A Hypermedia Version Control Framework

DAVID L. HICKS
Knowledge Systems
JOHN J. LEGGETT and PETER J. NÜRNBERG
Texas A&M University
and
JOHN L. SCHNASE
Missouri Botanical Garden Center for Botanical Informatics

The areas of application of hypermedia technology, combined with the capabilities that hypermedia provides for manipulating structure, create an environment in which version control is very important. A hypermedia version control framework has been designed to specifically address the version control problem in open hypermedia environments. One of the primary distinctions of the framework is the partitioning of hypermedia version control functionality into intrinsic and application-specific categories. The version control framework has been used as a model for the design of version control services for a hyperbase management system that provides complete version support for both data and structural entities. In addition to serving as a version control model for open hypermedia environments, the framework offers a clarifying and unifying context in which to examine the issues of version control in hypermedia.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution and Maintenance—*version control*; H.1.1 [**Models and Principles**]: Systems and Information Theory—*general systems theory*; H.2.1 [**Database Management**]: Logical Design—*data models*; H.2.4 [**Database Management**]: Systems—*distributed systems*; H.2.8 [**Database Management**]: Database Applications; H.3.4 [**Information Storage and Retrieval**]: Systems and Software; H.3.m [**Information Storage and Retrieval**]: Miscellaneous; H.5.1 [**Information Interfaces and Presentation**]: Multimedia Information Systems—*hypertext navigation and maps*

General Terms: Design, Management

Additional Key Words and Phrases: Hyperbase management system, hypermedia

This research was supported in part by the Texas Advanced Research Program under grant no. 999903-155.
Authors' addresses: D. L. Hicks, Knowledge Systems, RD2, Number 213A Evans Road, Export, PA 15632; J. J. Leggett and P. J. Nürnberg, Department of Computer Science, Texas A&M University, College Station, TX 77843; J. L. Schnase, Missouri Botanical Garden Center for Botanical Informatics, 660 South Euclid Avenue, Box 8132, St. Louis, MO 63110.

1. INTRODUCTION

Version control is the process of organizing, coordinating, and managing the development of evolving objects. In many application areas, the object evolution process can be characterized as a series of incremental refinements. For example, software developers continually make changes to software modules as errors are detected, and authors are seldom satisfied with the first draft of a manuscript, usually making several revisions before producing a final version. In each of these cases, objects under development are changed and updated in order to produce the next refinement in the evolutionary process.

In many development areas, the preservation of intermediate revisions of evolving objects is very important. As the number of revisions grows large, this task becomes difficult. Version control systems are automated tools that assist in the management of evolving objects [Tichy 1985], facilitating the tracking of intermediate revisions of objects and the relationships among them. For example, the version control facilities found in software engineering environments simplify the process of creating and reconstructing software systems, and, in an authoring environment, version control systems allow authors to easily return to previous drafts of their work.

Version control has been identified as a critically important area of research in hypermedia and one that must be addressed in future systems [Durand et al. 1995; Halasz 1988; Leggett 1993; Leggett et al. 1993].[1] The version control process in hypermedia involves maintaining version histories that support access to previous revisions of hypermedia objects—both data and structure—and automatically tracking the derivation information for those objects. Many of the application areas to which hypermedia technology has been applied have important requirements for versioning data that hypermedia systems must accommodate. For example, to serve as an effective environment for the software development process, a hypermedia system must provide version support for software modules. The capabilities that hypermedia provides for easily creating and manipulating structure allow it to evolve very rapidly. The highly dynamic, lightweight nature of hypermedia structure significantly increases the importance of version support for structural entities. As the variety of potential applications for hypermedia technology continues to expand, the lack of version control will become increasingly apparent and limiting.

The version control problem in hypermedia has been complicated and broadened by the introduction of open hypermedia architectures [Leggett 1993]. It is no longer sufficient to address the version control problem within the boundaries of an individual system. To completely address the version control problem in hypermedia, new approaches based on the storage management components of these new architectures, often called Hyperbase Management Systems (HBMSes), must now be considered. The hypermedia version control framework developed in this article was specif-

[1]Also mentioned by F. Halasz (Hypertext '91 Keynote Address, San Antonio, Tex., Dec.)

ically designed to accommodate the needs of an open hypermedia environment that includes a hyperbase management system [Leggett and Schnase 1994]. Section 2 introduces the framework along with a discussion of the major influences upon its design. The description continues in Section 3 with more detailed information on the framework and an examination of how it addresses hypermedia version control issues. In Section 4, integration of the framework into a hyperbase management system is presented along with a discussion of the version control environment it creates. Section 5 examines the implications of the framework for version control in hypermedia. The related literature is reviewed in Section 6, followed by a brief discussion of future research in Section 7, and a summary in Section 8.

2. A HYPERMEDIA VERSION CONTROL FRAMEWORK

A version control framework for an *open hypermedia environment* must support a diversity of versioning paradigms in order to accommodate the variety of development methodologies found among client applications. The wide variety of client applications in the open hypermedia environment motivates the specific design goals of flexibility, extensibility, and scalability for the framework. In order to support a variety of different development methodologies, version control services should be flexible, unconstrained by any specific development paradigm. Since the range of HBMS client applications is not known in advance, version control services should be extensible to accommodate new types of applications. Because the amount of version control support required varies across application boundaries, version control services should be scalable, allowing each application to utilize the appropriate amount of version control functionality. Additionally, the presence of version control services should not impede the use of the hypermedia system by applications that do not use versioning [Østerbye 1992].

This section introduces a hypermedia version control framework, beginning with an overview of the hypermedia model that influenced its design. Next, an analysis of the version control process and an examination of version control functionality are provided as background material for the overview and description of the specific version control framework that follows.

2.1 The HURL Process-Based Hypermedia Model

In the development of the version control framework reported in this article, emphasis was placed on enabling the framework to fully support the versioning of structural entities. In order to accomplish this, it is imperative that the hypermedia model completely abstract structural entities from data and behavioral entities [Leggett and Schnase 1994; Schnase et al. 1994].

The HURL process-based hypermedia model provides an open and extensible hypermedia framework. As illustrated in Figure 1, the model is

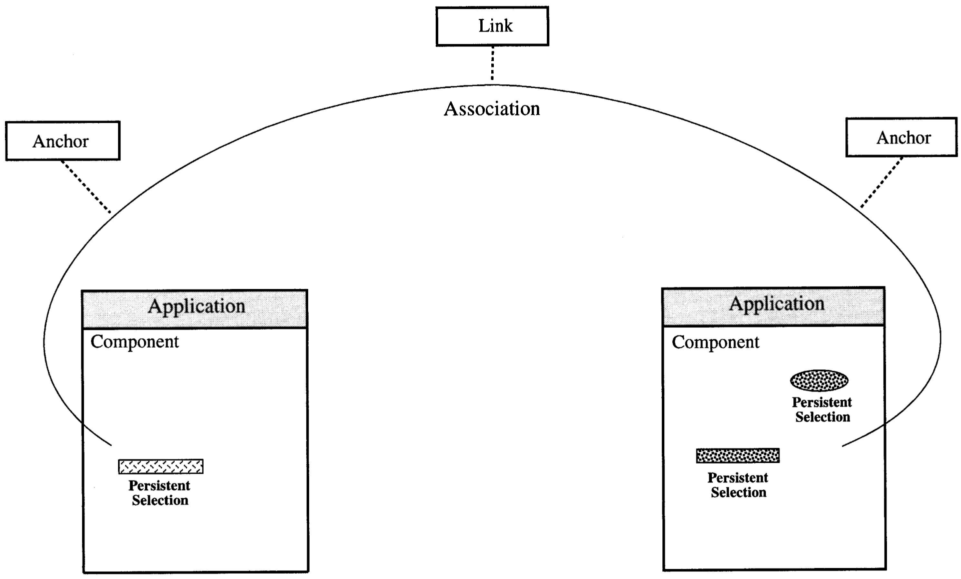
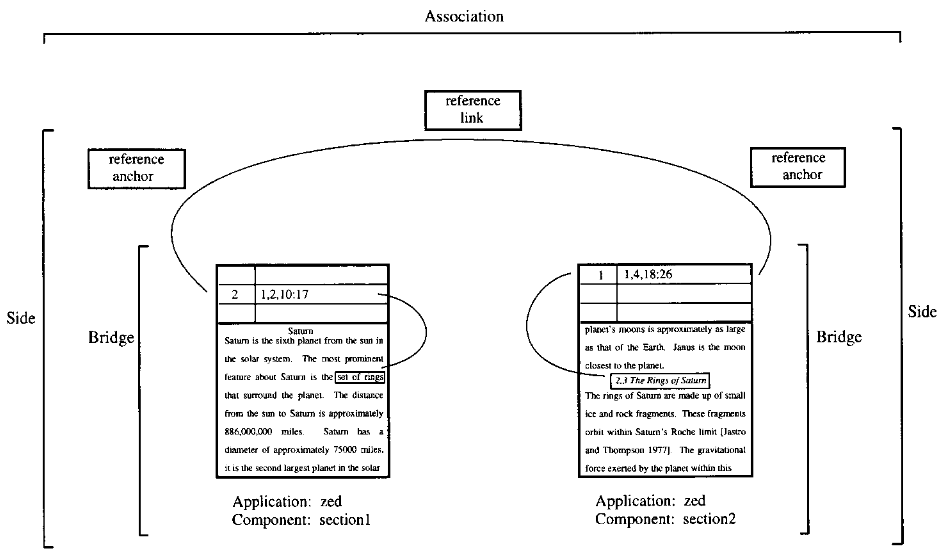


Fig. 1. The HURL process-based hypermedia problem.

composed of six basic entities: persistent selections, components, applications, anchors, links, and associations. An *application* in this model corresponds to a program. A *component* is a collection of data operated on by an application. A *persistent selection* is a selection defined within a component that persists across invocations of the application. For example, a simple text editor represents an application in the model. The ASCII files that the editor manipulates are components. A persistent selection for this application could be a highlighted string of text that persists across invocations of the editor.

Anchors and *links* in the model allow the specification of hypermedia behaviors. Anchors and links both are processes capable of arbitrary actions. In the example shown in Figure 1, the anchors and links all have basic reference behavior. The anchors reference persistent selections that are defined within application components, and the link references the associated anchors. In general, however, arbitrary anchor and link processes can be defined that implement a variety of hypermedia behaviors. The remaining element of the model, the *association*, is a structural representation mechanism. An association can be thought of as a collection of identifiers that defines a connection. Unlike anchors and links, an association represents connectivity information and does not implement behavior. The arc connecting the objects in Figure 1 corresponds to an association. Associations may be collected in *association sets* (discussed in Section 3.5).

