Fine-Granular Version Control & Redundancy Resolution

Normen Müller and Michael Kohlhase
Jacobs University Bremen
D-28759, Bremen, Germany
{n.mueller,m.kohlhase}@jacobs-university.de

Abstract

We propose an abstract theory of collaborative content management and version control for collections of semi-structured documents. Our *fs*-tree model generalizes version-controlled file systems and XML files alike, which allows us to specify and implement version control algorithms that seamlessly work across the file/file system border.

An added value of this model which incorporates symbolic links and XML inclusions as first-class citizens, is that we can significantly enhance workflow consistency, space-efficiency, and processing speed on working copies by eliminating link copy redundancy.

To evaluate our model and algorithms we have implemented a prototype SUBVERSION client based on the ideas put forward in this paper. The *locutor* system has been heavily used in day-to-day work and bears out the expected space efficiency and consistency gains.

If I had eight hours to chop down a tree,
I'd spend six sharping my ax.
— Abraham Lincoln

1 Introduction

In the last years we have seen sustainable growth in mathematical content management systems [Franke and Kohlhase, 2000; Allen *et al.*, 2002; Asperti *et al.*, 2001], as well as publication and education systems [Baraniuk *et al.*, 2002; CNX, 2007] for mathematics. Elaborated representation formalisms, such as MATHML [Ausbrooks *et al.*, 2008], OPENMATH [Buswell *et al.*, 2004] or OMDOC [Kohlhase, 2006], have been established to reflect the well-structured formalized mathematical information. However, Mathematical Knowledge Management (MKM) techniques for building up repositories of highly hierarchical structured and semantically interrelated mathematical knowledge lag behind. In practice we make use of version control systems and XML databases for this, since they promise to provide distributed *access* to mathematical knowledge, and on the other hand simplify the *creation* of mathematical knowledge, which is often a distributed and collaborative process. In this paper we concentrate on version control systems: XML databases already provide static, non-versioned, "knowledge access", but the dynamic, i.e. chronological, "knowledge creation" scenario is arguably more important for MKM systems, as knowledge can only be accessed after it has been created.

The core of most version control systems [Fogel and Bar, 2003; Collins-Sussman *et al.*, 2004; DARCS, 2008; GIT, 2007] is a central *repository*, which stores information in form of a file system tree. The question for these systems is how to allow users to share information, but prevent them from accidentally stepping on each other's feet? The most widely used solution is the *copy-modify-merge* model, where each user's client creates a private *working copy*, i.e. a local reflection of the repository's files and directories. Users can then work simultaneously and independently, modifying their working copies. The private copies are periodically committed to the repository, allowing other users to merge changes into their working copy to keep it synchronized.

In this paper, we are interested in version control for large and complex projects where it is often useful to construct a working copy that is made out of a number of different projects. For example, one may want different subdirectories to come from different locations in a repository, or from different repositories altogether. Most version control systems provide support for this via *external definitions*.

In MKM this model only partially applies, though. Version control systems (as well as XML databases) neglect both ways in which mathematical theories are organized: (1) logic-internally, i.e. by their highly hierarchical internal structure, (2) logic-externally, i.e. by their decomposition into various files. Thus, mathematicians are facing the constantly recurring decisions whether to model mathematical concepts internally or externally. For example, one way of organizing mathematical knowledge is by considering the mathematical knowledge as a book, and splitting it in sections and paragraphs, being included in and/or having links to other "books". The organization, however, can also be logic-internal, i.e. depending on the mathematical content. We consider the external/internal distinction to be a matter of taste. By making the change management algorithms independent of this we want to give the user freedom to choose.

In this paper we propose a mathematical data structure (the *fs*-tree model) that generalizes version-controlled file systems and XML files alike. This allows us to specify and implement version control algorithms that seamlessly work across the file/file system border. For instance, we are going to provide sophisticated equality theories on top of *fs*-trees, which are essential to provide notions of consistency and invariants facilitating the propagation of effects of local changes to entire document collections [Müller, 2006; Müller and Wagner, 2007]. Our *fs*-tree model already induces a non-trivial equality theory over file systems and XML files. Elsewhere we have investigated adding additional equality theories (e.g. to arrive at *fs*-trees over OMDoc documents [Kohlhase, 2006]) to enhance versioning and to arrive at less intrusive edit-scripts.

