ABSTRACT
One promise that has always been made in the field of e-learning is the possibility to create and deliver learning material that is adaptable to individual learners. Realising this vision requires the possibility to reuse existing learning objects, combining them in new contexts. Reuse, in turn, can only be supported, if a flexible versioning scheme is in place that can easily handle multiple versions of objects.

In this paper, we analyse the requirements of e-learning systems for versioning support. Considering well-known design spaces for versioning systems we develop a versioning scheme that perfectly meets the requirement list. This so-called implicit versioning scheme allows for transparent versioning and reuse of arbitrary content objects. We explicitly stress on the integration of the versioning schemes into the authoring process of e-learning material and discuss its effects on the usability.

KEYWORDS
Learning Objects, Reuse, Implicit Versioning, Version Conflicts, Conflict Resolution

1. INTRODUCTION
Throughout this paper we use the term e-learning for computer based learning environments that deliver the learning material to learners via a network, preferably the Internet and the World Wide Web. From its early stages, e-learning was considered to have the following advantages: (1) making learning material available anytime and anywhere, (2) adapting material to the learner’s individual needs, and (3) being much more cost effective than traditional classroom trainings. Over time, however, it became apparent that the production of e-learning material is quite expensive (Bacsich et al., 99). Due to the absence of a physical trainer and co-learners more emphasis needs to be put on the quality of the learning material making it self-understandable and attractive. For achieving the promise of adaptive learning material and for reducing the production costs to an acceptable amount, the reuse of already existing learning material is fundamental. The ability to reuse existing learning objects enables the flexible recombination of them for new contexts or for learners with special needs. The need for reuse of learning objects even between different learning systems has been also recognized by emerging standards like SCORM (ADL, 01). However, flexible reuse requires a versioning scheme that can easily handle multiple versions of objects.

Versioning of data objects is an issue in many areas like software development, document management and Web site management. Versioning systems provide a platform and tools to archive the history of an authoring process, to control concurrent authoring operations and to specify object version sets (so-called configurations or structures) that form a larger logical unit in some application dependant sense. Hypertext versioning systems especially address the issue of managing links between versioned objects.

This paper analyses the application domain e-learning and its needs on the underlying content management system and its versioning capabilities. Technically, e-learning content constitutes a huge hypertext that aggregates single content modules to courses. Therefore, e-learning systems have similar requirements to hypertext versioning systems. As we will show in the next section, the e-learning domain has 2 special aspects that make it different from most other domains: Firstly, the set of supported document formats
should be almost unlimited. Secondly, preserving link consistency poses some special problems, as several versions of the same object can be valid and used simultaneously at the same time. In this paper, we explore a versioning scheme that takes these special aspects into account and show how it can be applied in an authoring environment to assist authors in defining and controlling object reuse at a version level, both easily and flexible. The proposed versioning scheme does not constitute a radically novel approach for hypertext versioning. Instead, it is based on well-known versioning techniques and adapts them to the special needs for the field of e-learning.

The remainder of this paper is organised as follows: After discussing some fundamental requirements of e-learning authoring environments and related work in Section 2, we present an implicit versioning scheme in Section 3 emphasising the problem of version conflicts which might harm link consistency. The authoring process build upon this scheme is presented in Section 4. We discuss our approach and conclude with a summary and a brief outlook on future work in Section 5.

2. REQUIREMENTS AND RELATED WORK

2.1 System Architecture & Content Structure

We begin with the definition of a generic abstract architecture of an e-learning system: The authoring station enables authors to create, view and change any e-learning content. An unlimited set of distributed authoring stations may exist each providing a local repository to work with. The master repository centrally stores all content created by authors and delivers it to learners. The learning management system controls that delivery in a personally adapted and pedagogically reflected way. The learning station is the front-end of the learner in which she can view and navigate learning content.

E-learning systems may structure their content at various levels. At least 3 structural levels can be distinguished:

- **Media objects** are the most general (singular or compound) objects and range from multimedia, video or audio files up to style sheets or objects for storing meta-data.
- **Learning objects** are the smallest units that cover a distinct topic in a thematically coherent way. Learning objects contain several media objects that actually display the content.
- **Courses and sub courses** are used as containers to aggregate several subordinate learning objects or sub courses.

