Under CoVer: The Implementation of a Contextual Version Server for Hypertext Applications

Anja Haake
IPSI - Integrated Publication and Information Systems Institute
GMD - German National Research Organization for Information Technology
Dolivostr. 15, D-64293 Darmstadt, Germany
ahaake@darmstadt.gmd.de

ABSTRACT

At GMD-IPSI we are developing CoVer, a *contextual* version server for hypertext applications [11]. Another characterization of CoVer is that CoVer integrates state-oriented versioning concepts with task-oriented versioning concepts. While other version models in general support only one of these groups of concepts, we argue that the explicit composition of versions of complex hypertext networks has to be complemented by automatic version creation in the context of tasks or jobs performed while manipulating the hypertext network and vice versa. Regarding the implementation of version models, it turns out that the state-oriented implementation approach - representing every legal state of a hyperdocument explicitly - and the task-oriented implementation approach - computing versions of complex hypertext networks due to changes executed during a task or job - are interchangeable. While the separation of state- and task-oriented concepts at the conceptual level of the version model is desireable to support version creation and selection for different hypertext applications, the implementation of such a dual model can be based on a single implementation approach. This paper discusses both types of implementation with an emphasis to cope with alternative versions that are in particular meaningful for hypertext publishing applications.

KEYWORDS

Versioning, alternatives, state-oriented versioning, task-oriented versioning, implementation techniques, publishing applications

1 INTRODUCTION

This is the third in a series of papers introducing CoVer, a Contextual Version Server under development at GMD-IPSI in order to cope with problems of version support in hypertext-based publishing environments. While [11] introduces the CoVer version model and [9] discusses the integration of version support into the SEPIA cooperative hypermedia authoring system, this paper focuses on the implementation of CoVer.

After introducing the state-of-the-art of version models (Section 2), an overview of the CoVer version model is given (Section 3). We discuss the options in implementing a version server for hypertext applications (Section 4). Finally, we discuss the key issues in implementing a version server for hypertext applications in the context of related work and hypertext publishing applications (Section 5), and introduce future work (Section 6).

2 RELATED WORK

Approaches to versioning can be categorized into state-oriented and task-oriented version models. The former is also called version-oriented [18] or bottom-up [15], the latter change-oriented [18] or top-down [15] versioning.

State-oriented version models keep the versions of each individual object in a so-called version set. Versions of complex objects, i.e., objects composed of other objects, have to be defined explicitly by selecting a version of each component object.

The most problematic feature in state-oriented approaches is that coordinated changes spreading over many system components are not reflected appropriately. For example, a bug fix may address several files or modules. All affected component versions have to be equipped with user-supplied attributes to mark that these versions represent a coordinated change. The composition of versions of complex objects thus is error prone.

While state-oriented versioning focuses on individual versions of individual objects, task-oriented versioning focuses on versions of complex systems as a whole. The key idea of task-oriented approaches is to provide system support for maintaining the relationships between those versions that have been changed in a coordinated manner and to assure that the resulting versions can be identified by the set of coordinated changes. In this way, unintended combinations of component versions can be ruled out a priori and versions can be selected via change information.

The first task-oriented version model was proposed by PIE [8], the Personal Information Environment, a software development environment for Smalltalk. In PIE *layers* group changes of several components of a system into an identifiable unit. Once a layer has been created on a software system, all changes performed in this layer are recorded automatically to derive a new version of the software system. Layers can be laid one upon another to combine several changes to a version of the system. Fixed combinations of layers can be stored as *contexts*.

Whereas task-oriented versioning is mainly pursued in the software engineering community, the idea of task-oriented versioning has started to also influence the database community (cf. [3] or [14] for a review on versioning in databases). An approach towards automatic version creation due to coordinated changes is the task-oriented version approach intended by O2 [4], [5]. The task-oriented versioning concept of O2 is based on the notion of a *database version*. A database version contains exactly one version for each versioned object and all versions together form a consistent state of the real world modeled in the database. The database versions may be organized in a hierarchy. Each child database version sees all versions of its parent database version. Those versions are called *shared object versions* between database versions. If an attempt is made to update a shared object version, the update is performed automatically in a new version (*unshared object version*) that only belongs to the database version that caused the update of the shared version.

The primary purpose of database versions is to keep consistency of versions in object-oriented databases. So, O2 has no notion of immutability of database versions. Parent database versions can be manipulated independently from the presence of open child database versions. Thus, versions can not be identified by a fixed set of changes.

The Raleigh activity model introduced in [15] is a task-oriented version model for databases. Raleigh is an object-oriented database system that uses a functional object model. The purpose of *activities* is to maintain an "episode of work" and to attach version, concurrency and access control to this episode of work. Activities can be structured in a parent / child activity hierarchy and siblings can be qualified as predecessor / successor activities. Activities can be closed to record the resultant version of an activity. A parent activity is suspended as long as one of its children is still open.

Raleigh has no notion of state-oriented versioning. Due to the lack of a notion of state, Raleigh does not even have the notion of a version: If a property of an included version is updated in an activity, the new value of this property will be kept in the activity and will replace the previous value.

As for the domain of hypertext systems, Prevelakis [26] applies the PIE model directly to a hypertext system with atomic nodes and binary links. The model of Østerbye [25] provides versions for atomic and composite[1] nodes. A step towards task-oriented version support is that each composite node provides a selection criteria that allows so-called generic version links to select versions of versioned nodes dynamically, in particular on time-based information.

Hicks [12, 13] developed a general architecture to integrate version support into hypertext systems that distinguishes versioning at the application level and at the hyperbase level, i.e., an application-specific version server should be built on top of a general version server. The model served as a framework for the design of version control services for the HB3 hyperbase management system [17, 30], which offers state-oriented version management at the hyperbase level for simple and composite objects [17]. Wiil and Leggett [30] discuss the state-oriented HB3 model as an extension to concurrency control to provide conflict-free write operations. They emphasize that the model allows branching of versions for parallel activities and propose to make the time interval between version merges a parameter to the version control mechanism. Also the hypertext platform Hyperform [29] offers state-oriented version support as a basic hyperbase service.

Versioning is also a key issue in distributed hypertext systems [6, 24]. RHYTHM [19, 20] is a hypertext system that employs version control for distribution aspects. In RHYTHM, data is only referenced in the documents by so-called inclusions, although from the user's point of view the documents contain the data. Consequently, versions of hypertext nodes include the parts of the previous version that have not been changed. By keeping a version of the nodes at the end of each modification session, which may be considered a task-oriented version creation, changing spans of anchors of remote links in a distributed system can be computed by examining the development of the nodes over time. Palimpsest [7] is also based on fine-grained deltas and considers versioning for cooperative editing without synchronization like [30].

3 CoVer: A CONTEXTUAL VERSION SERVER INTEGRATING STATE-ORIENTED AND TASK-ORIENTED VERSIONING

The CoVer Contextual Version Server for hypertext applications is introduced in [11]. The main idea of the

[1] Østerbye uses the term composite as a generalization of contexts, a concept to provide hierarchical structure, and version groups, a concept to represent a versioned object. According to our terminology (cf. Section 3.1), Østerbye's contexts are the same as composite nodes and therefore we use the term composite node as a synonym for Østerbye's contexts.