In practice, the definition of additional constructs is helpful in the design and implementation of hypermedia services based on this model (see Figure 2). Specifically, two additional structural abstractions are useful. The first



Saturn
Saturn is the sixth planet from the sun in the solar system. The most prominent feature about Saturn is the set of rings that surround the planet. The distance from the sun to Saturn is approximately 886,000,000 miles. Saturn has a diameter of approximately 75000 miles. it is the second largest planet in the solar

planet's moons is approximately as large as that of the Earth. Janus is the moon closest to the planet.

2.3 The Rings of Saturn

The rings of Saturn are made up of small ice and rock fragments. These fragments orbit within Saturn's Roche limit [Jastro and Thompson 1977]. The gravitational force exerted by the planet within this

Fig. 2. Mapping of simple hypermedia connection to HURL model abstractions.

is a mechanism that spans the intranode and internode referencing domains involved in the specification of endpoints for hypermedia connections. A *bridge* is defined as a triple consisting of a persistent selection identifier, a component identifier, and an application identifier. The persistent selection identifier specifies a particular location within an object that corresponds to the endpoint of a hypermedia connection (the intranode level of specification). The component identifier indicates the object that contains a particular endpoint (the internode level of specification). The application identifier of a bridge triple indicates the application responsible for managing the component.

The second additional structural abstraction is a mechanism that serves as a building block for the definition of associations. A *side* is defined as a collection of one or more bridge identifiers along with an anchor identifier. It completely defines one side of an association. The bridge identifiers specify a set of connection endpoints for the side. The anchor identifier specifies behavioral information by indicating the anchor process affiliated with this side of an association.

A simple hypermedia connection is illustrated in Figure 2. The connection is defined between two highlighted text segments contained in separate components. Figure 2 also shows how the connection can be mapped to the structural abstractions of the HURL model. For example, the triple (2, section1, zed) represents a bridge object for the hypermedia connection endpoint shown on the left side of the figure. It specifies that the persistent selection labeled "2" in the component named "section1" serves as an endpoint for the connection and that the zed editor is the application responsible for managing that component. The triple (1, section2, zed) represents a bridge object for the other hypermedia connection endpoint shown in the figure. The anchors and the link involved in the connection all have standard reference behavior.

While this simple linking example does not demonstrate the extensive representational capabilities of the HURL model, it does illustrate how the model completely abstracts structure from data and behavior. Structural abstractions exist as independent objects, separate from the data and behavioral information they reference.

2.2 Analysis of Version Control

At its most fundamental level, the version control process involves supporting the development of evolving objects. This includes preserving intermediate states of objects as they evolve across time, tracking their derivation histories, and controlling storage requirements. These basic operations support the fundamental object evolution process as it occurs at the physical storage level. These operations are intrinsic to the version control process.

An important dimension, along which version control systems vary, is the degree of similarity between the version control services provided and intrinsic version control operations. The degree of similarity is primarily determined by the development methodology a system is intended to support. Basic, low-level development methodologies have a direct correspondence to intrinsic version control operations. For example, in many software configuration management systems, there is a direct mapping between the concept of a software module and an underlying file which represents the module. Operations supported in the system's development paradigm, such as create a new revision, have a direct mapping to corresponding intrinsic version control operations. Object evolution, as perceived by the user of this type of system, corresponds directly to object evolution as it occurs at the physical-storage level [Tichy 1985].

More recent version control systems are designed to support specialized, high-level development paradigms [Dart 1991; Goldstein and Bobrow 1984; Haake and Hicks 1996; Katz 1990]. High-level development methodologies often have no direct correspondence to intrinsic version control operations. The development processes they are intended to support have been abstracted from physical storage-level object evolution. Abstraction enables these methodologies to accommodate more elaborate evolution patterns that can better model development in a specific application area. For example, a complex design object in a CAD system might actually consist of data from several physical storage-level objects [Batory and Kim 1985]. As development proceeds, the CAD system maps object evolution as it occurs within its high-level development paradigm into the appropriate intrinsic version control operations. In this type of system, the object development process, as perceived by the user, does not directly correspond to physical-object evolution.

High-level development methodologies also differ within the same application area, reflecting different approaches for supporting the development process. For example, in the software development area, the Personal Information Environment (PIE) provides enhanced support for the definition of software design alternatives [Goldstein and Bobrow 1984], while Gandalf uses an inverted approach toward version control in order to provide extended transaction management facilities [Miller et al. 1989]. Although both of these systems are intended to support the same area of application, their facilities differ, each having been abstracted from the physical-object evolution process in order to expand support for a particular aspect of the software development process.

This analysis suggests that the majority of similarities found across a wide range of development methodologies are confined to lower-level version control operations. High-level development methodologies lack generality. Their services are optimized to support a particular style of development in a specific application area and do not directly share much in common with the more basic development methodologies. However, the facilities offered within systems based on high-level development methodologies must, at some point, be mapped to physical storage-level operations. It is at this lower level, the level of intrinsic version control operations, where overlap across the range of development methodologies occurs.

2.3 Partitioning of Functionality

Figure 3 illustrates the typical relationship between an HBMS and its client applications. Applications interact with the HBMS through the application interface to store and retrieve objects or perform hypermedia operations. The HBMS channels requests from applications to object and structure management subsystems which interact with a physical-storage manager to perform the desired operations.

The design goals of the version control framework, combined with the characteristics of the HBMS environment, motivate a partitioning of version control functionality. Following the analysis of the previous section, version control functionality is partitioned in the framework into two major categories: intrinsic version control operations and application-specific functionality.

Intrinsic version control operations are assigned to the HBMS level of the framework. Implementation at this level enables HBMS client applications to easily utilize this functionality. High-level, application-specific version control functionality is assigned to the application level of the framework. Building on the intrinsic version control platform provided at the HBMS level, client application developers can create customized version control services tailored for their specific application domain.

The partitioning of version control functionality is an important concept in the development of the version control framework. The assignment of intrinsic version control functionality to the HMBS level enables the maximum amount of version control functionality possible to be offered

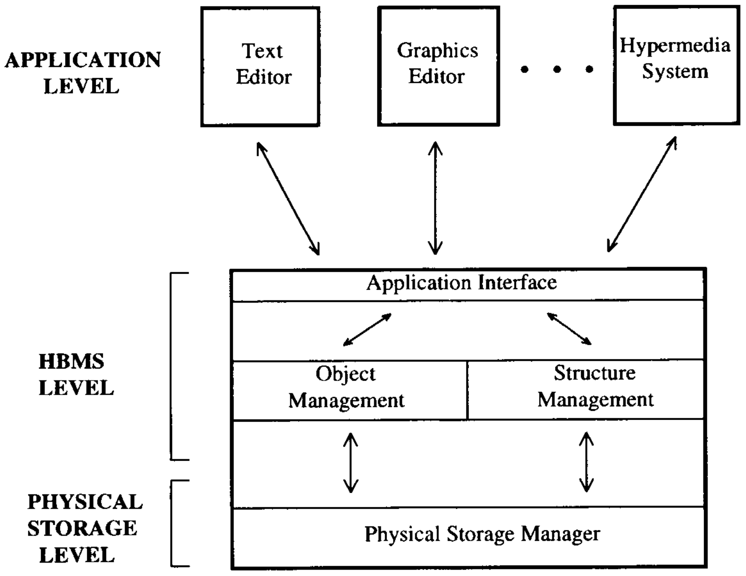


Fig. 3. Relationship between hyperbase management system and client applications.

directly within the framework. The inclusion of additional functionality at this level would begin to incorporate application-specific version control operations. This would not maintain the generality required in the HBMS environment. The assignment of less functionality to this level would exclude some of the intrinsic version control operations and would not represent the largest common subset of version control operations possible across a range of development methodologies.

2.4 Overview of the Framework

The description of the framework provided in this subsection reflects the partitioning of functionality that influenced its design. It is organized into two parts, one describing version control functionality at the HBMS level of the framework, the other discussing version control at the application level.

2.4.1 *Version Control at the HBMS Level.* Version control functionality at this level is provided in the form of a hypermedia version server. The version server provides a platform of basic hypermedia version control operations. The relationship between the version server and the HBMS environment is illustrated in Figure 4. Applications interact with the version server to access data and structural objects stored by the HBMS. The version server, in turn, interacts with the HBMS to perform the desired operation.

The version server provides a ubiquitous versioning environment at the HBMS level. A version history is maintained for all objects managed by the

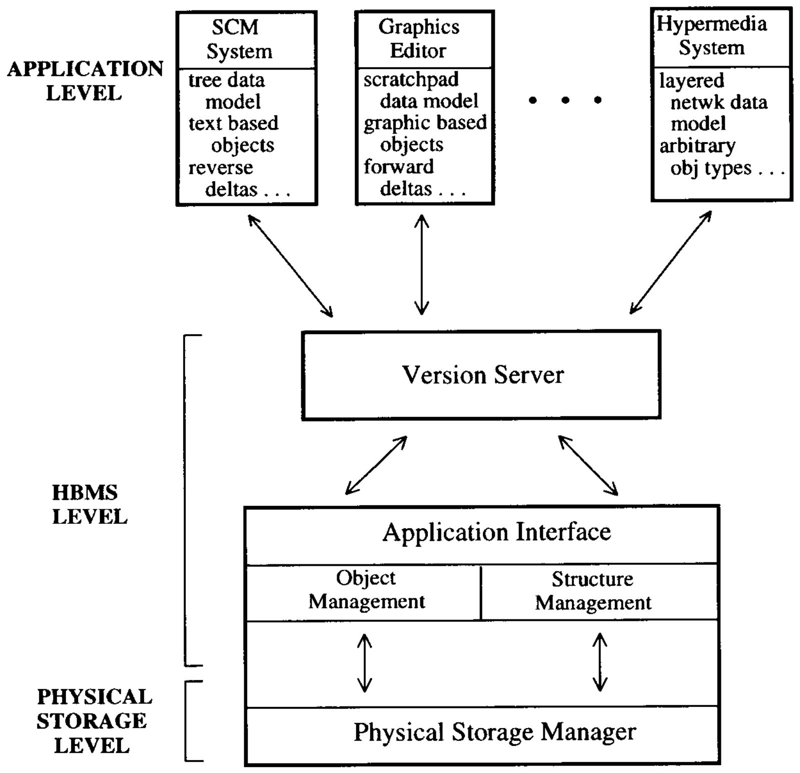


Fig. 4. Relationship among the hyperbase management system, the version server, and client applications.

HBMS, both data and structural. Every object has an associated version set history. A version set history is a container that manages all of the individual revisions of an object, maintaining the derivation history of individual revisions along with other type-specific information. The version server uses a graph-based version control data model to track the relationships of revisions within version set histories. A delta storage algorithm is used to reduce storage requirements.