Some systems may use more fine-grained structure levels; however, the precise layout of these levels does not influence our further discussion.

A version (or revision) is a snapshot of an object in its evolution. A versioned object is independent of any specific revision but contains the revisions of the object it signifies. A link associates a set of objects. A revision selection rule is an expression that is evaluated to select one revision of a versioned object. We say that an object A directly references object B, if any file within A contains a link to any file within B. Note that link detection is beyond the scope of this paper. We assume that such links are either detected automatically by appropriate tools or have been specified manually by the author.

Given this system architecture, we assume that a versioning system supports the following operations: checking-in of locally edited object versions into the master repository (thus turning them into persistent versions), copying of versions from the master repository to a local one, editing read-only versions which creates a new writable version in a local repository, reverting edited versions to their predecessor, and deleting versions in a local repository.

As long as authors do not need to move objects to or from the master repository, they can work in their local repository without being connected to the master repository. We call this work model offline scenario. Opposed to that, working in the local repository while being connected to the master repository is termed online scenario.
2.2 Requirements on Reuse and Versioning

Following the experiences gained from a large field study (Ehlers et al., 03) we identified the following requirements to be essential for any e-learning system that claims to meet the vision of flexible reuse:

F1. Usage of arbitrary content editors and formats. As content production is very expensive, we cannot rely on a single content editor and format. Instead, we should support any editor and format that can be displayed by the learning station. In combination with requirement F4, this allows content to be reused that has been developed in other system contexts.

F2. Reuse at arbitrary granularity. Object reuse should be enabled at all structural levels, starting from atomic content objects like single style sheets and media files up to complete sub-courses. This also corresponds to the definition of reusable learning objects by (Wiley, 00).

F3. Stable reuse of specific object versions. As we learned from a couple of projects (Theilmann, 02), (Theilmann & Altenhofen, 03) authors who reuse objects from other authors require those objects to remain stable. They want to refer to a specific version and want to be sure that this version will never change. However, this must not prevent authors from creating new versions of objects they (re)use.

F4. Import of external content. Many content assets are developed outside of the respective e-learning system. It should be easy to import such content into the system and to build reusable objects from that content.

These requirements have some important consequences: (1) Supporting arbitrary content editors requires a common storage system abstraction. The only feasible abstraction is the file system. Thus, the local repository must be built upon the file system and all important authoring processes like creating, changing and viewing content have to work on the file system. (2) The organisation of the local repository must be human readable as the authors have to use the built-in file-chooser of their respective content editor to create any cross-references to other documents.

In addition to these fundamental requirements, an authoring station should support/respect the following usability aspects:

A1. Easy update/adaptation to new/other versions. Authors that refer to explicit object versions need assistance when changing these references to other object versions.

A2. Collaboration among authors. As any other versioning system, an authoring station should provide means for viewing, using or even changing objects developed by other authors.

A3. Offline creation of new versions. An important convenience for authors is the ability to create new objects (versions) even in an offline scenario.

2.3 Related Work

The Sharable Content Object Reference Model SCORM (ADL, 01) is explicitly foreseen to enable reuse of learning objects even between different learning systems. However, it focuses on the exchange of material and assumes that all learning objects are completely self-contained. This restricts the reuse granularity making it hard if not impossible to share common objects (like, e.g. global style sheets) among learning objects.

Many research systems, such as JaTeK (Borcea et al., 00) offer reuse at all granularity levels. However, they often support only a small set of document formats. In addition, there is no versioning support. While the WINDS system (Specht et al., 02) is more open to a larger set of document formats it has no versioning support. Others (Ateyeh & Mülle, 02) restrict the set of reuse objects to a higher granularity level as SCORM.

Even well known commercial products such as the Hyperwave eLearning Suite (Hyperwave, 02a) do not offer support for flexible reuse. Hyperwave provides document parsing and link adaptation for a large document format set. However, references to specific object versions are not supported.