CoVer version model is to automatically maintain information that arises in the context of version creation. An example of context information is that a version may be created as a reply to an annotation. This context information is stored persistently and supports version selection.

The aim of this section is to introduce an alternative characterization of CoVer: CoVer integrates state-oriented versioning concepts as well as task-oriented versioning concepts [10]. As discussed previously, state-oriented version models provide no support for task tracking and task-oriented version models offer no notion of individual objects. For example, Kay et al. [15] argue that operations on single objects should be executed in extra activities. While we think this approach is completely right for the implementation of version support (cf. Section 4), we argue that at the conceptual level it must be possible to identify versions from both perspectives: as individual versions of a versioned object (state-oriented access) as well as depending on a given task (task-oriented access; i.e., what is the state of the versioned object O as of task T).

3.1 State-Oriented Versioning in CoVer

CoVer supports versions of all constituents of a hyperdocument. A hyperdocument consists of three kinds of hypertext objects: atomic nodes, composite nodes and links. Atomic nodes contain data (e.g., text, graphics, images, sound). Composite nodes contain a set of references to other hypertext objects, called components of the composite. Composites allow the clustering of objects into subgraphs of the overall hyperdocument, that itself is represented by a composite. Links represent binary relationships between the hypertext objects.

All versioning concepts presented in this subsection may be characterized as state-oriented concepts. They allow one to build versions of complex hypertext networks by explicit operations.

To represent versioned hypertext objects, namely nodes, links and composites, CoVer uses the concept of a multistate object (mob). A *mob* represents a versioned object by gathering all states of the versioned object in its *version set.* A mob may be conceptually thought of as a composite holding references to all states of the versioned object it represents. The states of a versioned object are called *versions* and are represented by individual nodes, links, or composites. CoVer offers operations to create and update versions explicitly (cf. also Table 1).

To cope with references to versioned objects, we extend references to versioned objects to a two step addressing mechanism (cf. Figure 1). A reference to a versioned object consists of two values, an *object identifier*, that is a reference to a mob, and a *version identifier*, a reference to the version(s). Whereas the object identifier has to be a single static value, the version identifier may be a set of references to versions of the mob. The version identifier can also be characterized by a query, called *version description*, that evaluates dynamically to some versions of the mob.

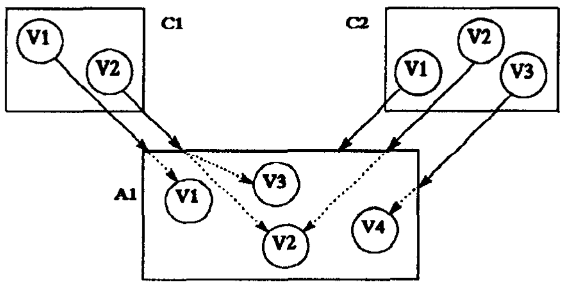

Figure 1. Two step addressing mechanism. Boxes represent mobs, circles in boxes represent versions of the versioned objects represented by the mobs. Object identifiers are indicated by solid arcs, version identifiers by dashed arcs. The versions of the composites C1 and C2 refer to versions of the atomic node A1. The first version V1 of C1 refers to the first version of A1. The second version V2 of C1 considers the second and the third version of A1 alternatives. The first version of C2 qualifies no version of A1 at all (empty version identifier).

If the version identifier contains several versions, we consider them *alternatives.* If the query of the version description returns several versions, we consider them alternatives with respect to the equality characteristic given by the query. If the version identifier is empty, no version of the referenced object fulfills the requirements of the reference of the referencing object.

Our notion of alternatives differs from approaches that define alternatives globally on the version set of a mob due to some absolute characteristics of the versions (usually their placement in a version graph). Since every object may be used by several composites or links for various purposes, we argue that alternatives have to be defined by the "user" of a versioned object, i.e., the concept that references the versioned object. Thus, any two or more versions are considered alternatives under a given abstraction formulated by a query. However, CoVer controls the *consistency of references* in a context given by a composite, i.e., all references in a composite to a certain versioned object refer to the same set of versions. In this way, CoVer secures that all links belonging to the composite and the references of the composite itself reference the same set of versions of a versioned component of the composite.

To preserve the states of versioned objects, versions of nodes, links, and composites can be *frozen* (cf. Table 1). We distinguish two internal modes of versions: updatable and frozen. The freeze operation is an explicit operation that saves the state of the version. Freezing a link or composite causes the evaluation of references to other hypertext objects and freezes the referenced versions implicitly.

To record the reuse of material across document boundaries, CoVer offers a so-called *derivation history.* The derivation history induces a (possibly) unconnected graph structure over all versions, independently of their structuring into mobs.

CoVer offers several operations to reuse content from one version in another: There are *derivation operations* to derive new versions or new objects as a (virtual) copy of a version (cf. Table 1), or operations to derive single attribute values from one version to another. On each derivation operation, a *derivation link* connecting the ascendant and descendant versions will be maintained by CoVer. The creation of a derivation link automatically causes a freeze of the ascendant version in order to preserve the derivation situation. The sum of all derivation links implements the derivation history.

All these concepts may be characterized as state-oriented concepts. They allow users to build versions of complex hypertext networks by explicit operations. To build a new version of a hypertext network represented by a composite, a new version of the composite has to be explicitly created and all the components and their concrete versions to be used in the composite have to be explicitly determined by the application. In addition, the derivation history may be characterized as a contextual versioning concept: it maintains the information that a new version or a special value of a version attribute has been created in the context of a derive operation.

3.2 Task-Oriented Versioning in CoVer

The task-oriented versioning concept of CoVer is task tracking: Users change their network to perform a job. Those jobs are represented by tasks. While mobs maintain the history of a single versioned object, the *tasks* supported by CoVer maintain the (versions of) various objects used and created in the context of performing a job. Eventually, a task maintains a state of the hyperdocument that fulfills the requirements of the respective job. Moreover, tasks form a task hierarchy providing a framework for the recursive decomposition of application processes. Keeping a log of all subtasks of a task, CoVer monitors the concrete work flow and supports task-oriented version selection.

To achieve these goals, the task concept is designed as follows: To create a new version of a specific hypertext object, tasks may profit from all available versions of all available hypertext objects in the overall system. Therefore, any version may be included into a task by an *include operation* provided by CoVer (cf. Table 1). If an attempt is made to update an included version, CoVer freezes the included version, derives a new version of the object and executes the update and all future updates on this derived version. Thus, all changes are kept automatically in derived versions.

Actually, to preserve the starting state of a task, the inclusion of an object causes its immediate freezing. So we may also say that CoVer performs ALL changes to a frozen object during a task on a new derived version.If a task will be completed by the application, its current state will also be frozen by CoVer. Thus CoVer creates and keeps versions in the context of performing a job.

Since the aim of a task - among other goals - is always to produce a consistent state of a hyperdocument, all versions contained in the task and referencing a certain versioned object will point to the same versions. Thus tasks show undecided questions or conflicts at all occurrences of the hypertext network. If a version will be included into a task, and the task already contains another version of this object, these versions will be considered *alternatives* in the overall context of the task. All referencing versions will be updated and if required, new versions of them will be derived.