An extensive set of attributes is maintained for each object. The set of object attributes is expandable to accommodate the definition of new object types for newly integrated applications. Object attributes are organized into two categories: standard system attributes common to all objects and type-specific attributes. Attributes can be used for object identification within the framework. They can be used to define static references that always refer to the same version of an object or dynamic references that change as new versions of an object are created. A unique identifier, assigned to each object when it is created, can also be used for object identification.

The functionality provided by the version server is made accessible to HBMS client applications through a set of modular object manipulation functions. In addition to standard object management operations, version control functions are available that allow new revisions of objects to be created. The design of the version server is not intended to reflect any specific versioning paradigm or strategy. Applications have complete control over the frequency and granularity of version control operations [Katz and Chang 1987; Katz et al. 1986; Prevelakis 1990]. New revisions of objects are created only when specifically requested by applications. The version server is intended to provide a mechanism only, not policy.

2.4.2 *Version Control at the Application Level.* The version control process is not rigidly defined at the application level of the framework. The complete set of version control operations of the HBMS-level version server is available for the creation of application-level version control facilities. HBMS-level version control services can be used directly to provide a basic version control facility at the application level. When more elaborate version control facilities are required, basic HBMS-level version control operations can be used as building blocks to construct a specialized application-level version control mechanism.

Figure 4 illustrates the diversity of version control services that can exist at the application level of the framework. A number of applications are shown that implement their own version control policies. The Software Configuration Management (SCM) system provides a basic version control service that uses a tree data model to support object evolution for software modules. It can use the HBMS-level version services directly to implement its versioning functionality. The Hypermedia System provides a more elaborate version control service based on the layered network version control data model [Goldstein and Bobrow 1984; Prevelakis 1990]. It builds upon the basic HBMS-level version services to construct the layered network versioning abstraction.

3. DISCUSSION OF THE HYPERMEDIA VERSION CONTROL FRAMEWORK

This section discusses how the version control framework addresses the important issues of version control in hypermedia. The discussion includes specific design decisions along with design rationale. Several issues are considered in this section. The issues are organized into the following categories: version control data model, endpoint specification, capturing structural evolution, storage management, composition mechanism, scope of versioning, and user interface considerations.

3.1 Version Control Data Model

The data model in a version control system describes the data entities allowed in the system and the relationships that can exist among them [Hicks 1993]. This dictates the paths of evolution that a version control system can support. Several data models have been suggested for version control systems including the linear, tree, and layered network models [Goldstein and Bobrow 1984; Leblang and Chase 1987; Rochkind 1975].

Reflecting the range of areas to which version control techniques have been applied, a widespread consensus on a single, optimal version control data model has not yet developed. Although the tree model is a very popular version control data model, additional models continue to be introduced as version control technology spreads to new application areas [Thomas 1989]. Current version control data models can be categorized into two major groups: basic models, such as the linear and tree models, and abstract models, such as the layered network model [Hicks 1993]. Basic models support simple object evolution patterns that are representative of the physical-object derivation process. Abstract data models do not necessarily correspond to physical-object evolution. Their facilities are designed to support evolution at a higher level.

In the framework two levels of the version control data model are identified, one at the HBMS level and another at the application level. At the HBMS level, a data model is needed that can support intrinsic version control operations. It should be able to model the physical-object derivation process. At the application level, the version control data model must be capable of modeling application-specific development paradigms. It defines how the object evolution process is presented to the user of a system.

The directed acyclic graph (DAG) model, which is based directly on the tree model, is used by the hypermedia version server at the HBMS level of the framework. The DAG model was selected for its ability to accommodate physical-object evolution patterns [Dittrich and Lorie 1988; Tichy 1988]. A more elaborate version control data model would not be appropriate for the HBMS-level version server. The facilities of abstract models cannot, in general, be easily and efficiently adapted and used to support alternative development methodologies. In contrast, basic version control data models such as the DAG provide a flexible, solid platform capable of supporting the intrinsic version control operations that form the basis of higher-level version control data models.

At the application level of the framework, developers are free to select the most appropriate version control data model for their particular application. In the earliest version control systems, the data model presented at the user interface was often the same model used for tracking physical-object evolution. As new development paradigms have emerged, systems have appeared in which the two models differ, in many cases significantly. In these systems, the data model presented to the user has been abstracted from the physical-object derivation process in order to enhance support for a particular development methodology.

The separation of the HBMS-level and application-level data models enhances the ability of the framework to readily accommodate both types of systems. The DAG version control data model provided at the HBMS level by the version server completely supports the physical-object evolution process. Its capabilities can be used directly in applications that are based on basic development methodologies. In systems based on high-level development methodologies, developers can exploit the intrinsic version control capabilities provided by the HBMS-level data model as they map their application-specific version control data models to physical storage operations.

3.2 Endpoint Specification

The granularity of endpoint specification supported in most hypermedia architectures allows connections to be forged between specific regions *within* nodes. To completely define this type of endpoint, two levels of specification are required: the *internode* and *intranode* levels. The internode level resolves an endpoint reference to the node level of granularity. It identifies the specific node or nodes that comprise a connection endpoint. At the intranode level, a location or region within the target node is designated as the precise endpoint of a hypermedia connection.

In a versioned environment, both levels of endpoint specification are complicated. Since multiple revisions of nodes are likely to exist, internode references must contain additional information to enable a specific revision of a node to be identified. At the intranode level, references can easily become corrupt as new revisions of nodes are produced when changes are made to locations that correspond to hypermedia connection endpoints.

In the hypermedia version control framework, the endpoint specification issue concerns both the HBMS and application levels. Internode referencing occurs at the HBMS level, since it involves the identification of objects contained within the hyperbase. Intranode referencing requires the identification of regions within hyperbase objects. Since the internal contents of hyperbase objects vary across application boundaries, intranode referencing is an application-level issue in the framework.

3.2.1 *Internode Referencing*. Internode referencing is actually a special case of the more general object identification problem in a versioned environment. Versioning complicates object identification for all tasks in the HBMS environment. In the framework, there are two mechanisms available for object identification: object identifiers and attributes. Each revision of every object is assigned a unique identifier when it is created. Unique identifiers provide a permanent object identification mechanism that is completely independent of object data values. Unique identifiers are unaffected by changes to object content or attribute values. This provides a convenient mechanism for the specification of static object references [Khoshafian and Copeland 1986]. It also eliminates the problems that can arise in a versioned environment when object names are tightly coupled to their derivation history [Cohen et al. 1988].

Alternatively, object specification can be done with object attributes. The version server provides an extensive set of attributes for every object managed by the HBMS. Two levels of attributes are maintained for every object: version set history attributes and individual revision attributes. Version-set-history-level attributes, such as “object-type,” apply to all revisions of the object while revision-level attributes, such as “creation-date,” apply to only a single revision. Version-set-history-level attribute values are constant for all revisions of an object; revision-level attribute values vary with every instance [Ahmed and Navathe 1991].

Attributes can be used either statically or dynamically for object specification. To statically identify an object, a collection of attribute values can be specified that directly identifies a specific revision. To reference an object dynamically, attribute values can be used as input to a predicate operation that selects a specific revision of the object. This capability allows dynamic object specification such as "most recent revision" or "latest revision with status of released." When an attribute value specification does not contain sufficient information to select a specific revision of an object, defaults are used to guide the selection process.

The combination of unique identifiers and flexible object attributes provides a powerful object specification facility for use in hypermedia structural entities, composite objects, object management operations, and object identification in general (see the following sections).

3.2.2 *Intranode Referencing.* Intranode referencing involves the identification of a region or object within a node as the precise endpoint of a hypermedia connection. As indicated earlier, this requires specific knowledge of the internal format of a node. In the version control framework, applications are completely responsible for managing the data contained within their own components. Intranode referencing is an application-level issue.

Various strategies have been suggested for the specification of intranode references. Character offsets are often used to identify endpoints within text nodes [Delisle and Schwartz 1987]. Object identifiers are often used to specify endpoints in object-based systems [Garrett et al. 1986; Yankelovich et al. 1988]. The most appropriate strategy depends on the application and the types of objects it manipulates. The separation of endpoint specification into the internode and intranode levels allows this issue to be addressed on an individual application basis, enhancing the ability of the framework to accommodate openness and extensibility in hypermedia environments [Leggett and Schnase 1994].

3.3 Capturing Structural Evolution

Versioning lightweight, extremely dynamic structure is one of the major challenges that hypermedia brings to research on version control. Structure in hypermedia can change as quickly as the data it references, leading to important versioning requirements. In addition to allowing users to return to previous revisions of rapidly evolving structural entities when necessary (an important capability), a structural versioning mechanism in hypermedia offers additional advantages as well. For example, creating snapshot images of both the data *and* structure defined over the argumentation space of an Issue Based Information System enables the complete historical record of its development to be captured and later analyzed, including information encoded in the structure of the system, such as when a user changes his or her position on an issue from "skeptical" to "convinced." The ability to version structure can also serve as the basis for the development of important related capabilities, such as allowing multiple simultaneous views to exist over a common set of hypermedia nodes. This capability enables individuals to personalize an information space as their perceptions develop over time.

An important aspect of versioning structure is the decision regarding which objects need to be versioned in order to capture the evolution of individual hypermedia connections. In an environment based on a limited hypermedia model, the ability to version structural items may be restricted. For example, if structure is not abstracted from data, it may not be possible to version individual links or anchors independently of the nodes they reference [Berners-Lee et al. 1992].

In the version control framework, structure exists independently of the data it references. The framework is designed to support the versioning of all structural entities. The structural abstractions in the HURL hypermedia model (associations, sides, and bridges) can each be represented with sets of identifiers. These structural abstractions exist as independent entities and can be stored and versioned as separate objects. As the structure over a set of objects evolves, new revisions of the corresponding structural entities can be created as needed to capture structural evolution.

An important consideration related to structural versioning is the effect that versioning data will have on structural entities. When a new version of a node (data) referenced by a link (structure) is created, what should the effect be on the link? Should it continue to reference the same version of the node or automatically be updated to point to the new version? In the version control framework, the answer to these questions is completely determined by the type of internode object reference used. The internode object references contained in structural entities can be static or dynamic. For example, the creation of new revisions of application components has no effect on a bridge structural abstraction containing static internode object references. However, if the internode reference is a dynamic one, such as "latest revision," the new revision will affect the bridge, causing it to automatically reference the new revision of the component. The references contained within other structural entities can also be defined statically or dynamically, and the creation of new revisions of the objects they reference has similar effects.

