioning for a directed graph is similar except that the attribute table does not have the "parent" attribute.

With this versioning scheme, the history of any node (logical unit) can be recorded. For example, if the tree in Figure 4 represents a program in which node 2 represents a method, the content of that method at *v1*,*v2*, and *v3* can be tracked by starting at the row "n2" of the attribute table at respective version, then following the "children" slot to display the method.

5.2 Unified versioning framework

To be able to support fine-grained versioning and hyperlinking at the logical level, we have developed a *unified versioning frame-*



	"operator"	"content"	"POBox"...
n1	intop("full...")	null	*undefined*
n2	intop("na...")	null	*undefined*
n3	textop	"MSL"	*undefined*
n4	intop("add..")	null	"781"
n5	intop("stre..")	null	*undefined*
n6	textop	"3200..."	*undefined*
n7	intop("city..")	null	*undefined*
n8	textop	"Milwa.."	*undefined*
n9	intop("state")	null	*undefined*
n10	textop	"WI"	*undefined*

Figure 5: Atomic component

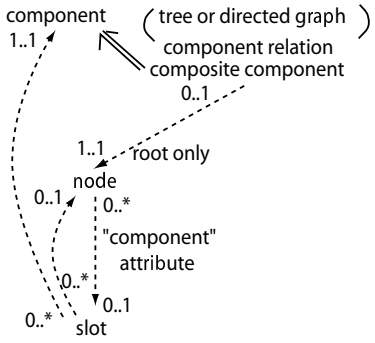


Figure 6: Composite Component Modeling

work for both *atomic* and *composite components* in which all components are versioned in the same global version space. The framework is based on the tree-based and graph-based version control scheme that was described earlier. In general, a *component* is an abstraction to represent a coarse-grained logical object, which may or may not be internally structured. It can be defined as a logical entity that can be versioned, saved, loaded, and exists within the version space of a software project. Each component carries an internal *component identifier* (CID) that serves to identify it uniquely within the software project. An external CID is a name that may be assigned by users. A component can represent any logical object such as documents, programs, directories, object-oriented classes, functions, modules, etc.

5.2.1 Atomic components

Atomic component is the basic unit for composition and aggregation in a composite component. The word "atomic" does *not* mean that the component can not be divided into smaller units. In fact, in Molhado, to accommodate fine-grained version control for components, an atomic component can be internally composed of *logical units* or *structural units* as mentioned. The internal structure of an atomic component is represented by a tree or a directed graph where each node represents a logical unit. Figure 5a) shows the part of our data model for an atomic component. Each atomic component referentially contains the root node (or pseudo-root) of a tree (or a graph), which is constructed via the node-slot pattern. A component's internal properties whose histories need to be captured are represented by versioned slots that are inclusively contained within the component. For example, a versioned slot is defined for a component's name, which might change at different versions.

Figure 5b) and c) show an example of how an XML document is represented. An XML document is represented by a *syntax tree*, which has an additional slot that stores an *operator* for each node via "operator" attribute. An operator identifies the syntactical type of its node and determines the number and syntactical types of the node's children. Nodes in an XML syntax tree have operators drawn from two categories: *intermediate* and *text* operators. The unique *text operator* is used to represent XML's character data (CDATA) construct. Each node associated with the text operator has an additional slot (defined by the "content" attribute) that holds the CDATA string. Each element node in an XML document is associated with an *intermediate operator*, whose name is the element's name. Each node associated with an intermediate operator has one additional slot for each XML element-level attribute that is defined for that element. In Figure 5, node "n1", representing the "fulladdress" element, is associated with an intermediate operator with the name "fulladdress". Node "n3", whose "content" slot contains the string "MSL", is associated with the unique text operator. The "POBox" attribute of "address" is valid only for "n4".

To represent for a Java program, a set of operators is defined to represent the Java AST, with each node in the AST being represented as a node in the node-slot pattern. In a Java syntax tree, each node is associated with a slot whose value refers to a Java operator that defines the syntax for the syntactical unit represented by the node. For example, an addition expression "x + y" would be represented by a node that is associated with a slot containing the "AddExpression" Java operator. Children nodes are left child and right child of the addition expression. They are both associated with the same operator named "NameExpression", and each of them has a special slot whose value refers to its own identifier.

5.2.2 Composite components

A composite component consists of atomic components and/or other composite components. Examples include compound documents, Java packages, architectural composite components, logical structures such as class hierarchy, UML diagrams, Entity Relationship (ER) diagrams, etc. Unlike in total versioning and composition versioning approaches, Molhado puts all components (atomic and composite) under a uniform global version space. Figure 6 shows the part of Molhado's data model for composite components using Whitehead's notations. A composite's internal structure is represented by either a tree or a directed graph. The composite contains the root (or pseudo-root) of the tree (or the graph). For a tree or graph node (except the pseudo-root), the associated "component" slot holds a reference to a component that the composite contains.