As a proof of concept for the generalized application of our approach to a version control system, we present the ***locutor*** system [locutor, 2007]. This is a full reimplementation of the UNIX SVN client focussing on smart management of change. In addition, the ***locutor*** client identifies relations between version-controlled entries and uses them for redundancy control. In future we want to extend ***locutor*** to use semantic knowledge about the document format to distinguish semantically relevant from irrelevant changes, where SUBVERSION just compares text files line by line.

2 The Fundamental Data Structure

In this section we propose an abstract data structure for modeling repositories and working copies. We utilize a simple tree structure for the *storage layer*, and model node sharing at a *semantic layer* via explicit links.

Definition 1 (*fs*-tree)**.** Let Σ be an alphabet and D a data set. We call a tuple $T = \langle \mathrm{V}, \mathrm{E}, \tau, \nu, \lambda \rangle$ a ***fs*-tree** if and only if

(1.1) $G = (\mathrm{V}, \mathrm{E})$ is a tree with root $r \in \mathrm{V}$ and leaves $\mathrm{L} = \{v \in \mathrm{V} | \eta^-(v) = \emptyset\}$ where $\eta^-(v)$ denotes the set of outgoing edges of v.

(1.2) $\tau \colon \mathrm{L} \mapsto \{f, s\}$ is a **node typing** function, such that $\mathrm{V}_f = \{v \in \mathrm{L} | \tau(v) = f\}$, $\mathrm{V}_s = \{v \in \mathrm{L} | \tau(v) = s\}$, and $\mathrm{V}_d = \mathrm{V} \setminus \mathrm{L}$ form a partition on V.

(1.3) $\nu \colon \mathrm{V} \mapsto \Sigma$ is a **node denomination** function.

(1.4) $\lambda \colon \mathrm{V}_f \mapsto \mathrm{D}$, $\lambda \colon \mathrm{V}_s \mapsto \Sigma^*$, and $\lambda \colon \mathrm{E} \mapsto \Sigma$ is a **encoding** function, where $\forall v, w \in \mathrm{V}. w \neq w' \Rightarrow \lambda((v, w)) \neq \lambda((v, w'))$.

We denote the set of all *fs*-trees with FS.

Without loss of generality we assume that D contains all of Σ^*, $\mathbb{N}$ and is closed under n-tuple, set construction and function definition.

To traverse a *fs*-tree we define a lookup function seeking a target node relative to a source node via a tree path. A tree path is a sequence of edge labels, not to be confused with a graph path being a sequence of node labels.

Definition 2 (*fs*-tree lookup)**.** Let $T = \langle \mathrm{V}, \mathrm{E}, \tau, \lambda \rangle$ be a *fs*-tree. We define a **lookup function** $\bullet \colon \mathrm{V} \times \Sigma^* \rightharpoonup \mathrm{V}$ such that

$$v \bullet \epsilon = v \tag{2.1a}$$

$$v \bullet \pi \bullet a = \begin{cases} w & \text{if } \exists z \in \mathrm{V}. v \bullet \pi = z \wedge \\ & (z, w) \in \mathrm{E} \wedge \lambda((z, w)) = a \\ \perp & \text{otherwise} \end{cases} \tag{2.1b}$$

for all $v \in \mathrm{V}, \pi \in \Sigma^*$ and $a \in \Sigma$. We naturally extend the lookup function on *fs*-trees, such that

$$T \bullet \pi = r \bullet \pi \ \text{if } r \text{ root of } T \tag{2.2}$$

and call π a ***fs*-lookup path**, or *fs*-path, in T if $T \bullet \pi$ is defined. We denote an empty *fs*-path with ε.

Note that $\bullet$ is well-defined: If a node $w \in \mathrm{V}(T)$ is a descendant of a node $v \in \mathrm{V}(T)$ then, because $G = (\mathrm{V}, \mathrm{E})$ is a tree, there is a *fs*-path π such that $v \bullet \pi = w$. The uniqueness is directly entailed by (1.4).

To retrieve the data stored in a *fs*-tree we define a *fs*-tree value function.

Definition 3 (Value function)**.** Let $T = \langle \mathrm{V}, \mathrm{E}, \tau, \lambda \rangle$ be a *fs*-tree. A **value function** γ over T is a strict function such that

$$\gamma_T \colon \mathrm{V}_f \mapsto \mathrm{D} \colon \gamma_T(f) = \lambda(f) \tag{3.1}$$

$$\gamma_T \colon \mathrm{V}_s \mapsto \mathrm{V} \colon \gamma_T(s) = \begin{cases} T \bullet \pi & \text{if } \pi = \lambda(s) \\ & \textit{fs-path in } T \\ \perp & \text{otherwise} \end{cases} \tag{3.2}$$

$$\gamma_T \colon \mathrm{V}_d \mapsto \mathcal{P}(\Sigma) \colon$$
$$\gamma_T(d) = \{\lambda((d, w)) | (d, w) \in \mathrm{E}\} \tag{3.3}$$

for all $v \in \mathrm{V}(T)$.