Design spaces for link versioning have been extensively explored by the research community. According to (Whitehead, 01) they can be described in 4 dimensions: The containment design space basically describes how objects contain each other (by reference/inclusion, ordered/unordered, single/multiple containment). The revision history design space describes different approaches for storing revisions (as individually addressable objects referenced by the versioned object container, as parts included by their versioned object or with independent revisions linked by designated link objects). The link versioning design space distinguishes links
as independent system objects and links as inner part of content objects. Finally, the structure versioning design space describes how to realize a structure container that contains a link set and that is versioned itself (selection of contained elements like links, revision selection rules or content objects; revision history type of containees; containment type of containees).

Traditional versioning systems like RCS (Tichy, 85) and many content management systems that are based on them treat all files/resources individually. They neither provide means for managing compound objects nor for managing dependencies (links) between objects. Sets of version instances can be described only by explicitly marked configurations or implicitly by the current head revisions.

Modern Web Content Management Systems (e.g. (Hyperwave, 02b), (Twiki, 02)) provide basic means for link management. E.g. before deleting a version they offer to check the link consistency, i.e. whether any other object refers to the deleted object. Of course, such checks only work for a limited set of file formats that can be parsed. At least files containing any source code (e.g. JavaScript) that dynamically computes cross-references cannot be analysed that way.

The emerging versioning scheme DeltaV (Clemm et al., 02) adds versioning support to the WebDAV protocol (Goland et al., 99). In contrast to traditional versioning systems, DeltaV allows to address single revisions by a URL (Uniform Resource Locator), thus enabling direct delivery of single versions via the Web. DeltaV does not address object dependencies.

(Hicks et al., 98) presented a framework for hypertext versioning systems. They concentrate on the core versioning scheme and do not consider the overall user process (what they call the version policy).

To our best notice, the most similar approach on hypertext versioning has been done by (Soares et al., 99). They introduce the concept of perspectives under which object dependencies can be seen. This resembles very much our approach of cascading conflict resolutions. In addition, they investigate the effect of versioning operations on single objects on related (e.g. referencing) objects. However, they abstract from different content formats and concentrate on different object representations realised as versions.

3. AN IMPLICIT VERSIONING SCHEME

Ideally, a versioning scheme in the e-learning domain should meet all of the aforementioned requirements. At the same time, usage of the scheme (and the authoring process build upon it) should be as simple as possible. Therefore, we made some design decisions that restrict the flexibility of our versioning schema (compared to the possible design spaces of hypertext versioning systems).

First of all, we decided that all objects in our system are versioned. We support singular objects (containing only a single file) and compound objects (containing a set of files), thus we satisfy requirement F2. In conformance with requirement F3, versions that have been checked into the master repository are unmodifiable, i.e. cannot be changed anymore. Versioned objects are realised as containers that have references to all of their versions. Object versions can reuse each other by referential containment. This meets the requirements F2 (reuse at arbitrary granularity), F3 (reuse of specific versions) and A1 (easy update by simply changing a reference).

3.1 Link Versioning

A major problem in hypertext versioning systems is link versioning. In the e-learning domain links occur as inherent parts of the documents. Authors create such links according to their view on their local repository, i.e. their local file system. Due to the unlimited set of document formats (see requirement F1) we cannot assume to be able to parse all these formats not to mention the adaptation of their cross-references. For this reason, we discarded approaches that rely on link analysis and adaptation.

The basic idea of the implicit versioning scheme is to store information about the version instance of a referenced object not directly within the reference but in the meta-data of the referring object. This allows versions to be stored in the local repository without having any version information attached to the object’s location.

We achieve this by introducing the concepts of logical and physical object locations. A logical object location does not contain any version information. Instead, each version of a versioned object has the same logical location. Within a local repository, the physical location and the logical location are identical. Within
the master repository, however, the physical locations obviously differ from the logical one, as the master repository has to store all versions of an object. Links within content files always point from the logical source location to the logical target location.

However, this approach has some effects that are worth to be mentioned. Firstly, at any point in time, only one version of a versioned object can be stored and used in the local repository. Consequently, each course may contain a reusable object in at most one version. Secondly, the master repository has to be aware of the distinction between logical and physical object locations when delivering them to a learner station. Depending on a given context, it must be able to lookup the physical object that corresponds to a logical object.