Figure 7 shows two examples of composite components. Figure 7a) shows package component A, which contains class components ("class1" and "class2") and package component B having "class3" and "class4". Package A is represented by a tree rooted at "n1". Package A referentially contains the root node "n1". Nodes "n2" and "n4" are associated with slots containing references to "class1" and "class2" respectively, while the "component" slot for node "n3" refers to the composite component B. For simplicity, other attributes (e.g. children and parent) are not displayed. In Figure 7b), composite component "program1" is represented in the same manner. The attribute table is not shown. However, the nodes "n2", "n4", and "n11" in Figure 7b) are associated with slots containing references to "class2", "class3", and "class2" respectively. Note that "class3" and "program1" are composite, while "class2" is atomic. In contrast, "class1" is a logical unit in "program1".

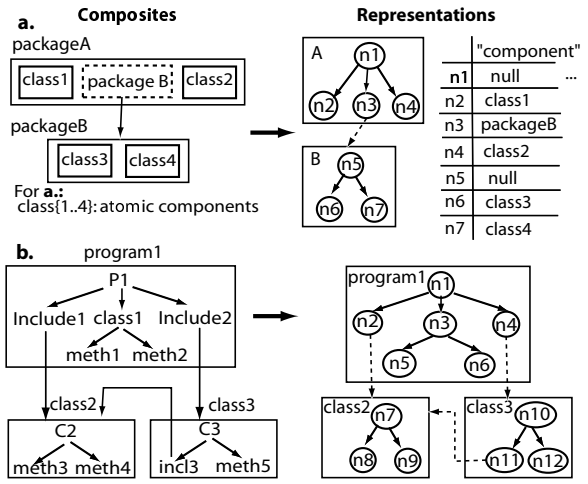


Figure 7: Composite Component Example

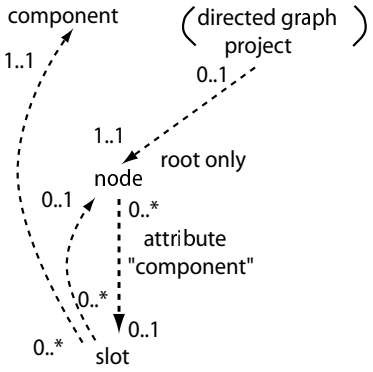


Figure 8: Data model for a project

This framework allows software components to be versioned at both fine (i.e. logical unit level) and coarse (i.e. atomic/composite component level) granularities in the same manner since the internal structures of both composite and atomic components are represented via trees and directed graphs. Versioning for composite objects in Molhado does not require version selection rules since the current version is globally set across a project. Unlike total versioning in many SCM systems, no version proliferation [10] occurs in Molhado. For example, assume that the name of "meth4" in Figure 7b) is changed to create a new version, total versioning also creates new versions of "C2", "incl3", "C3", "Include2", and "P1". While version proliferation need not involve physical copying, it stills creates cognitive overhead for users. In contrast, in Molhado only a single new global version is created and only the "content" slot of the node "n9" (representing for "meth4") changes its value at the new version. This composite mechanism (Figure 6) is also used to represent logical relations among components such as the parent-child relations among classes in an object-oriented paradigm. In this case, a tree or directed graph represents the logical relations among components, rather than the composite's internal structure. For example, a composite component can be created to represent a class hierarchy.

Based on this framework, we have built a rich set of components including Java classes, Java programs with embedded multimedia documentation, documentation in XML, HTML, Scalable Vector Graphics (SVG) documents, UML diagrams, and binary data documents. Molhado is able to import and export its internal documents (stored in our persistent format) from and to external formats (Java, XML, HTML, ASCII text, SVG) at any version. Binary data files such as image, audio, and object files are considered to have no internal structure for versioning purpose. To enable HTML-style *embedded hyperlinks* among these components, an "href" attribute is defined for each node in a component's tree or graph. A "href" slot contains a URL referring to a logical unit in any component. Therefore, embedded hyperlinks can be attached to and can point to any logical unit in components. Importantly, this approach to binding embedded hyperlinks to source code does not interfere with program compilation, which ignore the "href" attribute.