Moreover, tasks perform an *automatic merge* for composites and links. If two versions of a versioned composite or a versioned link to be included into a task are the same with respect to all attribute values and their references to versioned objects and they only differ with respect to the concrete versions they refer to, they are considered *equivalent* and will be represented by a single derived version.

The inverse operation to inclusion is *exclusion*. At any time it is possible to exclude a version from a task. The version will then be removed from all versions of composites referring to it in the context of the task. This does not imply that the reference to the respective mob is deleted, i.e., the reference to the mob still exists, but possibly with an empty version identifier (cf. Section 3.1).

Tasks themselves may be organized in a *hierarchy* of super- and subtasks and within each level of the hierarchy composed of predecessor and successor tasks. An initial top level task maintained by CoVer records all tasks created by the applications. So, CoVer maintains a *task history*.

If a task has subtasks that are still open, it is suspended, i.e., work cannot be continued until all subtasks are completed and have delivered their results (see below). To support the spontaneous creation of subtasks during work processes, CoVer distinguishes *application tasks* from *change tasks* (cf. Figure 2). Application tasks are the only kind of tasks that can be defined by the applications (cf. Table 1). Change tasks are created automatically by CoVer to build up a complete record of coordinated changes.

Each application task consists at least of one change task that contains the changes performed in the application task. If application tasks have subtasks, those may be interspersed by change tasks. If subtasks are completed and the suspension of the supertask is resumed, new changes performed in the supertask are recorded in a new automatically created change task. In effect, a single designated so-called *current change task* of an application task contains the state of the application task (cf. Table 1). The use of the term task in the previous paragraphs precisely applies to an application task and its current change task, e.g.: "If an application task has application subtasks that are still open, it is suspended, i.e., work cannot be continued until all application subtasks are completed and have delivered their results.

The definition of predecessor and successor tasks relates application subtasks of an application task and the change task of an application task (cf. dashed arcs in Figure 2). A successor task by default includes all versions of its predecessor tasks. If the task has more than one predecessor, the versions will be merged applying the automatic merge operation described above. If a task has no predecessor tasks, it includes the versions of its supertask, except if the supertask is the predefined system task. In this case, the task is initially empty.

If a task is completed by the application, its current state will be frozen and delivered as a result to the supertask, i.e., the result will replace the respective versions of the supertask. If a task has several parallel subtasks, the results of these subtasks all together will be merged and replace the respective versions of the supertask.

One may argue that the automatic merge and the extension of the scope of alternatives (i.e., all referencing objects in a task point to the same versions of referred versioned objects) destroys the structure of the explicit alternative network created by predecessor respectively subtasks. But since the goal of a task is to construct an overall consistent state of the network, we pursue the approach to unify the alternatives as much as possible to identify open decisions and potential conflicts. If the alternative networks are to be explored, the respective version of the network maintained by the specific predecessor or subtask may be examined, either via the task history (task-oriented access) or by investigating the respective mob of the versioned object (state-oriented access).

3.3 Integration of State-Oriented and Task-Oriented Versioning in CoVer

To combine state-oriented with task-oriented versioning, versions created by the automatic version creation mechanism of tasks should be available to explicitly build versions of hypertext networks. Moreover, although the automatic version creation mechanism of tasks has been proven to be useful to maintain versions of highly interlinked hypertext networks [9], it should not prevent the explicit creation and maintenance of versions: At any point in time it should be possible for an application programmer and in particular for a user, to freeze a version explicitly, or to spontaneously derive an alternative of a version in the context of a task (cf. requirements in [11]).

To make versions created in the context of a task available for state-oriented version creation operations, all versions created by a task are also registered in the version sets of the respective mobs. Therefore, versions created implicitly by tasks may be used in explicit version creation operations to manipulate hypertext networks outside the scope of a task. Thus, applications are free to use tasks or not to use them.

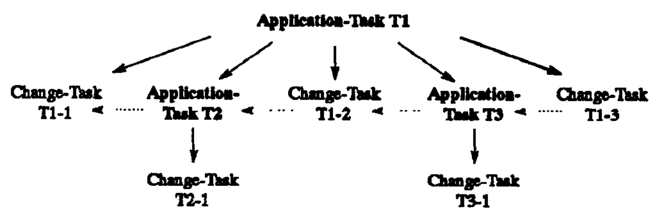

Figure 2. CoVer distinguishes application tasks and change tasks. The application tasks T2 and T3 are defined as subtasks of application task T1 (solid arcs). T2 and T3 keep their changes in the change tasks T2-1 and T3-1, respectively. The initial changes performed during T1 are maintained in the change task T1-1. Then, T2 was spontaneously created as a subtask of T1. After T2 comitted, additional changes performed in T1 were recorded in the change task T1-2. Once again a spontaneously created subtask, namely T3, was executed. The change task T1-3 keeps the changes executed in T1 after T3 comitted. T1-3 is the current change task of T1 (fat solid arc) and defines the current state of T1.

To integrate the task-oriented versioning concept with state-oriented versioning concepts, it is primarily possible to include versions into tasks, independently of whether they have been created explicitly or in the context of a task. Moreover, the exclusion may be considered a state-oriented operation. It addresses a state of a versioned object directly.

In addition, at any time a version in a task may be frozen explicitly. In order not to prevent the automatic version creation mechanism of tasks, new changes to the frozen object arising in the context of the task will be performed automatically on a derived version. But the frozen state will be kept in the respective mob and is not lost due to the subsequent update in the context of the task.

One may argue that the freezing of a version can also be modelled by a short subtask. While we think that from an implementation point of view this argument is completely right (cf. Section 4), we think that conceptually such an approach is misleading: The application programmer and in particular the users of applications are manipulating the individual versions in a state-oriented manner and this should be reflected appropriately by the version model.

Another state-oriented operation is that alternatives may be derived spontaneously from versions contained in a task.

As a summary, CoVer can be regarded as a contextual version model (derivation history, tasks, and annotations (not discussed here, see [11] for a discussion)) integrating state-oriented as well as task-oriented versioning concepts.

4 IMPLEMENTATION ISSUES

Regarding the implementation of version models, it turns out that the state-oriented implementation approach - representing every legal state of a hyperdocument explicitly - and the task-oriented implementation approach - computing versions of complex hypertext networks due to