### 3.2 Version Conflicts

If at most one version of a versioned object may occur in a local repository, but objects are supposed to be reused, a number of scenarios exist in which the standard operations of a versioning system would result in storing or using more than one version of a specific object within that local repository. We call such a situation a *version conflict*.

The issue that comes up with this problem is to detect version conflicts and to resolve them in an appropriate, hopefully automatic way. Considering usability and performance issues, this detection should be done as early as possible while not overloading the overall system with permanent check processes.

#### 3.2.1 Conflict Invariants

To achieve this goal we define 2 conflict invariants that must be preserved by each interaction with the versioning system:

1. Each version within the master repository must be conflict free, i.e. must not (directly or indirectly) refer to any other object in more than one version.
2. Each local repository must be conflict free, i.e. must not contain any versioned object in more than one version, where “contains” means that a version is either physically stored in the local repository or is referred from another, locally stored object.

Instead of demanding universal conflict freeness, the separation into two invariants makes the conflict checking process much easier and much more efficient. Invariant 1 must only be checked, when new objects are brought into the master repository. Invariant 2 can be checked completely locally even in an offline scenario.

#### 3.2.2 Cascading Conflict Resolutions

Consider the following scenario (depicted in Figure 1): Author A has created some versioned object X in version 1 (X-1) that uses another object Y in version 1 (Y-1). Author B is creating an object Z-1 that uses object Y in version 2 (Y-2) and now wants to use object X-1 as well.

![Figure 1. Example (a) and solution (b) for a cascading conflict resolution](image)

Given invariant I2, author B cannot use version X-1 as this would cause a version conflict. One solution would be to create a new version X-2 that would refer to Y-2. However, this would contradict the idea of reusability, as the reused object version would need to be changed. In addition, this would make the update to new versions of object X provided by author A extremely difficult.
To solve this problem, we introduce the concept of *cascading conflict resolutions*: A cascading conflict resolution defined in a superior (referring) object overrides the version information about references stored in inferior (referred) objects. In the example above, this would result in a conflict resolution assigned to version Z-1 that would point to object version Y-2. Note that this concept requires that each check of the conflict invariants has to consider cascading conflict resolutions when computing referred versions within a specific context.

4. **AUTHORING PROCESS**

The authoring process based upon the implicit versioning scheme has to take care about 2 aspects: First, it has to maintain object’s meta-data which contains version information about referenced (depending) objects. Second, it must check for and resolve version conflicts during various repository actions.

4.1 **Updating Reference Version Information**

While cross-references between files never contain any version information, object references contain both, the relative path to a logical object location and the identifier of a specific version. This information is stored in the meta-data of the referring (source) object. Storing and updating version information is done transparently to the user based on the versions that exist in the local repository. Basically, an object’s meta-data is updated whenever the references within that object have changed (due to changed content files or due to an explicit reference assignment). For each referenced object, the system checks if the object exists in the local repository. If so, the meta-data of the referring object is updated accordingly. As new references can only originate from objects in the local repository, this process guarantees to have the complete version information about all references of a given object.

4.2 **Preserving the Invariants**

Each repository operation that either creates or deletes a version may violate a conflict invariant, and thus requires a conflict check. More precisely, the following operations may affect the local conflict invariant I2:

- Editing a previously checked-in object version, this actually creates a new version.
- Reverting a new object version to its predecessor version.
- Copying any object version from the master repository to the local repository.

The only action that may affect the master conflict invariant I1 is:

- Checking-in object versions from a local repository to the master repository.

Note that creating new objects (in their initial version) and deleting checked-in object versions has no effect on the conflict invariants.

In the following sections, we will describe how each of these critical repository operations is embedded into a corresponding process that preserves the invariants.

4.2.1 **Edit Process**

Whenever a checked-in object version is edited in the local repository, a new object version is created. As all versions of an object do have the same physical location, this object version replaces its predecessor in the local repository (in conformance to conflict invariant I2). All objects in the local repository with a link to the predecessor version have to be adapted. This adaptation is simply done by updating the referring objects’ meta-data with the version information of the newly created version. Note that this may require a (recursive) edit process on additional checked-in object versions.