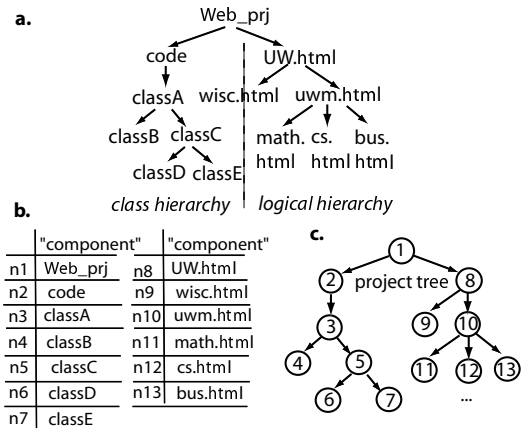


Figure 9: Versioning for a project

6. CONFIGURATION MANAGEMENT

The distinguished characteristic between Molhado and many existing SCM systems is its ability to organize and to version a software project in terms of *logical* components. Many of them treat a software system as a "set of files" in directories on a file system, and stable configurations are defined implicitly as sets of file versions with a certain label or tag. This creates an impedance mismatch between the design and implementation domain (logical level) and the configuration management domain (file level).

To minimize the mental gap and and to enable the version control for the logical structure of a software project, we introduce a notion of *project*. A project is a named entity that represents the *overall system logical structure* of a software project. A project's version is called *configuration*. Figure 8 shows the part of Molhado's data model for a project. Although a project is often tree-structured, complicated logical structures such as ER diagrams might require a directed graph representation. To be general, a project referentially contains a pseudo-root node of a directed graph. For a given graph node (except the pseudo-root), the associated "component" slot contains a reference to a composite or an atomic component. Since the overall architectural structure (i.e. a *project*) is represented as a directed graph, it is versioned in accordance to the directed graph versioning scheme. Notice that modeling for a project is very similar to a composite component. However, a project can *not* be used as a composite component in any compositional hierarchy. Instead, it must be regarded as the outermost composite of all components and sets up the context for the global version space. That is, the scope of the *current version* is at the project level.

Figure 9a) shows the logical structure of a Web project. Its representations in Molhado are in Figure 9b) and c). In this example, the Java class components are organized into a class hierarchy, while

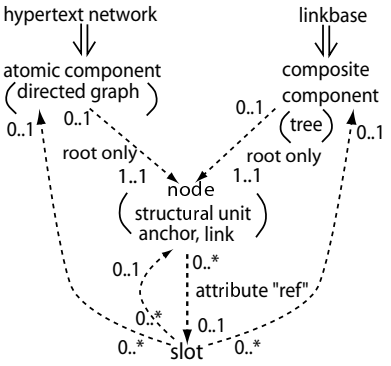


Figure 10: Data model for hypertext

HTML components are structured accordingly to the logical organization of the University of Wisconsin system. The project's structure is obviously logical and independent of a file system. Note that the *directory components* "Web-prj" and "code" are used to group other components and do not correspond to directories in a file system. Notice that this product versioning approach, which versions a software project as a whole entity, always assures the construction of a consistent configuration since when a project version is chosen as current, the project's directed graph will be correctly retrieved and versioned slots associated with nodes in the graph will refer to proper components at the current version as well. Then, the internal structure of each component and the contents of slots are determined as described in Section 5.

7. HYPERTEXT NETWORK VERSIONING

The structure versioning framework in Section 5 also provides infrastructures for the Molhado's versioned hypermedia model. The model is based on the following concepts: *linkbase*, *hypertext network*, *link*, and *anchor*. A linkbase is a container for hypertext networks and/or other linkbases. The relation between a linkbase and a hypertext network is the same as the relation between a directory and a file in a file system. A hypertext network can belong to only one linkbase. A hypertext network contains links and anchors. A link is an association among a set of anchors. An anchor can belong to multiple links. A link or an anchor can also belong to multiple hypertext networks. An anchor is used to denote the region of interest within a component, and it refers to either a component or a structural unit in a component. Links and anchors can be associated with any attribute-value pairs.

Figure 10 shows part of the data model for hypermedia entities. A hypertext network is implemented as another type of atomic component, whose internal structure is a directed graph. It contains only the pseudo-root of the directed graph. Each link or anchor is represented by a node in that graph. A directed edge connects an anchor's node to a link's node if the link contains the anchor. An additional attribute, attribute "ref", is defined to associate an extra slot to each anchor's node in the graph. The slot holds a reference to either a component or a structural unit within a component (but *not* to a hypertext network or to a linkbase). That structural unit (or that component) is considered as the position of the anchor. This separation between anchors and structural units allows for the separation between hypertext networks and component contents. Attribute-value pairs associated with links and anchors are implemented via the node-slot pattern as in XML documents. A linkbase is implemented as a composite component, whose internal structure is a tree. As in a composite component, for each node in the tree,