Mob ***class methods*** createVesionedNode -> Mob createVersionedComposite -> Mob createVersionedLink (from: ALCV, to: ALCV) -> Mob ***instance methods*** getVersions -> {ALCV}	A single mob contains the versions of a versioned node, link, or composite. All creation operations create an instance of the mob and an initial version. A versioned link monitors the development of a single relationship, i.e., the versions of a versioned link have to point to versions of their initial "FromVersion" and "ToVersion".
Atomic Node Version (ANV) ***instance methods*** setAttribute (name: String, value: Object) -> Bool deriveVersion -> ANV deriveNewANV -> Mob freeze -> Bool getMob -> Mob	An attribute of a node may also be the content of the node. Next to the derive operations to derive a new version or a new versioned object, other derive operations are provided (cf. Section 3.1). Derivation Links (cf. Figure 4) are created implicitly by CoVer (cf. Section 3.1). For each version, the corresponding mob can be retrieved.
Link Version (LV) ***instance methods*** addFromVersion (ALCV) -> Bool removeFromVersion (ALCV) -> Bool addToVersion (ALCV) -> Bool removeToVersion (ALCV) -> Bool	In addition to the operations to update the link attributes, to freeze links, and to create or derive new link versions (not shown here), the version identifiers of the link references can be updated (cf. Section 3.2, Figure 1), i.e., versions of the fromObject and the toObject may be added or removed from the link references.
Composite Node Version (CNV) ***instance methods*** addComponent (Mob) -> Bool removeComponent (Mob) -> Bool addComponentVersion (ALCV) -> Bool removeComponentVersion (ALCV) -> Bool	Next to the operations to update the composite attributes, to freeze composites, and to create or derive new composite versions (not shown here), components can be added and removed (i.e., a reference to a versioned object can be established / destroyed) or the version identifier of a reference can be updated.
Task ***class methods*** defineTask -> AT getTopLevelTask -> AT ***instance methods*** defineSuccTask -> AT includeVersion (ALCV) -> Bool excludeVersion (ALCV) -> Bool getHyperdocuments -> (CNV, ANV) open -> Bool leave -> Bool join -> Bool close -> Bool	Application tasks may initially be defined as subtasks of the top level task maintained by CoVer (cf. Section 3.2). An application task may also be defined as a successor task of another application or change task. The predecessor and supertask links (cf. Figure 4) are automatically created by CoVer. The inclusion of a version into an application task always results in an inclusion to its current change task. In the same way, the hyperdocuments valid in a certain application task will be retrieved from the current change task or the current subtasks, respectively (cf. Section 4.1.2)
Application-Task (AT) ***instance methods*** getCurrentChangeTask -> CT defineSubTask -> AT	Application tasks may also be defined as subtasks of other application tasks, but never as subtasks of a change task. Therefore the corresponding method is provided for application tasks only.
Change-Task (CT) ***instance methods*** getApplicationTask -> AT	Change tasks are created implicitly when application tasks are created and updated and each change task has exactly one application task.

Table 1. Some methods of the CoVer version model. The left column shows some class and instance methods for some of the classes shown in Figure 4. The right column gives some explanations to some of the methods mentioned on the left. Method parameters denote instances of the respective classes. The abbreviation ALCV ({ALCV}) denotes either a (set of) atomic node version(s), composite node version(s), or link version(s), i.e., ALCV := ANV | CNV | LV and {ALCV} := {ANV} | {CNV} | {LV}).

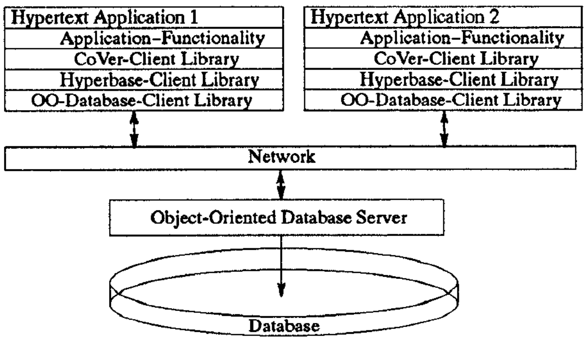

Figure 3. Integration of a version server into a hypertext application environment based on an object-oriented database. Relying completely on the functionality of an object-oriented database system, a multiuser hyperbase server ("Hyperbase-Client Library") and hypertext version server ("CoVer-Client Library") can be provided to support the implementation of various multiuser hypertext applications.

```
Object
  - Hypertext Object
    - Atomic Hypertext Node
      - Atomic Node Version
    - Composite Node Version
    - Mob
    - Task
      - Application-Task
      - Change-Task
    - Hypertext Link
      - Link Version
      - Derivation-Link
      - Predecessor-Task-Link
      - Super-Task-Link
```

Figure 4. Object Hierarchy of the State-Oriented Implementation

changes executed during a task - are interchangeable. While the separation of state- and task-oriented concepts at the conceptual level of the version model is desireable to support version creation and selection for different hypertext

applications, the implementation of such a dual model can be based on a single implementation concept.
We will discuss the state-oriented (Section 4.1) and the task-oriented implementation approach (Section 4.2) for CoVer's version model. For the respective implementation approach each section describes the implementation of state-oriented and task-oriented versioning concepts as well as the implementation of the integration of these concepts. In any case we assume that the implementation is based on a hypertext management system that has been realized on an object-oriented database system based on a client-server architecture (cf. Figure 3). Finally, we will sketch the actual implementation of CoVer (Section 4.3).

4.1 State-Oriented Implementation

The state-oriented implementation approach represents every legal version of a hyperdocument explicitly. A state-oriented implementation of CoVer's versioning model based on an object-oriented hypertext model is straightforward.

4.1.2 State-Oriented Implementation of State-Oriented Versioning

To implement the state-oriented versioning concepts (cf. Section 3.1) in a state-oriented implementation, the concepts of mob and version and the derivation links can be implemented as subclasses of the classes for nodes, links, and composites (cf. Figure 4).

A mob (an instance of "Mob" in Figure 4; Mob is also called the *mob class*) is a composite holding references to the versions of the versioned object it represents. Node, link, and composite versions ("Atomic Node Version", "Link Version", and "Composite Node Version" in Figure 4; also called *version classes*) are implemented as refinements of the respective hypertext classes ("Atomic Hypertext Node", "Hypertext Link", and "Composite Hypertext Node" in Figure 4). References in versions are extended to the two step addressing mechanism of CoVer and own a version description. Derivation links can be implemented as links. All operations on mobs, versions, and derivation links can be implemented as methods attached to the respective object classes (cf. Table 1).

From an application programmers point of view the additional mob class and the additional version classes in the hierarchy should be generated automatically from a declarative specification defining those types of application-defined hypertext objects (usually defined as subclasses of the general hypertext objects) that should be versioned.

4.1.2 State-Oriented Implementation of Task-Oriented Versioning

To implement the task-oriented versioning concepts (cf. Section 3.2) in a state-oriented implementation, tasks are introduced as special composites ("Task", "Application-Task", and "Change-Task" in Figure 4). The task relationships can be represented by special links ("Predecessor-Task-Link" and "Super-Task-Link" in Figure 4).

Application tasks consist of a set of change and application tasks which may be related to each other by predecessor task links (cf. Figure 2). In addition, application tasks that are not suspended indicate exactly one current change task.

A change task is associated to exactly one application task. A change task holds references to the versions of composites, nodes, and links its application task conceptually owns. If a composite is included into an application task, all its components are also included into the application task and kept by the current change task. Change tasks only have to point to the top most composite version and references of tasks to versions are usually few.

To support the task-oriented version creation mechanism, CoVer records for each client if it is working in an active task. There are operations for clients to open and close tasks and to join and leave open tasks. If a new version has to be derived to execute an update operation and the new version has to be integrated in (to be derived) versions of referencing objects, these new versions are represented explicitly in the version set of the affected mobs. In addition, the references of the task to the versions are updated and the client is informed about the replacement of versions.

If a task having several predecessors is opened the first time, the versions will be merged and the result will be represented by explicit new versions.

4.1.3 State-Oriented Implementation of the Integration of State- and Task-Oriented Versioning

Since the state-oriented implementation approach maps the conceptual versioning concepts directly onto implementation objects, the integration of state- and task-oriented versioning can be achieved as described in Section 3.3. Within a task, versions may be frozen or derived directly by the methods available for versions. To not prevent the automatic version creation mechanism of tasks, all updates to frozen versions within tasks will automatically be performed on derived versions.