To simplify matters, we usually omit indexing γ unless the context requires. To fortify our intuition about *fs*-trees we are going to investigate two applications.

Example 1 (File system). Let us assume the UNIX file system (UFS), i.e. a set of files $\mathrm{F} = \{F_1, \ldots, F_n\}$, where each F_i represents any stored object, e.g., a directory, a regular file, or a symbolic link. All these items are represented as "files" each having at least a location within the file system, a name and other attributes. This information is actually stored in the file system in an *inode* (index node), the basic file system entry. In general each inode contains the item's type (e.g. file, directory, or symbolic link), the item's size, and the location of the item's contents on the disk, if any. An inode stores everything about a file system entry except its name. The names are stored in directories and are associated with pointers to inode numbers. A symbolic link is a file that points to another file. Hence, the logical structure of a UNIX-like file system is a cyclic graph. For example, let us consider the UFS structure presented on the left side of Fig. 1. The root directory (`/`) contains three subdirectories (`/bin`, `/usr`, and `/tmp`). The `/bin` directory contains one regular file (`pwd`) and one symbolic link (`@tcsh`). The `/usr` directory contains one subdirectory (`/usr/bin`) which in turn contains one regular file (`tcsh`), the link target of `/bin/@tcsh`. Up to here the underlying data structure is an acyclic graph, however, the directory `/tmp`, in particular, the symbolic link `@r` turns the UFS structure into a cyclic graph. The subdirectory `/tmp/nrm` is empty.

The corresponding *fs*-tree T is displayed on the right side of Fig. 1. According to the respective file system type, directories are represented by gray colored nodes, regular files by transparent nodes, and symbolic links by rectangular nodes. Two nodes $v, u \in \mathrm{V}(T)$ are connected if u is in the list of content of v. Assuming an adequate alphabet Σ and data set D, respectively, file nodes are labeled by their corresponding file content, symbolic link nodes by the symbolic link target path, and edges are labeled by the names of the files they are pointing to. For convenience we left out the node labeling as for file system those are equal to the respective incoming edge label, i.e. $\nu(u) = \lambda((v, u))$ for

```
/
/bin/
/bin/pwd
/bin/@tcsh ⟶ ../usr/bin/tcsh
/usr/
/usr/bin
/usr/bin/tcsh
/tmp/
/tmp/@r ⟶ /
/tmp/nrm/
```

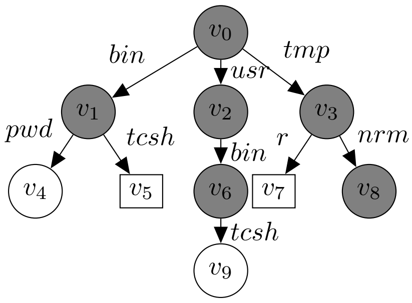

Figure 1: A *fs*-tree over a UNIX-like file system

```
<paper>
  <header>
    <title>fs-trees</title>
    <authors>
      <author>Normen Müller<author/>
      <author>Michael Kohlhase<author/>
    </authors>
  </header>
  <body><xi:include href="𝒰"/></body>
</paper>
```

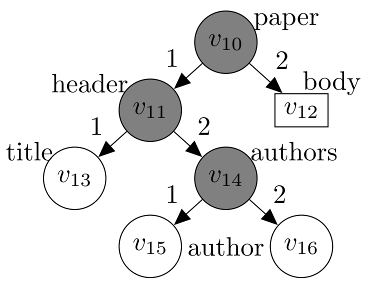

Figure 2: A *fs*-tree over an XML document

all $v \in \mathrm{V}(T)$. Note as to UFS, root nodes are special, though, i.e. the location of v_0 is encoded into a mount point.

Consequently, *fs*-trees allow us interpreting file system graphs as trees without any information loss. In the following example we demonstrate how to build *fs*-trees over semi-structured documents. The basic idea is that an XML file is an ordered tree and that redundancy can be avoided by sharing fragments through XINCLUDE references. We will use a straightforward analogy between XML files and UNIX file systems in our approach. In particular, a version control system based on *fs*-trees would blow up a file system with XML files into a big tree where files themselves are expanded according to their structure so that each unstructured textual content can be versioned individually.