It is important to emphasize that structural entities in this framework are simply considered to be another type of object. Like all objects, structural entities are not automatically versioned as they change. As changes are made to a structural entity, it must be explicitly versioned for its derivation history to be preserved. Clients completely control the versioning of structural entities by deciding which structural changes are significant enough to warrant new revisions of structural objects. For example, associations might be versioned with every minor change made to a connection endpoint in one application, while in another only major changes produce new revisions. This flexibility enables the application developer to select the granularity of versioning that most appropriately captures structural evolution in a given application domain.

3.4 Storage Management

Storage management is an obvious concern in a versioned environment. The presence of version control services can significantly increase the number of objects managed by a system. The version server at the HBMS level of the framework uses a separate delta storage strategy to help control storage requirements [Tichy 1985]. Each object type can have its own delta calculation algorithm [Reichenberger 1991]. Deltas can be calculated and stored in forward or reverse order. If necessary, specifically designated revisions of objects can be stored intact. This facilitates rapid access to frequently used revisions. When no delta calculation algorithm is available for an object type, all revisions can be stored intact by default [Cohen et al. 1988; Tichy 1988].

3.5 Composition Mechanism

As version control services are introduced into an environment, the number of objects will increase significantly. The ability to organize objects into groups eases the object management process and can serve as the foundation for the development of several important hypermedia object management tools [Wiil 1993]. The framework provides an object-grouping mechanism called a composite. A composite is a special type of object whose data or information field consists of references to other objects. The set of object references for a composite object identifies the constituent objects of the composite. Since composites include their constituent objects by reference, rather than physically partitioning the object space, objects can directly participate in more than one composite. Object references within composites can be specified with object identifiers or attribute values and can either be static or dynamic.

Complete version histories are maintained for composite objects. However, no predetermined policies exist to control their versioning. New revisions of composites are created only in response to application requests, and applications are free to implement any required versioning strategy. Composites can be versioned when changes are made to their set of constituent object references, when changes are made to the constituent objects themselves, or at any point required to implement a particular versioning paradigm.

Composite objects provide a very flexible and convenient mechanism for grouping individual revisions of hypermedia objects. Composites can be created that contain only structural information, such as a collection of hypermedia associations. This capability enables individual users to maintain personal views of the structure over a set of hypermedia nodes [Yankelovich et al. 1988]. Composites can also include data objects as well as structural objects. This capability enables composites to serve as a

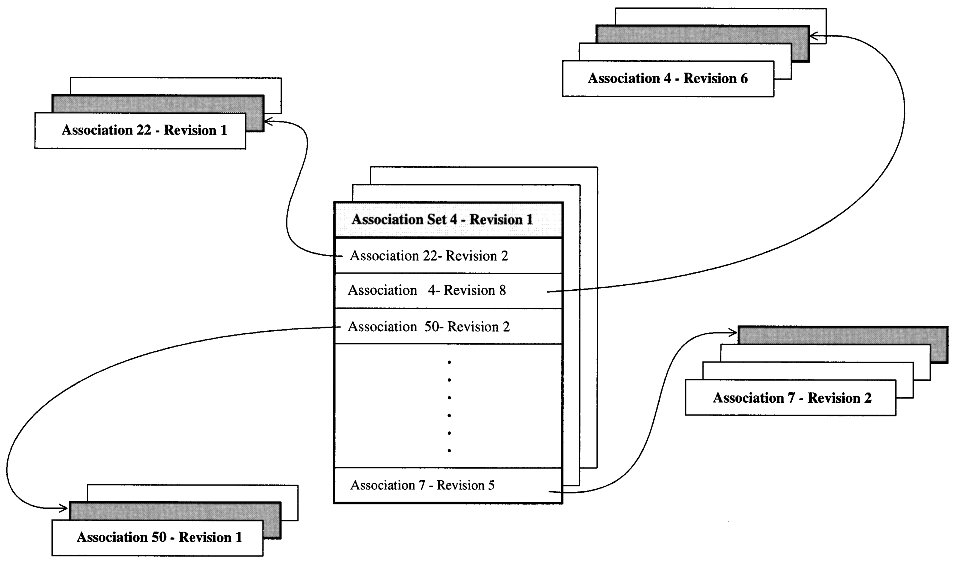

Association 22 - Revision 1
Association 4 - Revision 6
Association Set 4 - Revision 1
Association 22- Revision 2
Association 4- Revision 8
Association 50- Revision 2
Association 7 - Revision 5
Association 7 - Revision 2
Association 50 - Revision 1

Fig. 5. Versioning structure in the hypermedia version control framework.

logical partitioning mechanism for the hyperbase [Delisle and Schwartz 1987].

Composite objects have a significant effect on versioning structure within the version control framework by allowing structural versioning to occur at multiple levels. Structure can be versioned at the association level as described earlier. Versioning structure at this level tracks the derivation histories of individual hypermedia connections. Through the use of composites, structure can also be versioned at a higher level. By organizing a related group of associations into a set that is represented as a composite, the evolution of structure can be captured at the *association set* level as well. This enables the tracking of a coordinated set of changes made to a related group of structural objects.

Figure 5 illustrates the ability of the framework to version structure at multiple levels. Several individual associations are shown with their version histories. An association set composite object is also shown that is being used to organize association objects. The composite object contains references to specific revisions of several of the associations. Like all hyperbase objects, the composite itself is also being versioned. This arrangement allows structure to be versioned simultaneously at two levels: the individual association level and the association set level. The evolution of individual hypermedia connections is captured, as well as the evolution of the structure as a whole.

It is important to note the simplifications made to Figure 5 for illustrative purposes. First, the version histories shown for each of the objects would actually be DAGs. Second, the references shown in the composite object are all to structural objects. Composite objects can also contain references to data objects. Third, the references in the composite object are all static. Composites can also contain dynamic references. Finally, each association is actually a composite that references specific revisions of sides, anchors, and links, and sides are actually composites that reference specific revisions of applications, components, and persistent selections. The capability for versioning structure at multiple levels provides a comprehensive and extremely flexible structural versioning environment for client applications.

3.6 Scope of Versioning

The scope of versioning determines which objects will be versioned and varies widely in different systems. In some systems, certain types of objects are not eligible for version support [Østerbye 1992; Østerbye and Nørmark 1994]. Other systems divide the object base into classes of versioned and nonversioned objects and allow transformations between the classes [Ahmed and Navathe 1991; Haake 1992; 1994]. An explicit command is required in these systems to transform a nonversioned object into a versioned one.

In the framework all objects are versioned, both data and structural. This enhances the ability of the framework to readily support a variety of data and structural versioning paradigms and simplifies interaction with object management services. No interaction is required by applications to request version support for any object. Applications simply start issuing requests for new revisions to be created as needed.

The object management operations provided in the framework are specifically designed to accommodate all applications, regardless of whether they require version control services. These operations are low level and modular, allowing applications complete control of the frequency of version control operations. Through the use of appropriate object management operations and standard version set history defaults, version server functionality can be effectively masked if necessary, simplifying interaction with applications that do not require version support [Østerbye 1992].

3.7 User Interface Considerations

As an application is designed, several important decisions must be made regarding how and when version control operations are to be performed. These decisions are very important, as they collectively determine how users will interact with the version control portion of a system. Much of the variation in the version control functionality of existing systems is actually caused by differences in user interface policies.

Example user interface consideration issues include the process, granularity, and frequency of version creation and the immutability of versions [Hicks 1993]. Although these are all very important issues, the most appropriate policy for each is completely dependent upon the application for which the system will be used. For example, in a text editor application it might be appropriate to create new object revisions automatically and transparently as objects evolve, while in a software configuration management system it might be more appropriate to require explicit user interaction for version creation. User interface issues must be considered carefully. The policies addressing these issues have a direct impact on important related issues such as version proliferation [Ahmed and Navathe 1991; Rumbaugh 1988].

In the framework described here, user interface decisions are all application-level issues. The version control *mechanism* has been separated from the version control *policy*. A comprehensive set of basic version control functionality is provided by the HBMS-level version server. It is left to application designers to decide how this functionality can be used most effectively to implement the required application-level version control policies.

4. IMPLEMENTATION OF THE HYPERMEDIA VERSION CONTROL FRAMEWORK

A series of research prototype HBMSes has been developed at the Hypermedia Research Lab at Texas A&M University [Schnase 1992; Schnase et al. 1993]. This section describes the HB3 HBMS and the influence of the version control framework upon its design [Hicks 1993].

4.1 HB3 Overview

The HB3 hyperbase management system provides hypermedia and object management services for its client applications. It is based directly on its predecessor system, HB2 [Schnase 1992]. Figure 6 illustrates the HB3 system. It consists of four principal subsystems: storage manager (SM), object manager (OM), version server (VS), and association set manager (ASM).

The OM subsystem provides basic object management services. To facilitate the needs of the version control framework, OM provides two types of basic objects: simple and composite. A simple object has a content field that consists of an arbitrarily sized byte string. A composite object has a content field that contains references to other objects. As specified in the framework, both types of objects have a set of system-defined attributes and additional attributes that can be defined on an individual object type basis. The OM assigns a unique identifier to every object as it is created. The services offered by the OM are mapped into physical storage facilities by the SM.

The VS subsystem, building on the facilities of the OM, provides the functionality assigned to the HBMS level in the version control framework. It transforms the basic object management operations of the OM into a comprehensive suite of intrinsic hypermedia version control operations. As indicated in the figure, the VS is positioned so that all object management requests pass through it. This enables VS to enforce version-control-related integrity constraints and ensures that its facilities are available to all clients of object management services.

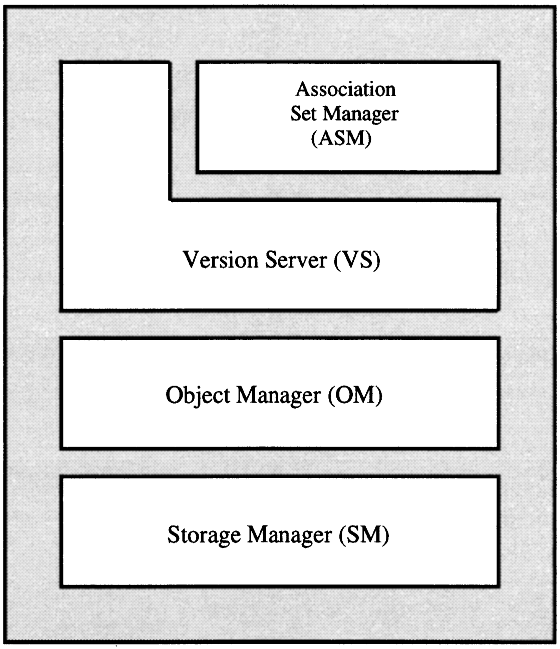


Fig. 6. The HB3 hyperbase management system.

The ASM subsystem manages the structural information required to represent hypermedia connections. It supplies the structural abstractions necessary to support the HURL hypermedia model described earlier.