4.2 Task-Oriented Implementation

The task-oriented implementation is motivated by the following observation: Though the versioning concepts of CoVer avoid the generation of unintended versions [9], a state-oriented implementation of the CoVer version model would cause the creation of a tremendous amount of objects. During a task new versions of links and composites belonging to the task have to be created due to the automatic generation of versions of the objects they refer to (cf. Section 4.1.2).

Figure 5 illustrates this problem by an example. Assume a task has included a version V1 of a composite C1 and this version contains two versions of two atomic nodes (V1 of A1 and V1 of A2) and a version of a link (V1 of L1). If a new version of a component, e.g., version V2 of A2, is generated and replaces the previous version, new versions of

C1 and L1 have to be derived to capture the state of the task and not to destroy the starting state of the task. If more complex networks are involved, all transitively referring versions are affected.

Although these new versions of links and composites are meaningful from a conceptual point of view their creation could be avoided from an implementation point of view. If a sufficiently detailed protocol of version creation operations is kept, the generation of new versions of links and composites due to the generation of new versions of referenced objects can be avoided. This is the idea of the task-oriented implementation: Only the new versions created in each task have to be recorded and the implied versions of referencing objects could be computed from this information.

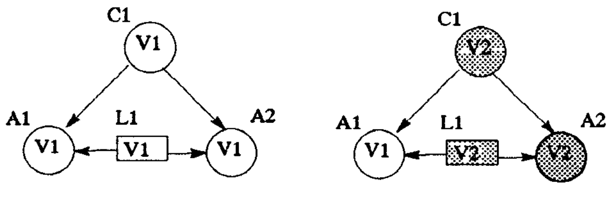

Figure 5. The creation of a new version of the atomic node A2 causes the automatic creation of new versions of the link L1 and the composite C1. The left side of the Figure shows the starting state of the task, the right side the subsequent state after creation of a new version of A2. Nodes are represented by circles, links by boxes, references to objects as directed arcs. The text inside the symbols denotes a version, the text next to the symbols the corresponding versioned object.

If this idea is extended to the level of object properties, versions as individual objects can even become superfluous: only the properties that have changed within a task are recorded and unchanged property values could be computed from previous tasks.

This idea can also be refined a step further to a property delta level: If a notion of delta for a certain property domain exists (e.g., a notion of delta for text) only the deltas for each property value have to be recorded and all property values could be computed from previous tasks. For example, keeping new versions of hypertext nodes after each session based on inclusions, like in RHYTHM [19, 20] or Palimpsest [7] could be considered as a task-oriented implementation at the property delta level. This paper neglects the latter implementation granularity and focuses on the former two approaches.

The two degrees of granularity of the task-oriented implementation approach discussed here are called *task-oriented implementation at the version level* respectively *task-oriented implementation at the property level*. The implementation at the version level keeps the changes at the granularity of individual versions whereas the implementation at the property level keeps the changes at the granularity of version properties.

In the following we will discuss the implementation at the version level and, if required, describe necessary extensions to data structures and algorithms for a task-oriented implementation at the property level. Since the task-oriented implementation approach stems from the task-oriented version creation mechanism, we discuss the implementation of task-oriented versioning (cf. Section 3.2) first.

4.2.1 *Task-Oriented Implementation of Task-Oriented Versioning*

The basis for the task-oriented implementation is a complete task history (cf. Figure 2). To realize the task-oriented implementation, the data structures of mobs and versions at the implementation level differ from the data structures at the conceptual level (cf. Figure 6).

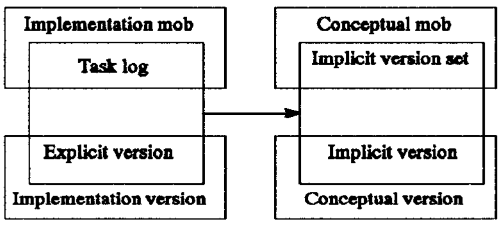

Figure 6. Implementation mobs and implementation versions together implement the conceptual versions and conceptual mobs offered at CoVer's programming interface.

Conceptually, references of versions consist of an object identifier and a version identifier. References of *implementation versions* consist of an object identifier only that points to *implementation mobs*. However, from the conceptual perspective (and thus for application programs), the interface to versions does not change, i.e., conceptually versions point to other versions and the application can access the version identifier in a reference transparently.

To achieve this behavior, internally the data management distinguishes explicit and implicit versions. The implementation versions are stored explicitly and point to implementation mobs. They are called *explicit versions*.

The conceptual versions point to other conceptual versions and are inferred for applications from the explicit versions in the context of a certain task. Since the complete task history and the set of all explicit versions together imply all conceptual versions, those versions are not stored explicitly. They are called *implicit versions*. The versions V2 of C1 and V2 of L1 in Figure 5 are examples of implicit versions. They can be derived from the presence of version V2 of A2 and the information, that V2 of A2 replaces V1 of A2 in the context of a task including V1 of C1 and all its components.

The data structure of implementation mobs differs from the data structure of conceptual mobs. The version set of an implementation mob contains explicit versions. In addition, an implementation mob owns a so-called *task log*. The task log is a stack of binary tuples called task entries. *A task entry* contains a single task identifier (i.e., a pointer to a task) and a set of references to explicit versions of the version set.

Figure 7a shows the implementation of the versioned composite C1 (cf. Figure 5), assuming that the versioned composite has been created (using the mob class method "createVersionedComposite () -> Mob", cf. Table 1) in the application task T1, respectively its initial change task T1-1 (cf. Figure 2). The implementation mob of C1 consists of a version set of explicit versions and a task log. The explixit version E-V1 of C1 contains the components A1, L1 and A2 (cf. Figure 5; those components have been added using the composite node version instance method "addComponent (Mob) -> Bool", cf. Table 1). The task log indicates, that E-V1 is valid in the change task T1-1.

Figure 7b shows the implementation of A2, assuming that the version E-V2 of A2 has been created during application task T2, respectively its initial change task T2-1. The task log indicates that E-V1 of A2 has been created in T1-1 and E-V2 of A2 in T2-1.

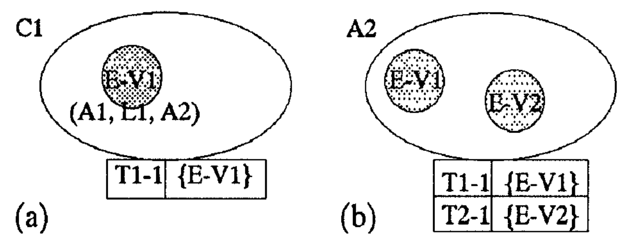

Figure 7. Task-oriented implementation of task-oriented versioning at the version level. Each implementation mob consists of a version set (ellipses) of explicit versions (all explixit versions carry an "E-" prefix to be distinguished from conceptual/implicit versions) and a task log (tables attached to the ellipses). The figure shows the implementation of the versioned objects C1 and A2 shown in Figure 5.