Example 2 (XML documents). The left side of Fig. 2 presents the structure of this paper in an imagined XML format. The `header` element specifies the cover page. Constituent parts are the title and the authors. The former one is represented by a `title` element and the latter one by an `authors` element, where the individual authors are marked up by an explicit `author` element. The actual text is specified by a `body` element and included via an XINCLUDE directive[1].

Note that in contrast to file systems, XML documents are ordered trees therefore we encode the *child_of* relation using rational numbers annotated to the respective tree edges. This always enables us to insert new tree nodes. For example, to add a third author between the first and the second one in the *fs*-tree, we would simply add a node v_{17} where $\lambda((v_{14}, v_{17})) = (\lambda((v_{14}, v_{15})) + \lambda((v_{14}, v_{15})))/2 = 3/2$. Similar to file system mappings, empty XML elements are *fs*-tree leaves, where links like XINCLUDE or other kind of XML reference statements are of *fs*-tree type s, and the remaining empty XML elements are of type f. Content, represented in XML text nodes, is treated like files and URIs in reference statements like symbolic links on file systems.

[1]XINCLUDE is a standardized XML vocabulary for specifying document inclusions.

Note, up to the `xi:include` element, the XML document does not contain any XML attributes. Encoding of XML attributes in *fs*-trees is modeled in Section 3.

We conclude this section with the observation that with this model we have obliterated the border between file systems and files as both can be mapped to *one* *fs*-tree. For example, let us consider to store the XML file represented at Fig. 2 in the `/tmp/nrm/` directory. This would extend the *fs*-tree in Fig. 1 at node v_9 with the *fs*-tree rooted at v_{10}. Consequently, with *fs*-trees we have the freedom to structurally decompose semi-structured files over file systems, but holding up the dependency graph. This generalization of XML documents promotes versioning as well as authoring processes. That is, regarding authoring process self-contained XML documents are appreciated to, for instance, realize consistent text replacements, but regarding versioning small decomposed XML documents are preferred to manage small-sized revision chunks.

Some of our ideas have already discussed in the context of file systems, e.g. REISER4 [reiser4, 2008], which proposes to blur the traditional distinction between files and directories introducing the concept of a "resource". For example, a resource named `kwarc.mp3` can be accessed as `./kwarc.mp3` to obtain the compressed audio and as `./kwarc.mp3/` for a "directory" of metata files. Our approach can model this behavior: we would interpret resource `kwarc.mp3` as a structured XML file that contains both the compressed audio in a CDATA section and the metadata in custom XML markup. All aspects of the resource can be addressed by standard XPATH queries, e.g. fine-grained access to document fragments, for example the title of the third song can be obtained with the query `./kwarc.mp3/metadata/song[position()=3]/@title`.

3 Version-Controlled *fs*-trees

In the following we model the core notions of version control systems, in order to map versioning workflows to *fs*-trees. In general, a version control system is a special file server, designed for concurrent editing and to store history information. A normal file server (e.g. NFS) can provide file sharing, but would keep only one version of each file (the most recent one). The core of a version control system, a *repository*, is a centralized store for data. It stores data in form of a filesystem tree, provides read/write access to the stored data[2], and remembers any modification made to it. A *working copy* is made up of two parts. A local copy of the directory tree of a project and an administrative directory (e.g. `.svn`) in each directory, storing version control information. Users edit their working copy locally and commit their changes to the repository. After a commit, all other users can access the changes by updating their working copies to the latest revision. For each file the respective administrative directory stores the working revision of the file and a time stamp of the last update. Based on this information, version control systems can determine the state of the file. The state of the repository after each commit is called a *revision.* To each revision, an identifier is assigned which identifies the revision uniquely. Without loss of generality, we follow the SUBVERSION model here, which uses natural numbers as identifiers and assigns revisions to the whole tree. A certain file can be left unchanged through different revisions, i.e. files in a working copy might have different revisions, but files in the repository always have the same revision.

Definition 4 (Repository)**.** A **repository** is a function $\varrho\colon \mathbb{N} \mapsto \mathrm{FS}$. We denote the set of all repositories with R.

To define a working copy, we need to capture the intuition that every (versioned) *fs*-tree node comes from a repository.