Figure 7 illustrates the relationship between the HB3 HBMS and its surrounding SP3 hypermedia environment. The link services manager (LSM) is a server process that provides runtime support for hypermedia activities. LSM coordinates communication of structural information between client applications and the ASM to support activities such as defining hypermedia connections and performing hypermedia operations. As indicated in the figure, the version control and general object management services offered by the HB3 system can be used directly by client applications. As described later, these services are also available to manager applications such as LSM and ASM.

The components of HB3 and SP3 have been implemented in the Unix environment. The subsystems are written in C and run under SunOS on the Sun4 platform. VS, OM, ASM, and LSM are each implemented as individual server processes. The functionality of SM is provided by Postgres [Stonebraker and Kemnitz 1991], an extended relational database management system.

4.2 HB3 Versioned Object Manager

As depicted by the dashed box in Figure 7, the VS and OM are collectively referred to as the versioned object manager (VOM) in HB3. This is the view of the system presented to client applications. This subsection provides

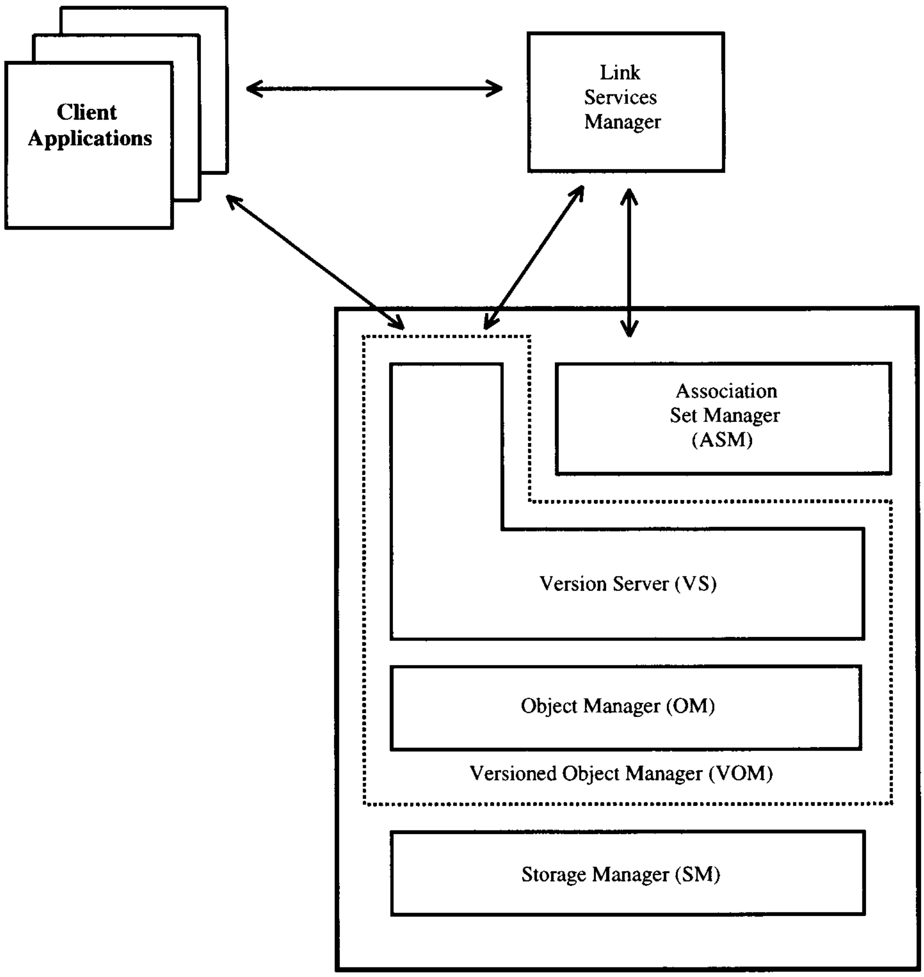


Fig. 7. The HB3 hyperbase management system in the SP3 hypermedia environment.

further detail on the specific services provided by the VOM and how these services are made available to client applications.

4.2.1 *Version Set Histories and Attributes.* The version set history is the principal data structure used by the VOM to track object evolution. Every object managed by the HB3 system has an associated version set history. The version set history maintains the derivation history of individual revisions along with other type-specific information including object attributes. An example version set history for an HB3 object is illustrated in Figure 8. As shown in the figure, the version set history maintains two classes of attributes for objects: (1) version-set-history-level attributes that contain information that applies to all revisions of an object and (2)

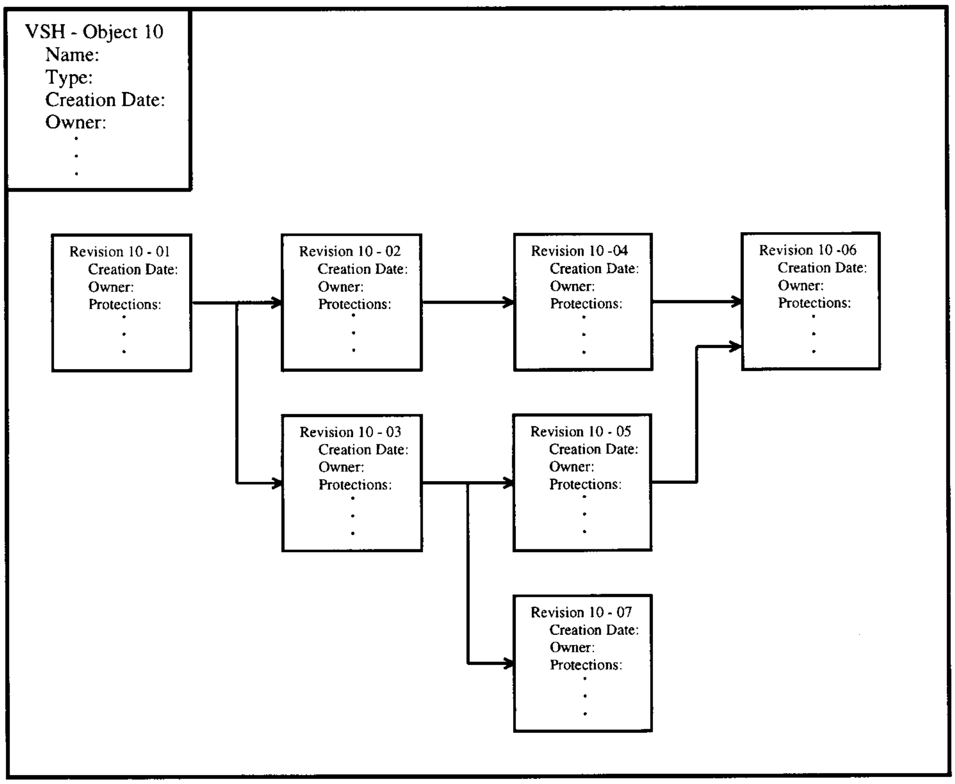

VSH - Object 10
Name:
Type:
Creation Date:
Owner:
Revision 10 - 01 Creation Date: Owner: Protections:
Revision 10 - 02 Creation Date: Owner: Protections:
Revision 10 - 03 Creation Date: Owner: Protections:
Revision 10 -04 Creation Date: Owner: Protections:
Revision 10 - 05 Creation Date: Owner: Protections:
Revision 10 -06 Creation Date: Owner: Protections:
Revision 10 - 07 Creation Date: Owner: Protections:

Fig. 8. Version set history for an HB3 object.

revision-level attributes that contain information that varies between individual revisions of an object. The revision labeled "10-06" in Figure 8 has two incoming arcs, indicating the ability of the DAG data model to capture the result of merge operations [Campbell and Goodman 1988].

Attributes for HB3 objects can also be classified as being system defined or type specific. Every object has a predetermined set of system-defined attributes at both the version set history and individual revision levels. These attributes contain standard information used by the VOM for object management purposes. Objects can also have type-specific attributes used to capture application-specific information. These attributes enable the version server to readily accommodate new object types. Figure 9 illustrates system defined attributes for HB3 objects and the relationship among the different classes of object attributes.

In HB3, an object's version set history completely records all information necessary for managing the individual revisions of the object. When the VOM encounters an object reference, it always consults the appropriate version set history which provides the information needed to locate the required revision. The controlled access to object revisions allows integrity

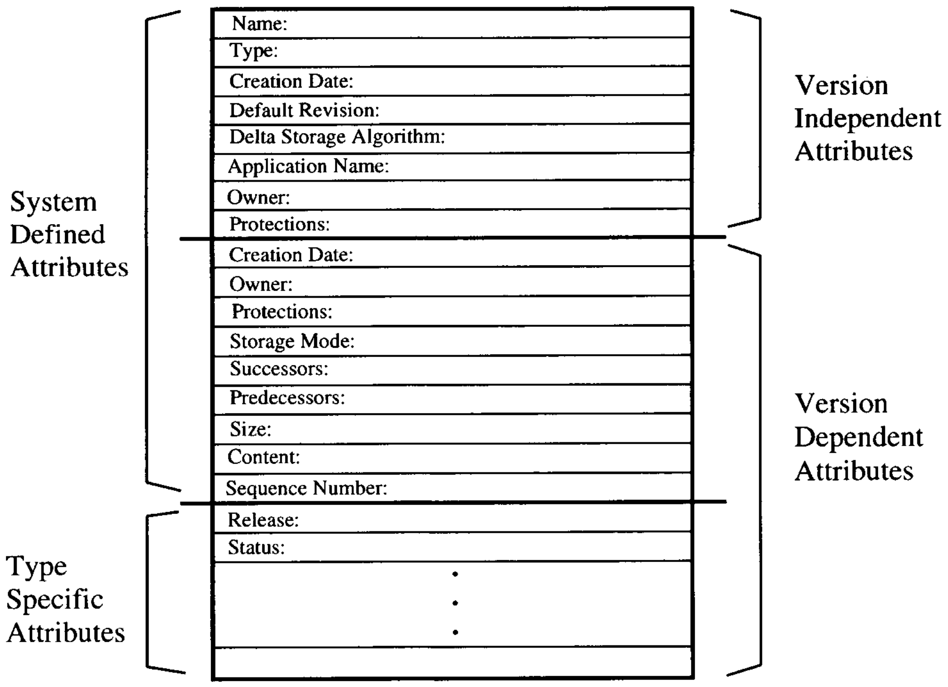


Fig. 9. HB3 system-defined attributes and relationship among object attribute classes.

constraints to be imposed on object manipulation functions. The composite object type is used in the implementation of version set histories.

4.2.2 *Object Identification.* The VOM completely supports the object identification strategy specified in the version control framework. Objects can be identified through the use of attributes or unique identifiers. To completely identify a specific revision, two levels of specification are required: the version set history level and the individual revision level (Figure 8). Attributes are useful for specification at both levels. For example, a combination of the "name," "type," and "creation date" attributes could be used to identify the version set history of a particular object.