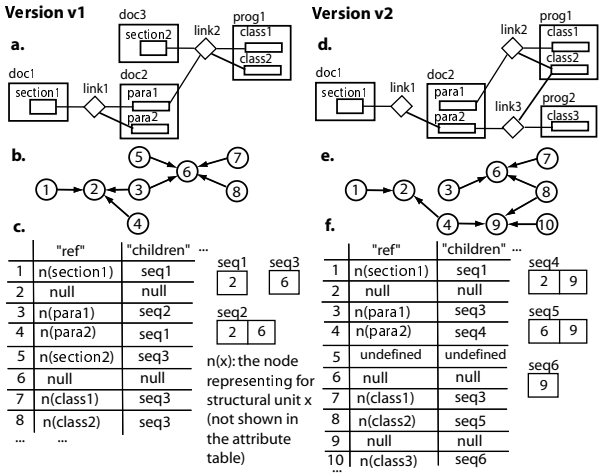


Figure 11: Versioning for hypertext networks

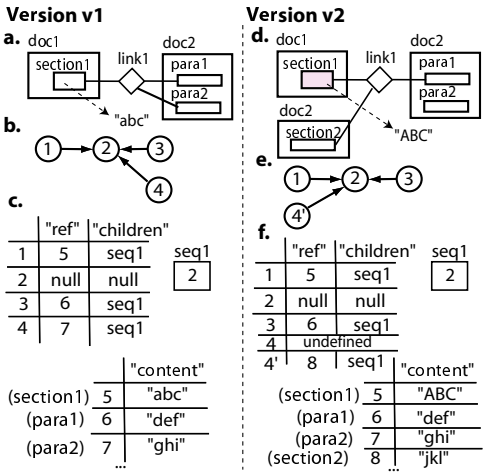


Figure 12: Versioning for hyperlinks

the associated "component" slot (not shown in Figure 10) contains a reference to a hypertext network or another linkbase.

Figure 11 shows an example of hypertext network versioning. Figure 11a) and d) display the network at two versions *v1* and *v2*. The directed graphs representing for structures of the network at these two versions are in Figure 11b) and e). Links' nodes (e.g. nodes 2 and 6) have edges coming into them and do not refer to anything. Figure 11c) shows part of the attribute table for the network at version *v1*. The "ref" cell for an anchor node (e.g. node 1) contains a reference to the corresponding document node (e.g. n(section1)). Figure 11f) shows the attribute table at version *v2*. Node 5 is deleted. Node 3 now has only one child. Node 9 (representing *link3*) and node 10 (representing *class3*) are just created. The "ref" cell for node 10 points to *class3*.

In general, via the directed graph versioning scheme, the history of a hypertext network is recorded. If the position of an anchor is changed from a structural unit to another, the "ref" slot for the anchor's node will be changed to refer to the new structural unit. Deletion to document nodes might result in deletion of anchors. It is true that any modification to a document node's content, for example "section1" of "doc1", will change the "content" slot of the node. Therefore, any change to a hypertext network and related

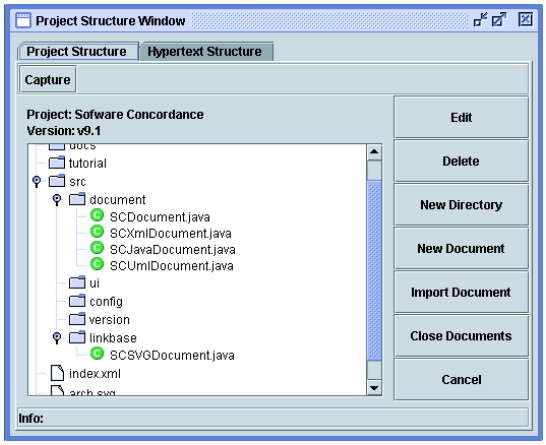


Figure 13: Project structure window

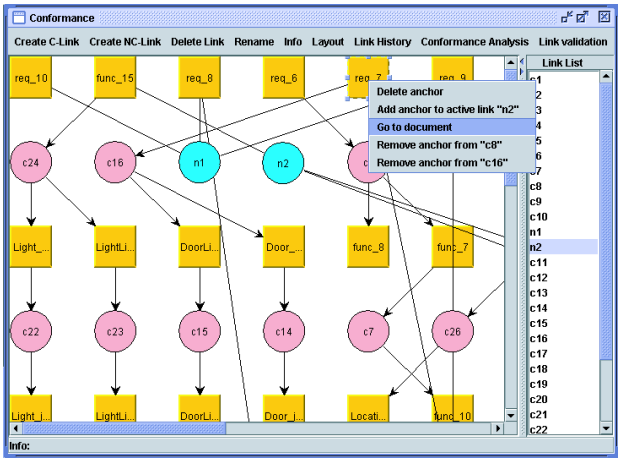


Figure 14: A hypertext network editing window

components will be reflected in a newly created version. In addition, the history of a link is recorded as well. Figure 12 shows an example of link versioning. Figure 12a) and d) show the link at two versions *v1* and *v2*. In version *v2*, "para2" is removed from the link, "section2" of "doc2" is inserted, and the content of "section1" is changed from "abc" to "ABC". Figure 12b) and e) display the directed graph representing the link at *v1* and *v2*. Figure 12c) and f) show the attribute table for the graph and document nodes at *v1* and *v2*. Similar to versioning for a hypertext network, changes in the attribute table reflect changes in member anchors of the link and contents of document nodes.