Together with the task information, the explicit versions determine all implicit versions of the versioned object in a certain task. All task entries together deliver the implicit version set, i.e., the version set containing all conceptual versions of a versioned object. Figure 8 illustrates the computation of the state of a hyperdocument as of a certain task. Using the *global task history*, the version(s) of a versioned object valid in a certain task T can be computed following the inclusion rules for tasks. The important point is that the application task history can be mapped onto a directed acyclic graph (DAG) of change tasks. The state of a hypertext object can be computed from this DAG of change tasks and the task log entries in the mob of the hypertext object. Actually, application tasks are higher-level concepts that identify meaningful versions of the complete change log kept by the change tasks. The application task hierarchy provides different levels of detail and abstraction for version selection (cf. [11]).

To compute the state of a hyperdocument as of a certain task, the rules described above are applied to the composite representing the hyperdocument. The identified explicit versions of the composite point to implementation mobs. The valid versions of the versioned objects represented by these mobs are computed in the same manner. In this way, the state of a hyperdocument in a certain task can be computed and delivered to the applications as implicit versions. The server stores with respect to what task each implicit version has been computed and which task entry and explicit version eventually determined the implicit version.

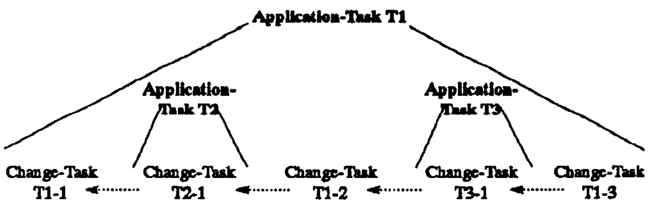

Figure 8. The predecessor relationships between application tasks and change tasks and the supertask relationships between application tasks can be mapped on predecessor relationships of change tasks. This Figure shows the mapping applied to the example task history shown in Figure 2, resulting in a linear order of change tasks. In general, considering parallel application tasks, the application task history can be mapped onto a directed acyclic graph of change tasks.

More formally, the valid state(s) of a hypertext object as of a certain change task, i.e., the valid alternatives of a hypertext object as of a certain change task, is
either defined by a task entry pointing to the change task in the task log of the respective mob,
or by the union of the states of the hypertext object as of the predecessor change tasks of the change task.

The state(s) of a hypertext object as of a certain application task is
either defined by the current change task of the application task,
or by the union of the states of the hypertext object as of the most recent parallel application subtasks.

For example, using the task history shown in Figure 8 and the example given in Figure 5 and 7, the state of C1 as of application task T2 is computed as follows: The change task of application task T2 is T2-1, i.e., the state of C1 as of change task T2-1 has to be computed. Since the task log of C1 does not contain a task entry for T2-1, we have to look for a task entry of the predecessor of T2-1, namely T1-1. In this way, the task log delivers E-V1 of C1 as the valid version as of application task T2 (cf. Figure 7). E-V1 of C1 has references to A1, L1 and A2. The valid versions of those versioned objects are computed in the same way. For A2, the task log delivers E-V2 of A2 as valid version as of T2. Considering the state of C1 as of change task T1-1 would deliver E-V1 of A2 as the valid component version. Thus, the conceptual version V1 of C1 as of T1-1 pointing to V1 of A1, V1 of L1, and V1 of A2 and the conceptual

version V2 of C1 as of T2, which is the explicit version E' V1 of C1 pointing to V1 of A1, V1 of L1 and V2 of A2, can be derived.
In case of updates in a task T, the task log has to be updated accordingly. Note, that the task log only contains pointers to the change tasks. Whenever an entry for the change task is present in the task log, the update can be performed on this entry. If there is no entry for the task, the versions of T have to be computed using the selection rules described above, a corresponding entry in the task log has to be created, and the update has to be performed on this new entry. In summary, the update rules for a version V of a versioned hypertext object in a specific task entry are as follows:

If *the version V is created* in T, add V to the entry.

If *the version V is explicitly included* (state-oriented operation) into T, freeze V and add V to the entry.

If *the version V is updated* in T, two cases have to be considered. If V is not frozen, update V. If V is frozen, derive a new version V', compute the changes on V' and replace V by V' in the entry of the task log.

If *the version V is excluded* of T, remove V from the entry. An empty set indicates that no version is valid in the task.

For example, change task T2-1 initially included V1 of A2 from T1-1 and an update of V1 of A2 during T2-1 resulted in the automatic derivation of V2 of A2 (third update rule mentioned above, cf. Figure 7).

4.2.2 *Task-Oriented Implementation of State-Oriented Versioning*

In the task-oriented implementation of task-oriented versioning, each task-entry defines the versions of a versioned object valid in this task for all referencing objects. To describe the individual references of a composite or link version due to state-oriented version creation operations, e.g., explicit adding of a component to a composite, each implicit version of a composite or link should be related to an extra task. These tasks indicate an implicit version for a certain composite or link and determines the versions of the objects the composite or link version refers to.
These tasks provided as placeholders for versions are called *version tasks* in contrast to the application and their change tasks discussed above. Essentially, version tasks play the role of short subtask proposed in pure task-oriented version models like Raleigh [15]. Version-tasks reflect the concrete references between conceptual versions similar to the relationships in the state-oriented implementation (cf. Section 4.1) or the relationships created at run time for implicit versions (cf. Section 4.2.1). Versions of referenced objects are represented as version subtasks, and derived versions are represented as version successor tasks.

The evaluation of the *version task history* compared to the *application task history* differs in the respect that version subtasks do not overwrite task entries made by their version supertasks and vice versa. Version tasks only describe the referenced versions for one object and not the referenced versions uniformly for the whole hypertext network, as application tasks do.
In the task-oriented implementation at the version level, no version tasks have to be provided for atomic nodes. Since atomic nodes do not depend on other hypertext objects, all conceptual (implicit) versions of atomic nodes are represented by explicit versions. However, for a task-oriented implementation at the property level, each version of an atomic node also has to be represented by a version task, since the computation of a version depends on the property values valid in a certain task. Each property of a versioned object could be equipped with a task log, defining the validity of property values in tasks. If an update on a derived version occurs, only the new values of properties affected by this update have to be stored explicitly. Thus, versions as objects can become superfluous.
Another feature of version tasks is that the explicit state-oriented freeze of a version results in closing the corresponding version task. This behavior is consistent with the behavior of application tasks, i.e., closing an application task results in freezing all versions of the task.

4.2.3 *Task-Oriented Implementation of the Integration of State- and Task-Oriented Versioning*

To integrate state-oriented and task-oriented versioning in a task-oriented implementation, application and version tasks have to be related to each other. The integration is based on the idea to keep all version creations in version tasks and to represent the occurrences of versions in application tasks respectively their change tasks as occurrences of the corresponding version tasks in the change task log.

So, for an implementation of the integration of state-oriented and task-oriented versioning, task log entries of change tasks point to version tasks. In an implementation at the version level, version tasks point to explicit versions. In an implementation at the property level, a task log entry for change tasks is kept with the object and task log entries for version tasks are kept at the property level, i.e., there are no versions as individual objects at all (cf. Section 4.3).