The object specification process within version set histories is similarly flexible. For example, attribute values can be specified that statically identify a specific revision within a version set history such as "creation date" and "owner," or alternatively, dynamic object specification can be accomplished through attribute values such as "date = latest." If no object specification is provided for an object reference at the individual revision level, the "default revision" attribute of the version set history guides the selection process.

Any combination of attributes can be used for object specification at the version set history or revision levels. An attribute-based object reference is specified with three vectors of information indicating the relevant object attributes, values for each attribute, and a relational operator to be applied to each attribute/value pair. Five relational operators are currently supported: equality, greater than, less than, greatest, and least. This strategy provides a wide range of object specification capabilities significantly enhancing the utility of associations, composite objects, and object management operations.

The unique identifier assigned by the VOM to each object when it is created can also be used for identification. Unique identifiers provide a convenient mechanism for static object references. Additionally, the strategy used by the VOM for the assignment of unique identifiers enables them to be used in conjunction with attributes for object identification. For example, a unique identifier can be used to specify a version set history for an object, and attribute values can then be used to select a specific revision.

4.2.3 *Object Management Operations*. The services offered by the VOM are made accessible through a set of modular object management operations. These operations support routine object management activities and allow version control operations to be performed. VOM client applications are in complete control over the frequency and granularity with which these operations are performed. It is through the selective use of these operations that VOM clients implement application-specific version control policies.

The basic object management operations of the VOM are create, delete, update, and retrieve. Each of these operations is defined for both version set histories and individual revisions. These primitive operations form the basis of all object management activities in the VOM. Table I summarizes both levels of object management operations along with their parameters and the server responses. Example uses of VOM object management operations include a revision-level create command used to create a new revision of an object and a revision-level update command that could be used to write over an existing revision; alternatively, a version-set-history-level create command could be used to create a completely new object.

The VOM supports the creation of convenience functions, which are intended to provide a slightly higher-level interface to the VOM for frequently performed operations. For example, initial experiences indicated that the majority of revision-level update and retrieve operations primarily involve the content attribute. Rather than require applications to continually specify an attribute vector for these operations indicating that only the content attribute is to be involved, "content_retrieve" and "content_update" convenience functions have been defined. Convenience functions prevent the flexibility provided by the low-level modularity of the standard object management operations from unnecessarily complicating interaction with the VOM.

Table I. HB3 VOM Object Management Operations and Their Parameters

	Operation	Parameters to Server	Server Response
Version Set History Level	create	New version set history attribute vector	Initial revision object identifier
		Initial revision attribute vector	RC (return code)
	update	Version set history identifier Update attribute vector Attribute value vector	RC
	retrieve	Version set history identifier Retrieve attribute vector	Attribute value vector RC
	delete	Version set history identifier	RC
Revision Level	create	Version set history identifier	New revision object identifier
		Parent revision identifier Initial attribute values	RC
	update	Revision object identifier Update attribute vector Attribute value vector	RC
	retrieve	Revision object identifier	Attribute value vector(s) (this returns a value vector for all matching revisions)
		Retrieve attribute vector	RC
	delete	Revision object identifier	RC

4.3 HB3 Association Set Manager

In HB3, the association set manager (ASM) is responsible for handling the structural information required to represent hypermedia connections. As indicated earlier it provides the structural abstractions necessary to support the HURL hypermedia model. The ASM uses the VOM for object management services. The VOM supplies a platform that is well suited for the implementation of ASM structural abstractions. The composite object type is used extensively to represent the collections of identifiers that correspond to structural information in the HURL model. Additionally, the versioning capabilities of the VOM are used directly by the ASM to provide version support for structural data.

Using VOM functionality, the ASM supports complete version histories for the association, side, and bridge structural abstractions. Client applications can utilize the services of the ASM, through the link services manager (LSM), to maintain version histories for these structural entities. The availability of version support for these structural entities allows structural evolution to be completely captured at the individual hypermedia connection level.

The ASM provides an additional construct to version structure at higher levels. A *context* is a grouping mechanism for structural data. The ASM, using VOM functionality, supports version histories for contexts. This enables changes made to the structure as a whole to be captured. Alternatively, application developers can use the composite object type to define their own specialized constructs for capturing structural evolution at higher levels.

The versioning capabilities provided by the ASM, like those of the VOM, are policy neutral. Associations, sides, bridges, and contexts are not automatically versioned. Client applications, through the selection of appropriate LSM/ASM operations, control the versioning of structural entities. This enhances the tailorability and flexibility of the ASM and allows structure to be versioned at any frequency and at any level of granularity required within HB3 client applications.

The complete object identification capabilities of the VOM are available to and supported by the ASM. Object references contained within the association, side, bridge, and context structural abstractions can be defined dynamically as well as statically. This enables the ASM to fully support structural dynamic linking, significantly increasing the utility of its structure management services.

4.4 Version Control in the HB3/SP3 Environment

The V text editor was the first application-level HB3 client to use the version control facilities of the VOM. The V text editor application is a derivative of the "vi" editor from the Unix environment. V has been used to extensively test the object management and version control operations of the VOM and the ASM. Currently, V provides a linear version history for the text objects it manages. It uses a derivative of the Unix "diff" command as a delta calculation algorithm. V uses the policy neutrality of the VOM and ASM to support a user-controlled version creation policy. Whenever a change is made to a data or structural item, the user has the option to update an existing revision of the object or create a new one. A number of additional application-level client applications are planned that will use the version control services of the SP3 environment in a variety of ways.

As described earlier, the version control functionality of the VOM is not used exclusively by end-user client applications. As illustrated in Figure 10, the services of the VOM can be utilized by manager applications as well. The use of the VOM by internal managers has had a significant effect on the HB3 system. It has enabled these managers to augment their services in important ways, such as the addition of version control capabilities to the structural management services provided by the ASM.

The organization of components in the HB3/SP3 environment has resulted in a *metadata* architectural layer. The metadata layer, as illustrated in Figure 10, is an area of functionality in which managers of various kinds

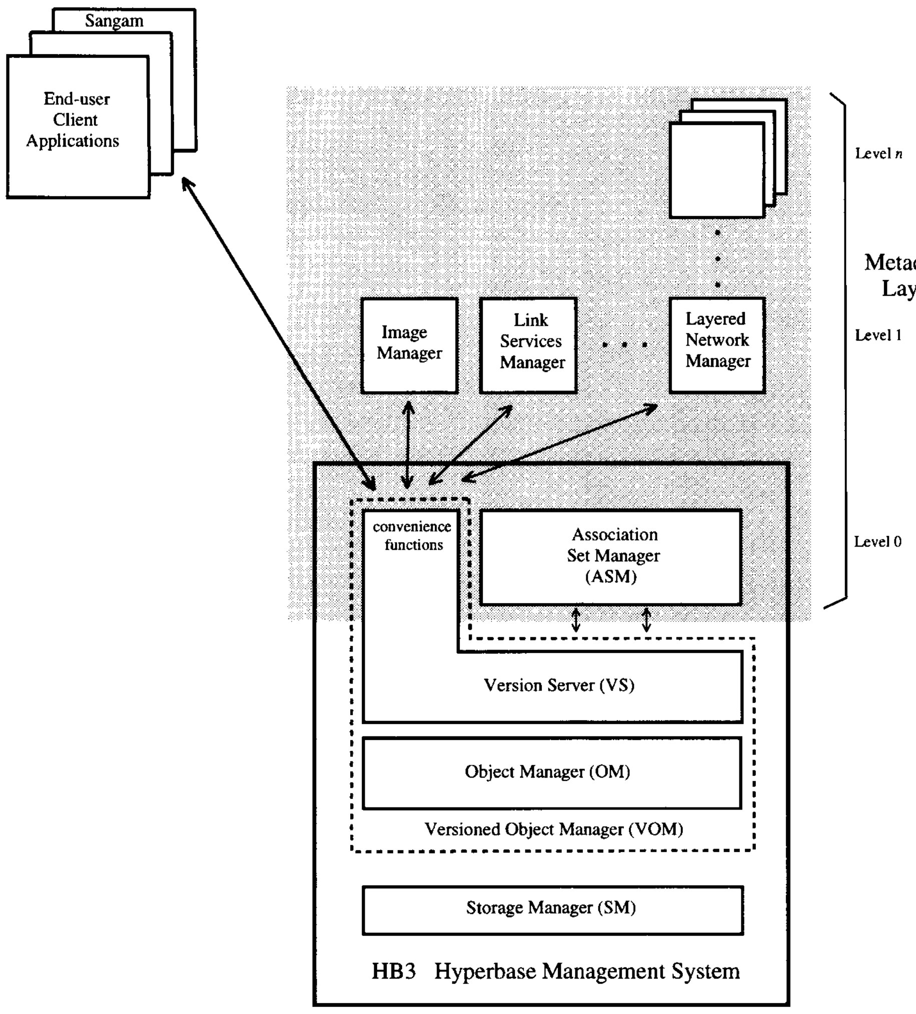


Fig. 10. Use of the versioned object manager in the SP3 environment.

build upon the functionality of the VOM to implement useful abstractions for SP3 client applications. Figure 10 identifies two levels within the metadata layer. Level 0 is contained within HB3, and it includes those managers such as ASM that supply functionality usually associated with a HBMS. Level 1 metadata managers build on HBMS functionality to implement higher-level abstractions. Metadata managers can build upon the abstractions of other metadata managers. The number of metadata levels possible in the architecture is open ended.

The implementation of manager applications in the metadata layer, especially those providing version-control-related abstractions, is facilitated by the availability of the HBMS-level VOM. It is anticipated that the metadata layer will be very useful in the examination of high-level development methodologies and abstract version control data models. Implementation of metadata managers that provide this type of functionality will enable high-level version control to be easily incorporated into client applications. This will produce an environment in which developers can experiment with a range of development methodologies and version control data models to determine which can best support the development process in their application area.

The first level 1 metadata manager in the SP3 environment, an image manager, has been completed [Shah and Leggett 1993]. The image manager uses HB3 services to create abstractions based on the cel animation process and makes extensive use of the HB3 composite object type to represent the cell, frame, and sequence abstractions it offers. The image manager also uses the versioning capabilities of the VOM to support complete version histories for each of these object types. The Sangam group drawing and annotation tool is an end-user SP3 client application. It uses the facilities of the image manager for the storage and general management of the cell, frame, and sequence abstractions it provides at the user interface level [Shah and Leggett 1993].

The collection of task-specific object management operations into a distinct metadata architectural layer, separate from the underlying intrinsic object management operations of the HBMS, is a generalization of the defining concept of the version control framework. A number of useful metadata managers providing higher-level version control functionality are planned, including a graphical version history inspection tool, a layered network data model manager, and a task-oriented development manager.