8. SC ENVIRONMENT

This section describes Molhado's hypertext versioning services that were integrated into the Software Concordance (SC), which supports program analyses, structural editing, and HTML hyperlinks among Java programs and multimedia documentations [35].

8.1 SCM operational model

Molhado and the SC user interfaces support a variety of transactions. First of all, a user can open an existing project. After selecting the *current (working) version* from a project history window, the system displays the project's structure and its components in a project structure window (see Figure 13). This window serves as an overview and table of contents for the project at the selected version. From this window, the user can choose to edit, delete, import, or export any component and hypertext network. Also, via the window, users can graphically modify the project's structure. The version that is initially displayed in this window is called *the base version*. If any modification is made to the project at this base version, a new version would be temporarily created, branching off the base version. The word "modified" will be attached to the base version's name in this window and the window now shows information at the temporary version derived from the base version. The user can choose to discard any derived (temporary) version (i.e. any changes to the base version), or to *capture* the state of the project at a version. Capture will change a temporary version into a captured one. Bookkeeping information such as name, date, authors, and descriptions can be attached to the newly captured version for later retrieval. The captured version plays the role of a checkpoint to which the user can refer and becomes the *new* base version of the project structure window. However, no data is saved after a capture.

Switching to work on a different version can be done explicitly or implicitly, whether or not the current version has been captured. If the user moves the mouse focus to a window, the working version is automatically set to the version that window is displaying. The user can also explicitly select a different version from the project history window and open it. The user may *commit* changes at any time. Upon issuing this command, the user is asked which uncaptured, temporary versions should be saved and the chosen versions are then saved to the file system along with any already captured versions. Only the differences are stored. The user may also save complete version snapshots, which can improve version access time. Each user does not see changes from others, therefore, no locking mechanism is needed. Users will share the data files and using merging tools to collaborate. A more centralized versioning repository similar to CVS [32] is being implemented. A set of *diff tools* is provided to compare two arbitrary versions (not necessarily predecessor or successor of each other) at different levels: 1) structural unit, 2) component, and 3) the whole system, in both structural and line-oriented manners.

8.2 Versioned hypermedia services

Versioned hypermedia functionality in SC can be divided into two groups: 1) linkbase and hypertext network services, 2) link and anchor services. The first group are mainly provided via menus at the project structure window. The window displays the structure of the project as well as of linkbases. Users can create a linkbase or a hypertext network, delete existing hypertext networks or linkbases, re-structure linkbases and relocate networks among linkbases, open an existing hypertext network, select a hypertext network to be *active*, import and export a hypertext network from and to XLink format at any version.

Figure 14 shows an example of a hypertext network. A circle represents for a link, a rectangle for an anchor. In this hypertext network, there are two types of links: *causal* and *non-causal*. These two types are used to build supports for traceability and consistency management among software documents [34]. Causal links carry with them an implied logical ordering of the documents involved. They may have multiple source and multiple target anchors. Non-causal links exist when documents or parts of them must be related and agree with each other, but the causality cannot be clearly identified. While non-causal links are directly represented by Molhado's links, causal links are extended from them. In Figure 14, there are directed edges coming from source anchors to causal links, and from causal links to target anchors. There are only non-directed