The explicit inclusion and exclusion of versions into tasks at the conceptual level results in adding or removing the respective version tasks from the corresponding application tasks respectively their change task at the implementation level. Freezing a version explicitly closes the corresponding version task and the version tasks of all referenced versions. Moreover, the affected version tasks will be connected as subtasks and supertasks according to the references between the frozen versions. Whenever an update to a frozen version occurs in an application task, a new version task for the derived version will be generated, replace the corresponding closed version task in the corresponding change task entry, and keep the changes. In this way, the frozen object is still kept explicitly in the version task history. The derivation of an alternative works in the same manner, i.e., a new version task will be created, having the version task of the ascendant version as a predecessor task.

4.3 Current Implementation of CoVer

The current CoVer prototype is implemented according to the task-oriented implementation at the property level. CoVer has been implemented as an extension of SFK (Smalltalk Frame Kit [27]), a frame-system based on Smalltalk. *Frames* [21] are similar to objects, except that each property, called *slot*, may have additional features besides a property domain restriction. These features are called *facets*. Since as a frame-based system the implementation of SFK focuses on the implementation of properties, the task-oriented implementation at the property-level of the current CoVer prototype suggested itself.

The schema definition language of SFK has been extended to distinguish versioned and non-versioned objects and to identify those properties (slots) of objects (frames), that are affected by versioning. (CoVer distinguished state-dependent and state-independent attributes, that have not been introduced in this paper. [11] contains a discussion of these concepts.) Each frame keeps a task log for change tasks indicating version tasks and for each state-dependent slot an extra task log for version tasks. Thus, mobs and versions are not explicitly represented as frames. A frame represents both the versioned frame and all versions of the versioned frame (cf. Section 4.2.3).

All read and write actions in SFK are performed in transactions. Read actions have also to be performed in transactions, since they may trigger the computations of inferred values in SFK. The identification of versions happens in the contexts of tasks and this task information is used by transactions to map each update on a correct entry of the task logs, following the rules described in Section 4.2.

SFK is also the implementation basis for the SEPIA [28] cooperative hypermedia authoring system. The implementation of CoVer in SFK extends CoVer's versioning model to frames in general. Hypertext objects as well as versioned hypertext objects are implemented as refinements of frames. Thus, the integration of version support into SEPIA (cf. [9], [11]) has been straightforward from an implementation point of view.

5 DISCUSSION

The O2 version model, the activity concept of Raleigh, and the version model of CoVer can be conceived as further developments of the PIE versioning scheme. PIE [8] keeps changes to objects in separate immutable layers that can be flexibly combined to contexts. A context defines a linear ordering of layers that defines the visibility and covering of versions among layers. Only one version for every versioned object is valid in a given context.

O2 (cf. [4], [5]) database versions can be viewed as layers. O2 imposes a fixed structure on layers to keep consistency in a versioned object-oriented database. Raleigh [15] and CoVer also impose a fixed structure on layers - but with the aim of workflow tracking. Therefore, the visibility rules between layers are extended to a hierarchy with a notion of successor and predecessor layers.

Whereas Raleigh's primary goal is to associate version, access, and concurrency control with activities, CoVer's primary goal is to enhance the possibilities of version creation and version selection in hypertext applications. For this reason, CoVer also chooses an extended notion of alternatives that has proven to be useful in particular for publishing hypertext applications, such as the SEPIA hypertext authoring system [9] or the TEDI hypermedia terminology editor [22]. In Raleigh, only one version per object is valid at a time. To compute the state of an activity that has several predecessors, the application has to specify which of the parallel activities should cover the other activities.

All other models only support alternatives of a whole network due to parallel layers. CoVer's notion of alternatives may be problematic in applications that require a definite single version of each object, for example to make computations on hypertext networks. Of course, our notion of alternatives requires that the applications can cope with alternatives.

If the occurrence of only a single version of each object would be a requirement in a hypertext application, a specific alternative task may be explicitly chosen, as in Raleigh or O2. However, in these cases the changes implied by alternative activities or alternative database versions are not merged at all by the version support system and the application has to cope with this merge problem. Consequently, O2 and Raleigh offer no merge support for alternatives.

Due to our notion of alternatives, version tasks have to be introduced as an implementation concept next to application tasks. While we think that at the network level alternatives of objects should be merged in the context of application tasks (cf. [9]), we believe that alternative property values should not be combined in an arbitrary manner.

Our notion of alternatives also requires the computation of all valid versions of a task T in case of update operations. If only one version per task is valid, as in PIE [8], O2 (cf. [4], [5]), or Raleigh [15], just a new entry for T has to be generated to overwrite the previous value[2]. This approach is not applicable to implement a version model using alternatives, since it would overwrite the presence of all alternatives and not just the occurrence of a single updated alternative. If the overhead in the implementation caused by the support of alternatives is considered to high for general version models, a general version server at the hyperbase level [12, 13, 29] might support a single state of each hyperdocument only.

[2] To indicate in these approaches that a version has been removed from the task, a special NULL-version or NULL-value overwrites the previous values. In CoVer, the empty set in a task entry indicates that no version is valid in a certain task.

However, as for the cooperative creation of hyperdocuments, management of alternatives by applications is an ultimate demand [9]. Will and Leggett propose to merge individual versions [30] in a branching version history kept to extend concurrency control. The version creation in tasks and the merge rules applied to tasks that have several predecessor or subtasks, may also be seen from this perspective. However, meaningful merging is a non-trivial problem that can best be solved by the user. The maintenance of alternative versions in the version model allowing to show all potential conflicts at all occurrences to the user - in particular by awareness mechanisms in cooperative hypertext systems [9] - is a first step to support the interactive merge process. A next step is to support the comparison of alternative versions (cf. Figure 3, p. 49 in [11]). Like in Palimpsest [7], the implementation of RHYTHM [19, 20] makes the computation of the precise deltas between subsequent versions obsolete. However, to detect what was going on and then to make meaningful merge decisions requires flexible "diff-ing" of documents, in particular for publishing applications such as cooperative writing [23]. Thus, keeping versions at the property delta level does not free one from providing various comparison algorithms to support meaningful merging.

The decision of how to handle alternatives is independent of the integration of state- and task-oriented versioning: In an integrated model it is possible to spontaneously freeze versions in tasks and to use versions created by tasks in state-oriented version creation operations. To our knowledge, CoVer's version model is the only version model that integrates state- and task-oriented versioning.

The choice between state-oriented and task-oriented implementation is influenced by space/time considerations. Whereas the state-oriented implementation offers direct access to all versions but uses a lot of space to represent each version explicitly, the task-oriented implementation saves space but requires the computation (at least) of versions of referencing objects. A concrete choice depends on the average size and amount of properties, the expected kind of version creation in favour of the application (i.e., state-oriented or task-oriented), and the preferred access to versions (i.e., via the version set of the versioned object (state-oriented access) or primarily via the task history (task-oriented access)).

6 FUTURE WORK

In the context of the project HyperStorM (Hypermedia Document Storage and Modelling) [1] we are currently implementing a new version of CoVer on top of the object-oriented database system VODAK. First of all, this implementation makes CoVer available for a variety of applications. Moreover, the aim of this project is to extend the CoVer version model for hyperdocuments to a general version model for objects (following a similar approach to the extension of the version model to general frames in SFK (cf. Section 4.3)), to explore the different kinds of implementations (state-oriented implementation, task-oriented implementation at the version level, and task-oriented implementation at the property level), and to estimate the impact of the different implementations on database operations, for example query processing [2].