5. IMPLICATIONS OF THE HYPERMEDIA VERSION CONTROL FRAMEWORK

A direct consequence of the original design goals for the framework has been to focus attention on the partitioning of hypermedia version control functionality. The only version control functionality explicitly specified within the framework is the set of intrinsic operations supplied by the HBMS-level version server. This facility alone, however, represents a very capable version control infrastructure offering as much functionality as many existing version control systems. The version control capabilities of many existing software configuration management systems could easily be implemented within the framework through the direct use of version server functionality.

Although much of the emphasis in the framework is on intrinsic operations, it does not neglect higher-level version control functionality. As illustrated in Figure 10, this functionality is simply elevated to a more appropriate level of the architecture. Application developers are provided with a comprehensive version control platform at the HBMS level upon which to construct higher-level version control services. For example, consider the facilities of the CoVer context-oriented version server, which was designed to support the hypermedia publishing process [Haake 1992;

1994]. The data model of the CoVer system is an abstract one, specifically designed to support hypermedia publishing. It provides two types of objects: single state and multistate. It is specialized to track the reuse of materials across document boundaries, independent of physical-storage-level considerations. Additionally, CoVer provides a task mechanism which supports structured work activities and controls when version operations such as version creation are performed.

The abstract version control facilities of the CoVer system could be implemented within the framework of building directly on the functionality of the versioned object manager (VOM). Both the single-state and multistate object types of the CoVer system could be implemented as standard VOM objects. Use of the appropriate object management operations would allow single-state objects to be updated in place (revision-level update command) and new versions of multistate objects to be created as necessary (revision-level create command). Attribute fields could be defined to record and track derivation information (e.g., a "derived_from" attribute). The policy-neutral nature of the framework enables the developer to completely control when and at what granularity version-related operations are performed, providing the flexibility necessary to use the VOM platform of intrinsic functionality to define and implement the task mechanism.

Abstract version control data models, such as the one described above, can be implemented either within an application, or as a manager process residing in the metadata layer of the framework (Figure 10). Implementation within the metadata layer allows a versioning abstraction to be utilized by other applications and manager processes. As the metadata architectural layer becomes populated with metadata managers, developers will have the ability to easily experiment with a number of different higher-level data models and development methodologies to determine which works best for their application area.

The implications of the version control framework are not limited to the hyperbase management system area. The framework brings a unifying and clarifying influence to hypermedia version control issues in general. A range of proposals has been suggested to address version control in hypermedia [Durand et al. 1995; Haake 1992; Hicks et al. 1996; Østerbye 1992; Prevelakis 1990]. The research efforts represented by these proposals have not produced a consensus on the appropriate strategies to cope with hypermedia version control issues. Experiences during the development of this framework suggest that much of the variance in proposed solutions can be directly attributed to differences in the context in which version control is being investigated.

The primary areas in which existing research frameworks differ are the intended application area and the hypermedia model that have influenced their design. These two factors significantly affect the design of hypermedia version control services. Differences in the intended application area are reflected in the policies with which user interaction issues are addressed. For example, the frequency or process of version creation might differ significantly between an authoring system and a software configuration management system. Differences in the hypermedia model primarily affect the structural versioning capabilities of a system. For example, it might not be possible to version hypermedia associations individually in a system that does not abstract hypermedia structure from data. The version control framework described here was designed to avoid limiting influences from either of these categories.

Application independence is a defining characteristic of the HBMS-level version server. It motivated the functional partitioning of hypermedia version control operations into the application-independent (HBMS level) and application-dependent (application level) categories. The organization of version control functionality produced by the partitioning process can explain and reconcile many of the differences and disagreements found in existing proposals for version control in hypermedia. It illuminates the application-specific decisions that are responsible for many of the differences.

Additionally, the version control framework is designed to support an open model of hypermedia. The generality of the model is reflected in the extensive capabilities provided within the framework for versioning structure. The structural versioning environment produced by the framework represents a metric that can be useful in identifying limitations of hypermedia version control facilities caused by restrictions in the underlying hypermedia model.

These two important characteristics of the version control framework produce an enhanced context in which to investigate version control in hypermedia, one that encourages discussion of relevant issues by exposing disagreements and dissimilarities that are inherently caused by differences in target application areas and underlying hypermedia models.

6. RELATED WORK

The majority of research related to version control is found in the software configuration management area. Several systems have been developed to assist with the software development process [Dart 1991]. A number of the version control issues encountered during the design of these systems are relevant to the hypermedia area, including the version control data model [Cohen et al. 1988; Glasser 1978; Goldstein and Bobrow 1984; Leblang and Chase 1987; Rochkind 1975; Tichy 1985], storage management [Leblang and Chase 1987; Lelewer and Hirschberg 1987; Reichenberger 1991; Scacchi 1988; Tichy 1988], configuration management [Estublier 1988; Feldman 1988; Leblang and McClean 1985; Mahler and Lampen 1988; Miller et al. 1989; Tichy 1988; van der Hoek et al. 1996], object identification [Demleitner 1988; Estublier 1988; Leblang et al. 1988; Mahler and Lampen 1988], and user interaction [Belkhatir and Estublier 1986; Cohen et al. 1988; Tichy 1985; 1988].

Significant research on version control has also been conducted in the computer-aided design area. Several CAD systems have been constructed to facilitate the design and development of complex engineering artifacts

[Katz 1990]. Many issues encountered in the design and development of CAD systems are similarly relevant to version control in hypermedia including the data model [Ahmed and Navathe 1991; Chou and Kim 1986; Katz et al. 1986], storage management [Ecklund et al. 1987; Katz and Lehman 1984; Katz et al. 1986], structure management [Ahmed and Navathe 1991; Batory and Kim 1985; Dittrich and Lorie 1988; Katz et al. 1986], object identification [Ecklund et al. 1987; Katz and Chang 1987], user interaction [Ecklund et al. 1987; Katz and Chang 1987; Katz et al. 1986], and version proliferation [Ahmed and Navathe 1991; Rumbaugh 1988].

Much of the related research in the hypermedia area has focused on version control within individual hypermedia systems [Akscyn et al. 1988; Garrett et al. 1986; Kydd et al. 1995; Prevelakis 1990; Yankelovich et al. 1988]. Versioning has been identified as an important requirement for future systems based on the World Wide Web (WEBDAV).[2] Version control has also been examined in conjunction with HBMS research. The Hypertext Abstract Machine (HAM) provides a linear version thread for the objects it manages [Campbell and Goodman 1988]. The Neptune hypertext system, utilizing the capabilities of the HAM, provides a linear version thread for hypertext objects [Delisle and Schwartz 1987]. The CoVer contextual version server has been designed to provide task-oriented version control services targeted for the hypermedia publishing environment [Haake 1992; 1994]. It has been implemented in the HyperBase HBMS environment and is used by the SEPIA cooperative authoring system [Haake 1994; Schütt and Haake 1993]. The Fenris HBMS has been extended to incorporate version control services to support the Hyperstructure Programming Environment [Østerbye 1992].

The primary distinction between the hypermedia version control framework and related approaches is in the intended application area. Almost all research efforts that focus on version control in hypermedia have been designed to support the version control needs of a specific hypermedia system. Others have been conducted in the HBMS environment, but focus on support for a particular application area (e.g., hypermedia publishing or software development). Unlike existing approaches, the hypermedia version control framework has no specific target application area. The partitioning of hypermedia version control functionality that was an integral part of its design was meant to produce a completely general-purpose hypermedia version control framework capable of supporting all HBMS client applications with the maximum amount of hypermedia version control functionality possible.

7. FUTURE WORK

The next major focus of research will be on version control within client applications, with an emphasis on higher-level version control data models

[2]Further information available from http://www.ics.uci.edu/~ejw/authoring.

and development methodologies. A number of metadata managers are planned that will offer higher-level version control abstractions for client applications. In particular, a layered network data model manager and a task-oriented development manager are planned. The design and implementation of these managers, along with others, will provide further data with which to evaluate the version control framework and prototypic implementation. These managers will also facilitate the examination of the usefulness of these version control abstractions in different application areas.

Another important area of future work is the ongoing development and refinement of the version control framework and its prototypic implementation. Current plans include an expansion of the access control facilities along with the examination of additional issues related to the support of collaboration within the context of the framework such as concurrency control, merge support, and notification control [Haake et al. 1995; 1996; Munson and Dewan 1994; Wiil and Leggett 1993].

8. SUMMARY

The areas of application of hypermedia technology, combined with the capabilities that hypermedia provides for manipulating structure, create an environment in which version control is very important. A hypermedia version control framework has been developed to specifically address the version control problem in open hypermedia environments. It is based on a functional partitioning of the hypermedia version control process into the categories of intrinsic and application-specific operations. The framework defines a comprehensive version control environment, providing application-independent version control functionality in the form of a hypermedia version server. Application-dependent version control functionality is implemented within client applications utilizing the intrinsic operations of the version server.

The version control framework has been integrated into a prototype HBMS. The HB3 HBMS provides complete version histories for the objects it manages, both data and structural. The organization of components within the HB3 system has resulted in a metadata architectural layer that has proved very useful in the investigation of higher-level version control data models and development methodologies. The analysis of the version control process in hypermedia performed during the development of the framework brings a clarifying influence to hypermedia version control issues. The organization of hypermedia version control functionality within the framework helps to identify and expose the application-specific version control decisions that represent most of the differences found in existing approaches to version control in hypermedia.

ACKNOWLEDGMENTS

The authors would like to thank Knowledge Systems for making the KMS hypermedia system available for our distributed, collaborative writing effort.

REFERENCES

AHMED, R. AND NAVATHE, S. B. 1988. Version management of composite objects in CAD databases. *Commun. ACM 31*, 7 (July), 218–227.

BELKHATIR, N. AND ESTUBLIER, J. 1986. Protection and cooperation in a software engineering environment. In *Proceedings of the International Workshop on Advanced Programming Environments* (Trondheim, Norway, June 16-18), R. Conradi, T. M. Didriksen, and D. H. Wanvik, Eds. Springer-Verlag, Berlin, Germany, 221–229.

BERNERS-LEE, T., CAILLAU, R., GROFF, J., AND POLLERMAN, B. 1992. World-Wide Web: The information universe. *Elec. Netw. Res. App. Pol. 1*, 2.

CAMPBELL, B. AND GOODMAN, J. M. 1988. HAM: A general purpose hypertext abstract machine. *Commun. ACM 31*, 7 (July), 856–861.

COHEN, E. S., SONI, D. A., GLUECKER, R., HASLING, W. M., SCHWANKE, R. W., AND WAGNER, M. E. 1988. Version management in Gypsy. *SIGPLAN Not. 24*, 2 (Feb.), 201–215.