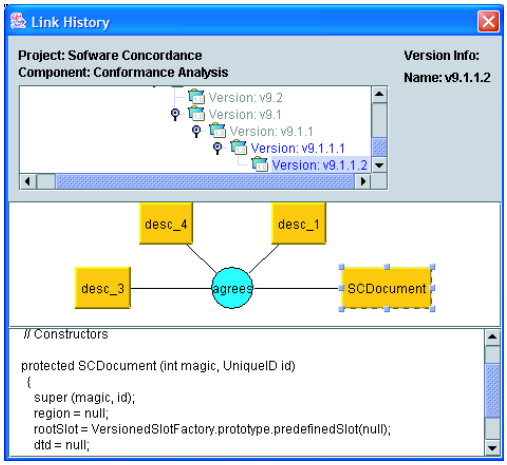


Figure 15: Versioning for a link

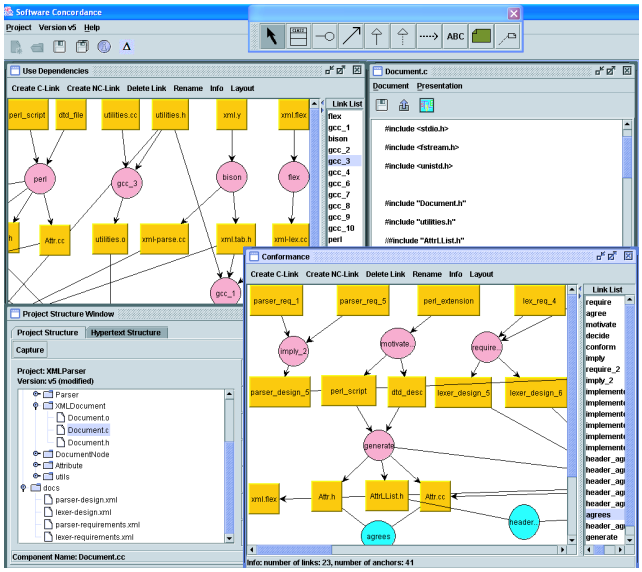


Figure 16: Multiple hypertext networks

edges for non-causal links. Services for links include link creation, deletion, renaming, attribute's value viewing, and link history viewing. Figure 15 shows the history of the link "agrees". Note that the link was not created until the version *v9.1.1.1*. Therefore, the earlier versions on the top window are "disabled". The constructor of the class "SCDocument" was displayed in the bottom window since the user clicked on the corresponding anchor. The version history of a hypertext network is displayed in the same manner. Figure 16 shows two hypertext networks. The one on top plays the role of a graphical makefile while the other shows a dependency network on the same set of components. Also, the versions of the graphical makefile are consistently maintained with those of the project.

Services for anchors are also provided at the hypertext network editing window. They includes deleting an anchor, adding an anchor into the active link, removing an anchor off some link, renaming an anchor, and displaying the structural unit that the anchor refers to (see Figure 14). If the user chooses to open the associated structural unit of an anchor, the editor of the corresponding component is invoked (such as structured Java program, XML, HTML,

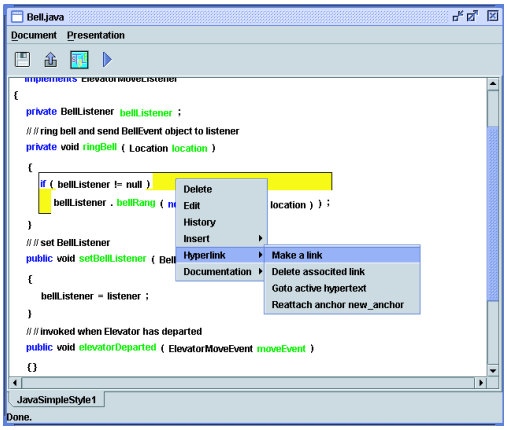


Figure 17: Component editor

SVG graphics, UML diagrams, or plain text editors). The editors are all hypertext-savvy and version-savvy. Figure 17 shows the structured editor for Java programs. When the user right-clicks on a document node, a popup menu is displayed to allow the user to create an anchor at the node and add it to the active hypertext network, or to open the active hypertext network of an anchor, or to relocate an anchor to a different document node, etc.

Molhado's product versioning model for hypertexts facilitates the construction of these graphical user interfaces (GUIs) for versioned hypermedia services. For example, users' traversal can be done via embedded HTML-style hyperlinks or via a hypertext network without involving users' selection of revisions of individual hypertext entities since the traversal occurs among components at the current version, which is implicitly and globally determined via user interface actions. Therefore, the user does not have to specify which revisions of hypertext elements they want to refer to. Similar to traversal, other hypertext operations such as deletion or insertion of anchors or links do not require any version selection rules. Molhado's hypertext versioning model also reduces the cognitive overhead for users in version creation of links, anchors, or nodes in a hypertext structure. When the user is ready to record the state of the project after modifying hypertext networks or components, a capture or a commit command can be issued and a new version will be created. The user does not need to check in or check out components or hypertext entities individually.

9. CONCLUSIONS

This paper describes Molhado, a new hypertext versioning and SCM system that are well-suited for software logical relationship management. It contributes a novel versioned hypermedia model and a unified structure versioning framework for components and hypertext structures among them. The model avoids the complications of version selection rules in composition model and the version proliferation problem in total versioning model. It also facilitates the development of a GUI for versioned hypermedia services, which reduce the cognitive overhead for users in version creation and version selection of hypermedia elements in hypertext operations. Software components and hypertext structures are uniformly versioned in a fine-grained manner, allowing users to return to a consistent previous state not only of a hypertext network but also of a single node and of a hyperlink.