Another line of future work regards versioning from the user's perspective [9]. Various interfaces for state-oriented and task-oriented version selection facilities, support for cooperative work and in particular cooperative interactive merge-tools for hypertext applications are subject to future work.

REFERENCES

1. K.Aberer, K.Böhm, Ch.Hüser. The Prospects of Publishing Using Advanced Database Concepts. In: Ch. Hüser, W. Möhr, and V.Quint (Eds.), Proc. of the 5th Int. Conf. on Electronic Publishing, Document Manipulation and Typography, Darmstadt, Germany, April 13-15, 1994. Electronic Publishing, Vol. 6, No. 4, 1994, pp. 469 - 480.

2. K.Aberer, Gisela Fischer. Object-Oriented Query Processing: The Impact of Methods on Language, Architecture and Optimization. Arbeitspapiere der GMD No. 763, St. Augustin, 1993.

3. S.Ahmed, A.Wong, D.Sriram and R.Locher. *A Comparison of Object-Oriented Management Systems for Engineering Applications. Massachusetts Institue of Technology.* Intelligent Systems Laboratory. Department of Civil Engineering. Order No. IESL90-03, 91-03, Research Report R91-12, May 1991.

4. F.Bancilhon, C.Delobel and P.Kanellakis. *Building an Object-Oriented Database System - The Story of O2.* Morgan Kaufmann Publishers, San Mateo, California, USA, 1992.

5. W.Cellary and G.Jomier. *Consistency of Versions in Object-Orienited Databases.* Proc. of the 16th Int. VLDB Conf., Brisbane, Australia, 1990.

6. H.Davis. Applying the Micorcosm Link Service to Very Large Document Collections. In [16].

7. D.G.Durand. Cooperative Editing Without Synchronization. In [16].

8. I.Goldstein and D.Bobrow. A Layered Approach to Software Design. In: D. Barstow, H. Shrobe, and E. Sandewell (Eds.), Interactive Programming Environments, Mc Graw Hill, 1984, pp. 387-413.

9. A.Haake and J.M.Haake. *Take CoVer: Exploiting Version Support in Cooperative Systems.* In: Proc. of INTERCHI '93 - Human Factors in Computing Systems, Amsterdam, The Netherlands, April 23-29, 1993, pp. 406-413.

10. A.Haake. *How to Integrate State-Oriented and Task-Oriented Versioning?* In [16].

11. A.Haake. *CoVer: A Contextual Version Server for Hypertext Applications.* In: D.Lucarella, J.Nanard, M.Nanard, P.Paolini (Eds.): Proc. of the Fourth ACM Conf. on Hypertext, Milano, Italy, November 30 - December 4, 1992, pp. 43-52.

12. D.Hicks. Version Control in the Hyperbase Management System Environment. In [16].

13. D.Hicks, J.Leggett and J.Schnase. Version control in hypermedia: An open systems perspective. Department of Computer Science Technical Teport No. TAMU-HRL 93-001,Texas A&M University, College Station, TX.

14. R.H.Katz. Towards a Unified Framework for Version Modelling in Engineering Databases. *acm computing surveys* 22, 4 (Dec. 1990), pp. 375-408.

15. M.H.Kay, P.J.Rivett and T.J.Walters. *The Raleigh Activity Model: Integrating Versions, Concurreny, and Access Control.* In: Advanced Database Systems. In: P.M.D.Gray and R.J. Lucas (Eds.). Proc. of the 10th British National Conf. on Databases, BNCOD 10, Aberdeen, Scotland, July 1992, pp. 175-191, Lecture Notes in Computer Science 618.

16. J.J.Leggett, Report of the HT'93 Workshop on Hyperbase Systems. Department of Computer Science Technical Teport No. TAMU-HRL 93 -009, Hypertext Research Lab, Texas A&M University, College Station, TX, November 1993.

17. J.J.Leggett and J.L.Schnase. Viewing Dexter With Open Eyes. CACM, Vol.37, No. 2, Feb. 1994, pp. 76-86.

18. A.Lie, R.Conradi, T.M.Didriksen, E.Karlsson. *Change Oriented Versioning in a Software Engineering Database.* In: Proc. of the 2nd Int. Workshop on Software Configuration Management, Princeton, New Jersey, October 24, 1989, pp.56-65.

19. C.Maioli, S.Sola, and F.Vitali. Wide-Area Distribution Issues in Hypertext Systems. SIGDOC '93, Kitcenet (Ontario Canada). October 1993

20. C.Maioli, S.Sola and F.Vitali. External anchors as a means of avoiding bottlenecks in the wide-area distribution of hypertext data. In [16].

21. M.Minsky. A Framework for Representing Knowledge. In: P.H.Winston (Ed.): *The Psychology of Computer Vision.* New York, McGraw-Hill, 1975, Chapter 6.

22. W.Möhr, L.Rostek, D.H.Fischer. *TEDI: An Object-Oriented Terminology Editor.* In: *Proc. of TKE '93 - Third International Congress on Terminology and Knowledge Engineering.* Cologne, Germany, August 25-27, 1993.

23. Ch.Neuwirth, R.Chandok, D.S.Kaufer, P.Erion, J.Morris, and D.Miller. Flexible Diff-ing in a Collaborative Writing System. In Proceedings of the ACM 1991 Conference on Computer-Supported Cooperative Work, Toronto, Canada, October 13 - November 4, 1992, ACM Press, pp. 147- 154.

24. J.Noll and W.Scacchi. Hypertext Object Management in Heterogeneous, Distributed Environments. In [16].

25. K.Østerbye. Structural and Cognitive Problems in Providing Version Control for Hypertext. In: D.Lucarella, J.Nanard, M.Nanard, P.Paolini (Eds.): Proc. of the Fourth ACM Conf. on Hypertext, Milano, Italy, November 30 - December 4, 1992, pp. 33-42.

26. V.Prevelakis, Versioning Issues for Hypertext Systems, in: D.Tsichritzis (Ed.),Object Management, Atélier d'Impression de l'Université de Genève, 1990, 89 - 105.

27. L.Rostek and D.H.Fischer. *SFK: A Smalltalk Frame Kit - Concepts and Use.* GMD-IPSI, Darmstadt, Germany, January, 1993.

28. N.Streitz, J.Haake, J.Hannemann, A.Lemke, W.Schuler, H.Schütt and M.Thüring. *SEPIA: A Cooperative Hypermedia Authoring Environment.* In: D.Lucarella, J.Nanard, M.Nanard, P.Paolini (Eds.): Proc. of the Fourth ACM Conf. on Hypertext, Milano, Italy, November 30 - December 4, 1992, pp. 11 - 22.

29. U.K.Wiil and J.J.Leggett. Hyperform: Using Extensibility to develop Dynamic, Open and Distributed Hypertext Systems. In: D.Lucarella, J.Nanard, M.Nanard, P.Paolini (Eds.): Proc. of the Fourth ACM Conf. on Hypertext, Milano, Italy, November 30 - December 4, 1992, pp. 251-261

30. U.K.Wiil and J.J.Leggett. Concurreny Control in Collaborative Hypertext Systems. In.: Proc. of the Fifth ACM Conference on Hypertext, Seattle, Washington, USA, November 14-18, 1993, pp. 14-24