4.2.2 **Revert Process**

Reverting a new object version in the local repository is done by deleting that version. All objects in the local repository with a link to the reverted version have to be adapted to the version predecessor in the same manner as it is described for the editing of an object.
In case the author is in an online scenario, she has the option to load the predecessor version to the local repository. This is equivalent to copying object versions from the master to the local repository (described in the next section). Note, that this loading can also cause version conflicts as described below.

4.2.3 Copy Process

An object version that is copied from a master repository to a local repository can cause version conflicts in various (non-exclusive) ways:

C1. Another version of that object already exists locally.
C2. A local object refers to another version of that object.
C3. The object contains links to other objects that conflict with local objects or local object references.

Note that this check is only necessary, if the copy object does not induce a direct conflict (C1, C2) or if that object version has been selected to resolve a previous conflict resolution.

All these kinds of version conflicts have to be resolved interactively by asking the user for selecting her preferred version. Ideally, this selection process allows the user to preview the selected version and to preview the actions (i.e. the influenced local objects) that would result from the selection.

The resolution of the 3 conflict types is done in the following way: (C1) If the user selects the local version nothing needs to be done. Otherwise, the local version has to be deleted. (C2) If the user selects the local version nothing needs to be done. Otherwise, all local objects with references to versions conflicting with the copy version must be adapted. (C3) If the user selected the reference of the copy version, a local conflicting version must be deleted and all local conflicting references must be adapted as in C2. Otherwise, this reference must be adapted in the copied version as in C2.

4.2.4 Proxy Conflict Resolutions

There is still one scenario that limits the reuse capabilities for an author. Suppose (see also Figure 2) an author created object Z-1 that relies on object X-1. Then, he wants to incorporate object Y-1 developed by another author into his course. Suppose also that Y-1 references X-2. When the author copies Y-1 to his local repository a version conflict is detected. If the author wants to stick to version X-1, the copy process will create an object version Y-2. This contradicts the initial idea of reuse, as the author needs to change Y-1 (creating Y-2) even if he only wants to reuse it. Basically, we have the concept of cascading conflict resolutions for solving such problems. However, this concept is not applicable in this scenario because object Y-1 is not yet referenced from object Z-1 and we cannot assign a cascading conflict resolution to Z-1.

Therefore, we introduce the concept of proxy conflict resolutions. These resolutions are to be used as a proxy until an ordinary cascading conflict resolution can be assigned. A proxy conflict resolution is stored in the local repository’s meta-data. It consists of a target object (an object that induced a version conflict by one of its referred objects) and a conflict resolution (a version that is to be used for all objects recursively referenced by the target object). A proxy resolution is converted into a cascading resolution when its target object is edited. A proxy resolution is adopted when its target object is referenced by another object. If the referring object has been edited, the adoption is done via a cascading resolution. Otherwise, another proxy resolution is created with the referring object as the target object and the same conflict resolution that is used in the original proxy resolution.

In the above example, we would create a proxy resolution for target object Y-1 that points to object version X-1. When object Y-1 is referenced by object Z-1, this proxy resolution is adopted by object Z-1 so that object Z-1 gets a cascading conflict resolution that points to object X-1.
The concept of proxy resolutions is applied in the copy process, described in the previous section. Instead of editing any local object (which might have unwanted effects on the reusability) we create proxy conflict resolutions for the respective objects.

Like ordinary cascading conflict resolutions the proxy conflict resolutions must be considered when checking the local conflict invariant.

4.2.5 Check-in Process

The only operation that changes the state of the master repository and thereby influences the conflict invariant I1 is checking-in new object versions from a local repository into the master repository.

To assure invariant I1 we enforce that an object version must always be checked-in together with all its (recursively) referenced objects. Without this assumption, the conflict check would be incomplete as local referenced objects could still be changed after the check-in process has completed.