Acknowledgments: We would like to thank Dr. Jim Whitehead at the University of California - Santa Cruz for his helpful comments and the sketch for figures 3a), 5a), 6, 8, and 10.

10. ADDITIONAL AUTHORS

Cheng Thao (Department of EECS, University of Wisconsin-Milwaukee, email: chengt@cs.uwm.edu)

11. REFERENCES

[1] Robert M. Akscyn, Donald L. McCracken, and Elise A. Yoder. KMS: a distributed hypermedia system for managing knowledge in organizations. *Communications of the ACM*, 31(7):820–835, 1988.

[2] Maurice Amsellem. ChyPro: A hypermedia programming environment for SmallTalk-80. In *Proceedings of ECOOP*, 1995.

[3] Kenneth M. Anderson, Richard N. Taylor, and E. James Whitehead, Jr. Chimera: hypermedia for heterogeneous software development environments. *ACM Transactions on Information Systems (TOIS)*, 18(3):211–245, 2000.

[4] Ulf Asklund, Lars Bendix, Henrik Christensen, and Boris Magnusson. The unified extensional versioning model. In *Proceedings of the Ninth International Symposium on Software Configuration Management, SCM-9*, pages 100–122. Springer, 1999.

[5] L. Bendix, Antonina Dattolo, and Fabio Vitali. Software configuration management in software and hypermedia engineering. *Handbook of Software Eng. and Knowledge Engineering*, 1, 2001.

[6] Lars Bendix and Fabio Vitali. VTML for Fine-grained Change tracking in Editing Structured Documents. In *Proceedings of the Software Configuration Management Workshop*. Springer, 1999.

[7] James Bigelow and Victor Riley. Manipulating source code in DynamicDesign. In *Proceedings of the Hypertext*, 1987.

[8] John T. Boyland, Aaron Greenhouse, and William L. Scherlis. The Fluid IR: An internal representation for a software engineering environment. In preparation. For information see http://www.fluid.cs.cmu.edu.

[9] Mark C. Chu-Carroll, James Wright, and David Shields. Supporting aggregation in fine grained software configuration management. In *Proceedings of the tenth Foundations of software engineering symposium*, pages 99–108. ACM Press, 2002.

[10] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Computing Surveys (CSUR)*, 30(2):232–282, 1998.

[11] R. Cronk. Tributaries and deltas. *BYTE*, pages 177–186, Jan 1992.

[12] Cybulski and Reed. A Hypertext Based Software Engineering Environment. *IEEE Software*, 9(2):62–68, March 1992.

[13] Delisle and Schwartz. Neptune: A hypertext system for CAD applications. In *Proceedings of SIGMOD*, pages 132–142. 1986.

[14] Norman M. Delisle and Mayer D. Schwartz. Contexts: partitioning concept for hypertext. *ACM Trans. Inf. Syst.*, 5(2):168–186, 1987.

[15] David Durand. *Palimpsest: Change-oriented concurrency control for the support of collaborative applications*. PhD thesis, Boston University – Boston, 1999.

[16] James C. Ferrans, David W. Hurst, Michael A. Sennett, Burton M. Covnot, Wenguang Ji, Peter Kajka, and Wei Ouyang. HyperWeb: a framework for hypermedia-based environments. In *Proceedings of the Symposium on Software Development Environments*, pages 1–10. ACM Press, 1992.

[17] Pankaj K. Garg and Walt Scacchi. A hypertext system to manage software documents. *IEEE Software*, 7(3):90–98, May 1990.

[18] Goldstein and Bobrow. *A Layer Approach to Software Design*. Interactive Programming Environments. McGraw-Hill, 1984.

[19] Jon Griffiths, David Millard, Hugh Davis, Danius Michaelides, and Mark Weal. Reconciling versioning and context in hypermedia structure servers. In *Proceedings of the 1st International Metainformatics Symposium*, 2002.

[20] Anja Haake. CoVer: a contextual version server for hypertext applications. In *Proceedings of the ACM conference on Hypertext*, pages 43–52. ACM Press, 1992.

[21] Anja Haake and David Hicks. VerSE: towards hypertext versioning styles. In *Proceedings of the seventh ACM conference on Hypertext*, pages 224–234. ACM Press, 1996.

[22] David L. Hicks, John J. Leggett, Peter J. Nrnberg, and John L. Schnase. A hypermedia version control framework. *ACM Transactions on Information Systems (TOIS)*, 16(2):127–160, 1998.

[23] Thomas Kejser and Kaj Gronbak. The GAIA Framework: Version Support In Web Based Open Hypermedia. In *proceedings of IADIS International Conference on WWW/Internet*, 2003.