DART, S. 1991. Concepts in configuration management systems. In *Proceedings of the 3rd International Conference on Software Configuration Management* (Trondheim, Norway, June 12–14), P. H. Feiler, Ed. ACM Press, New York, NY, 1–18.

DELISLE, N. M. AND SCHWARTZ, M. D. 1987. Contexts—A partitioning concept for hypertext. *ACM Trans. Inf. Syst. 5*, 2 (Apr.), 168–186. Formerly *ACM Trans. Off. Inf. Syst.*

DURAND, D., HAAKE, A., HICKS, D., AND VITALI, F., EDS. 1995. *Proceedings of the Workshop on Versioning in Hypertext Systems*.

ECKLUND, D. J., ECKLUND, E. F., EIFRIG, R. O., AND TONGE, F. M. 1987. DVSS: A distributed version storage server for CAD applications. In *Proceedings of the 13th Conference on Very Large Data Bases* (Brighton, England, Sept.). VLDB Endowment, Berkeley, CA, 443–454.

ESTUBLIER, J. 1988. Configuration management: The notion and the tools. In *Proceedings of the International Workshop on Software Version and Configuration Control* (Grassau, Germany, Jan.). Prentice-Hall, Inc., Upper Saddle River, NJ, 38–61.

FELDMAN, S. 1988. Evolution of Make. In *Proceedings of the International Workshop on Software Version and Configuration Control* (Grassau, Germany, Jan.), 413–416.

GOLDSTEIN, I. P. AND BOBROW, D. G. 1984. A layered approach to software design. In *Interactive Programming Environments*, D. R. Barstow, H. E. Shrobe, and E. Sandewall, Eds. McGraw-Hill, Inc., New York, NY, 387–413.

HAAKE, A. 1992. CoVer: A contextual version server for hypertext applications. In *Proceedings of the European Conference on Hypertext (ECHT '92)* (Milan, Italy, Nov. 30–Dec. 4), D. Lucarella, J. Nanard, M. Nanard, and P. Paolini, Eds. ACM Press, New York, NY, 43–52.

HAAKE, A. 1994. Under CoVer: The implementation of a contextual version server for hypertext applications. In *Proceedings of the European Conference on Hypermedia Technology (ECHT '94)* (Edinburgh, Scotland, UK, Sept. 18–23). ACM Press, New York, NY, 81–93.

HAAKE, A. AND HICKS, D. 1996. VerSE: Towards hypertext versioning styles. In *Proceedings of the 7th ACM Conference on Hypertext (Hypertext '96)* (Washington, D.C., Mar.). ACM Press, New York, NY, 224–234.

HAAKE, A., HAAKE, J., AND HICKS, D. 1995. On merging hypertext networks. In *Proceedings of the Workshop on the Role of Version Control in CSCW*, D. Hicks, A. Haake, D. Durand, and F. Vitali, Eds.

HALASZ, F. G. 1986. Reflections on NoteCards: Seven issues for the next generation of hypermedia systems. *SIGMOD Rec. 15*, 2 (June), 836–852.

HICKS, D. L. 1993. A version control architecture for advanced hypermedia environments. Dissertation. Texas A&M University, College Station, TX.

HICKS, D., HAAKE, A., DURAND, D., AND VITALI, F., EDS. 1995. *Proceedings of the Workshop on the Role of Version Control in CSCW*.

KATZ, R. H. 1974. Toward a unified framework for version modeling in engineering databases. *IEEE Trans. Softw. Eng. SE-5*, 3 (May), 375–409.

KATZ, R. H. AND CHANG, E. 1987. Managing change in a computer-aided design database. In *Proceedings of the 13th Conference on Very Large Data Bases* (Brighton, England, Sept.). VLDB Endowment, Berkeley, CA, 455–462.

KATZ, R. H. AND LEHMAN, T. J. 1984. Database support for version and alternatives of large design files. *IEEE Trans. Softw. Eng. SE-10*, 2 (Mar.), 191–200.

KYDD, S., DYKE, A., AND JENKINS, D. 1995. Hypermedia version support for the online design journal. In *Proceedings of the Workshop on Versioning in Hypertext Systems*, D. Durand, A. Haake, D. Hicks, and F. Vitali, Eds.

LEBLANG, D. B. AND CHASE, R. P. 1987. Parallel software configuration management in a network environment. *IEEE Softw. 4*, 6 (Nov.), 28–35.

LEBLANG, D. B. AND MCCLEAN, G. D. 1985. Configuration management for large-scale software development efforts. In *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large*, 122–127.

LEBLANG, D., CHASE, R., AND SPILKE, H. 1988. Increasing productivity with a parallel configuration manager. In *Proceedings of the International Workshop on Software Version and Configuration Control* (Grassau, Germany, Jan.), 21–37.

LEGGETT, J. J. 1993. Report of the Workshop on Hyperbase Systems held in conjunction with Hypertext'93. Tech. Rep. TAMU-HRL 93-009, Texas A&M University, College Station, TX.

LEGGETT, J. J. AND SCHNASE, J. L. 1994. Viewing Dexter with open eyes. *Commun. ACM 37*, 2 (Feb.), 76–86.

LEGGETT, J. J., SCHNASE, J. L., SMITH, J. B., AND FOX, E. A. 1993. Final report of the NSF workshop on hyperbase systems. Tech. Rep. TAMU-HRL 93-002, Texas A&M University, College Station, TX.

LELEWER, D. A. AND HIRSCHBERG, D. S. 1987. Data compression. *ACM Comput. Surv. 19*, 3 (Sept.), 261–296.

MAHLER, A. AND LAMPEN, A. 1988. Shape—A software configuration management tool. In *Proceedings of the International Workshop on Software Version and Configuration Control* (Grassau, Germany, Jan.), 228–243.

MILLER, D. B., STOCKTON, R. G., AND KRUEGER, C. W. 1989. An inverted approach to configuration management. In *Proceedings of the 2nd International Workshop on Software Configuration Management* (Princeton, NJ, Oct. 24, 1989), R. N. Taylor, Ed. ACM Press, New York, NY, 1–4.

MUNSON, J. P. AND DEWAN, P. 1994. A flexible object merging framework. In *Proceedings of the Conference on Computer Supported Cooperative Work* (Chapel Hill, NC, Oct. 22–26). ACM Press, New York, NY, 231–242.

ØSTERBYE, K. 1992. Structural and cognitive problems in providing version control for hypertext. In *Proceedings of the European Conference on Hypertext (ECHT '92)* (Milan, Italy, Nov. 30–Dec. 4), D. Lucarella, J. Nanard, M. Nanard, and P. Paolini, Eds. ACM Press, New York, NY, 33–42.

ØSTERBYE, K. AND NØRMARK, K. 1994. An interaction engine for rich hypertexts. In *Proceedings of the European Conference on Hypermedia Technology (ECHT '94)* (Edinburgh, Scotland, UK, Sept. 18–23). ACM Press, New York, NY, 167–176.

PREVALAKIS, V. 1990. Versioning issues for hypertext systems. In *Object Management*, D. Tsichritzis, Ed. Universitaire d'Informatique, Université de Genéve, Geneva, Switzerland, 89–105.

REICHENBERGER, C. 1991. Delta storage for arbitrary non-text files. In *Proceedings of the 3rd International Conference on Software Configuration Management* (Trondheim, Norway, June 12–14), P. H. Feiler, Ed. ACM Press, New York, NY, 144–152.

ROCHKIND, M. J. 1975. The source code control system. *IEEE Trans. Softw. Eng. SE-1*, 4 (Dec.), 364–370.

RUMBAUGH, J. 1988. Controlling propagation of operations using attributes on relations. *SIGPLAN Not. 23*, 11 (Nov.), 285–296.

SCACCHI, W. 1988. Summary of plenary discussion "CM for non-textual representation". In *Proceedings of the International Workshop on Software Version and Configuration Control* (Grassau, Germany, Jan.), 363–368.

SCHNASE, J. L. 1992. HB2: A hyperbase management system for open, distributed hypermedia system architectures. Dissertation, Texas A&M University, College Station, TX.

SCHNASE, J. L., LEGGETT, J. J., AND HICKS, D. L. 1993. HB1: Initial design and implementation of a hyperbase management system. *Electron. Publ. Origin. Dissem. Des. 6*, 2 (July).

SCHNASE, J., LEGGETT, J., HICKS, D., NUERNBERG, P., AND SANCHEZ, A. 1994. Open architectures for integrated, hypermedia-based information systems. In *Proceedings of the 27th Annual Hawaiian International Conference on System Science (HICSS '94)* (Maui, Hawaii, Jan.). IEEE Computer Society Press, Los Alamitos, CA, 386–395.

SCHÜTT, H. AND HAAKE, J. 1993. Server support for cooperative hypermedia systems. In *Proceedings of Hypermedia '93 Conference* (Zurich, Switzerland, Mar.).

SHAH, A. AND LEGGETT, J. 1993. Image manager. Tech. Rep. TAMU-HRL 93-003, Texas A&M University, College Station, TX.

STONEBRAKER, M. AND KEMNITZ, G. 1991. The POSTGRES next-generation database management system. *Commun. ACM 34*, 10 (Oct.), 78–92.

THOMAS, I. 1989. Version and configuration management on a software engineering database. *SIGSOFT Softw. Eng. Notes 17*, 7 (Nov.), 23–25.

TICHY, W. F. 1985. RCS: A system for version control. *Softw. Pract. Exper. 15*, 7 (July), 637–654.

TICHY, W. F. 1988. Tools for software configuration management. In *Proceedings of the International Workshop on Software Version and Configuration Control* (Grassau, Germany, Jan.), 1–20.

VAN DER HOEK, A., HEIMBIGNER, D., AND WOLF, A. 1996. A generic, peer-to-peer repository for distributed configuration management. In *Proceedings of the 18th International Conference on Software Engineering* (Berlin, Germany, Mar. 25-29). IEEE Computer Society Press, Los Alamitos, CA.

WIIL, U. K. 1993. Extensibility in open, distributed hypertext systems. Ph.D. dissertation, Aalborg University, Aalborg, Denmark.

WIIL, U. K. AND LEGGETT, J. J. 1993. Concurrency control in collaborative hypertext systems. In *Proceedings of the 5th ACM Conference on Hypertext (Hypertext '93)* (Seattle, WA, Nov. 14–18). ACM Press, New York, NY, 14–24.

YANKELOVICH, N., HAAN, B., MEYROWITZ, N., AND DRUCKER, S. 1988. Intermedia: The concept and the construction of a seamless information environment. *Computer 21*, 1 (Jan.), 81–96.

Received: April 1995; revised: May 1997; accepted: May 1997