Conflicts may result from the fact that we do not enforce that all objects referenced by a local object version have to exist in the local repository. For example, we do not enforce an author to store large video files on his local file system when he only wants to edit some other small parts of the same course. Therefore, it may happen that a local object indirectly references different versions of the same object but that this has not been detected by the invariant I2. An example scenario that leads to such a situation is described in (Theilmann & Altenhofen, 03).

To detect and resolve all version conflicts at check-in time the following observations provide an elegant solution: If for a given check-in object all recursively referenced objects would exist locally, no version conflict could happen. Consequently, if we would start a copy process for all locally missing objects, all potential version conflicts would have been resolved by this process. Therefore, the check-in process simulates the copy process for all referenced, but locally missing objects. The only difference between the simulation and the true copy process is that no objects are actually copied to the local repository. Apart from that, all checks and resolution dialogs are identical. This procedure ensures that eventually the check-in objects are globally conflict free, i.e. they do not cause any conflict with any of their referenced objects. Thus, the check-in can be done without violating conflict invariant I1.

4.4 Dealing with Open Proxy Resolutions

Proxy conflict resolutions may result in a slight inconsistency between a local and the master repository since they have no counterpart in the master repository. Therefore, object references of checked-in objects may differ between the local and the master repository. Basically, this does not cause a problem unless the proxy resolution is assigned to a root object. Basically, a root object is an object that is not referenced by other local objects. In this case, an author might think he has finished his job and deletes all locally stored content. However, starting the course from the master repository would have an unexpected result. Therefore, we automatically check all root objects for proxy resolutions. If such an object exists in the local repository, a special flag is shown to the author indicating that there is a mismatch between the local and the master repository. Then, the author can edit the respective object, thereby convert the proxy resolution into an ordinary cascading conflict resolution, and finally check the new object version into the master repository.

5. CONCLUSION

The implicit versioning scheme perfectly meets the requirements F1-F4 and A1-A3 for an authoring station. Through the automatic updating mechanism (Section 4.1) authors automatically select that object version for reuse in their course context that they have loaded into the local repository. In other words, it is a perfect WYSIWYG (what you see is what you get) approach (see F3). As local objects are always stored in logical locations (Section 3.1) external content can be directly imported into a local repository. Reusable objects of arbitrary granularity can be created from the imported content without changing any content links (see F2 & F4). The adaptation to new/other object versions is implicitly done when copying the respective version to a local repository. No links within content files need to be changed when switching to another version of a referenced versioned object (see A1). Finally, updating reference version information and creating new versions (via the edit process) can be done offline (see A3).
Authors need to consider versioning aspects fairly seldom. Most of the required operations for version detection and update are automatically done by the system. If the author always keeps the complete course in the local repository a version conflict that needs to be resolved by the author can only happen when copying new object versions from the master repository. Otherwise, version conflicts may occur at check-in time though we expect this to happen very seldom. Another situation in which authors are bothered with versioning aspects are proxy resolutions assigned to root objects. Again, we expect this to occur extremely rarely and experiments with pilot authors have confirmed this assumption.

The implicit scheme bears the well-known restriction that each versioned object can be used at most in one version within the local repository. To overcome this restriction, we developed the concept of workspaces that partition the local repository. Each workspace may contain object (versions) independently of the objects within other workspaces. More details on workspaces can be found in (Theilmann & Altenhofen, 03).

It is worth mentioning that the approach of implicit versioning offers further flexibilities for object reuse. As content links do not contain any version information, logical reuse concepts can be easily introduced. For example, links can be resolved against the most currently released version; authors can restrict the reuse of their objects to a certain subset and can change these restrictions over time. Future work should concentrate on such logical reuse concepts and their embedding into the authoring process.

REFERENCES


Twiki, 2002, A Web Based Collaboration platform. see also URL: http://twiki.org

Wiley, David. A., 2000, Connecting Learning Objects to Instructional Design Theory: A Definition, a Metaphor, and a Taxonomy. The Instructional Use of Learning Objects, Agency for Instructional Technology, Bloomington(IN), USA

Whitehead, E. James, 2001, Design spaces for link and structure versioning. Proceedings of the twelfth ACM conference on Hypertext and Hypermedia. Århus, Denmark, pp. 195-204