[24] D. Leblang. The CM challenge: Configuration management that works. *Configuration Management*, 2, 1994.

[25] A. Lie, R. Conradi, T. Didriksen, E. Karlsson, S. Hallsteinsen, and P. Holager. Change oriented versioning. In *Proceedings of the Second European Software Engineering Conference*, 1989.

[26] Y. Lin and S. Reiss. Configuration management with logical structures. In *Proceedings of the Eighth International Conference on Software Engineering*, pages 298–307, 1996.

[27] Boris Magnusson and Ulf Asklund. Fine-grained revision control of Configurations in COOP/Orm. In *Proceedings of the Software Configuration Management Workshop*, pages 31–47. Springer, 1996.

[28] C. Maioli, S. Sola, and F. Vitali. Versioning for Distributed Hypertext Systems. In *Proceedings of ACM Conference on Hypertext*, 1994.

[29] Melly and Wendy Hall. Version control in Microcosm. In *Proceedings of the Workshop on the Role of Version Control in CSCW*, September 1995.

[30] Danius Michaelides, David Millard, Mark Weal, and D. DeRoure. Auld Linky: A contextual open hypermedia link server. In *Proceedings of the 7th Open Hypermedia System Workshop*, 2001.

[31] Dave E. Millard, Luc Moreau, Hugh C. Davis, and Siegfried Reich. FOHM: a fundamental open hypertext model for investigating interoperability between hypertext domains. In *Proceedings of the Conference on Hypertext*, pages 93–102. ACM Press, 2000.

[32] Tom Morse. CVS. *Linux Journal*, 1996(21es):3, 1996.

[33] Theodor Holm Nelson. *Literary Machines*. Mindful Press, 1987.

[34] Tien N. Nguyen and Ethan V. Munson. A model for conformance analysis of software documents. In *Proceedings of the International Workshop on Principles of Software Evolution*, 2003.

[35] Tien N. Nguyen and Ethan V. Munson. The Software Concordance: A New Software Document Management Environment. In *Proceedings of the 21th International Conference on Computer Documentation*. ACM Press, 2003.

[36] Kasper Østerbye. Structural and cognitive problems in providing version control for hypertext. In *Proceedings of the ACM conference on Hypertext*, pages 33–42, 1992.

[37] Kasper Østerbye. Literate SmallTalk using hypertext. *IEEE Transactions on Software Engineering*, 21(2):138–145, Feb 1995.

[38] Christoph Reichenberger. VOODOO: A Tool for Orthogonal Version Management. In *Proceedings of the Software Configuration Management Workshop, SCM-5*, pages 61–79. Springer, 1995.

[39] L.F.G. Soares, G.L.d. S. Filho, R.F. Rodrigues, and D. Muchaluat. Versioning support in HyperProp system. *Multimedia Tools and Applications*, 8(3):325–339, 1999.

[40] Norbert Streitz, Jorg Haake, Jorg Hannemann, Andreas Lemke, Wolfgang Schuler, Helge Schutt, and Manfred Thuring. SEPIA: a cooperative hypermedia authoring environment. In *Proceedings of the ACM conference on Hypertext*, pages 11–22. ACM Press, 1992.

[41] Walter F. Tichy. RCS - a system for version control. *Software - Practice and Experience*, 15(7):637–654, 1985.

[42] Andre van der Hoek, Dennis Heimbigner, and Alexander L. Wolf. A generic, peer-to-peer repository for distributed configuration management. In *Proceedings of the ICSE'96*. IEEE, 1996.

[43] Tim A. Wagner and Susan L. Graham. Incremental analysis of real programming languages. In *Proceedings of the 1997 ACM SIGPLAN conference on Programming language design and implementation*, pages 31–43. ACM Press, 1997.

[44] L. Wakeman and J. Lowett. *PCTE: the standard for open repositories*. Prentice Hall, 1993.

[45] E. James Whitehead, Jr. A proposal for versioning support for the Chimera system. In *Proceeedings of the Workshop on Versioning in Hypertext Systems*. ACM Press, 1994.

[46] E. James Whitehead, Jr. *An Analysis of the Hypertext Versioning Domain*. PhD thesis, University of California – Irvine, 2000.

[47] E. James Whitehead, Jr. WebDAV and DeltaV: collaborative authoring, versioning, and configuration management for the Web. In *Proceedings of the ACM conference on Hypertext and Hypermedia*, pages 259–260. ACM Press, 2001.

[48] Uffe K. Wiil and John J. Leggett. Hyperform: using extensibility to develop dynamic, open, and distributed hypertext systems. In *Proceedings of the Conference on Hypertext*. ACM Press, 1992.