Definition 5 (Repository mapping)**.** Let T be a *fs*-tree. A **repository mapping** is a function $\omega\colon \mathrm{V}(T) \rightharpoonup \mathrm{R} \times \Sigma^* \times \mathbb{N}$ and we define the operation

$$\mu^\omega(v) := \begin{cases} \varrho(r).\pi & \text{if } r \in dom(\varrho) \text{ and} \\ & \omega(v) = \langle \varrho, \pi, r\rangle \\ \bot & \text{otherwise} \end{cases} \tag{5.1}$$

that computes the **corresponding** repository node for $v \in \mathrm{V}(T)$. Note that $\mu^\omega \in \mathrm{V}(\omega_1(v)(\omega_3(v)))$. For convenience we write $\omega(v) = \varrho/\pi@r$ instead of $\omega(v) = \langle \varrho, \pi, r\rangle$.

The correspondence μ^ω is sufficient for versioning: For a versioned node v in a *fs*-tree T, there has to be a repository $\omega_1(v)$ with a *fs*-tree at revision $\omega_3(v)$ such that $\omega_2(v)$ is a valid *fs*-path in $\omega_1(v)(\omega_3(v))$ to the corresponding node of v.

Definition 6. We call a node $v \in \mathrm{V}(T)$ **version-controlled**, or versioned, with respect to ω *iff* $v \in dom(\mu^\omega)$ and a *fs*-tree **uniform** *iff* there exists a $\varrho \in \mathrm{R}$ and a $r \in \mathbb{N}$ such that $\omega_1(v) = \varrho$ and $\omega_3(v) = r$ for all $v \in \mathrm{V}(T)$.

[2]We neglect modeling of access control lists (ACL) for convenience.

For working copy directories we also want to enforce that all the contents of the directories are actually checked out. This ensures that all inode pointers in a directory file are valid (cf. Example 1).

Definition 7 (Repository complete)**.** Let $T = \langle \mathrm{V}, \mathrm{E}, \tau, \nu, \lambda\rangle$ be a *fs*-tree. We call a versioned node $v \in \mathrm{V}_d(T)$ with $\omega(v) = \varrho/\pi@r$ **repository complete** *iff* for all $a \in \Sigma$ with $e' := (\mu^\omega(v), w') \in \mathrm{E}(\varrho(r))$ and $\lambda(e') = a$ there exists a $w \in \mathrm{V}(T)$ such that $\mu^\omega(w) = w'$, $e := (v, w) \in \mathrm{E}(T)$ and $\lambda(e) = a$.

The next definition formalizes the previously described notion of a working copy as a local reflection of the respective repository directory tree.

Definition 8 (Working Copy)**.** Let $T = \langle \mathrm{V}, \mathrm{E}, \tau, \nu, \lambda\rangle$ be a *fs*-tree. We call a tuple $\langle S, \omega, \Delta\rangle$ a **working copy** *iff*

(8.1) S is a subtree of T ($S \sqsubseteq T$)

(8.2) If $v \in \mathrm{V}_f(S)$ then $\gamma(v) = \Delta(v, \gamma(\mu^\omega(v)))$.

(8.3) If $v \in \mathrm{V}_s(S)$ then $\lambda(v) = \Delta(v, \lambda(\mu^\omega(v)))$.

(8.4) If $v \in \mathrm{V}_d(S)$ then v is repository complete and for all $a \in \gamma_S(v)$ we have $a \in \gamma_{S'}(\mu^\omega(v))$ and $\gamma_S(v.a) = \Delta(v.a, \gamma_{S'}(\mu^\omega(v.a)))$ where $S' = \omega_2(v)(\omega_3(v))$.

(8.5) $\Delta\colon \mathrm{V}_f \times \mathrm{D} \mapsto \mathrm{D}$, $\Delta\colon \mathrm{V}_s \times \Sigma^* \mapsto \Sigma^*$, and $\Delta\colon \mathrm{V}_d \times \mathcal{P}(\Sigma) \mapsto \mathcal{P}(\Sigma)$.

(8.6) for all $v \in \mathrm{V}(S) \cap dom(\mu^\omega)$ we have $\mu^\omega(v) = \omega_1(v)(\omega_3(v)).\omega_2(v)$.

Equation (8.1) specifies a locality condition for working copies. That is, working copies are "local" in sense of accessible within the current *fs*-tree, e.g. the current file system. Equations (8.2) – (8.4) ensure for each node the value is equal to the value of the corresponding node modulo any local modifications. The *difference function* Δ may be considered as a parameterized collection of transformation functions $\delta_v\colon \mathrm{D} \mapsto \mathrm{D}$ for each node $v \in \mathrm{V}(T)$, such that $\Delta(v, \gamma(\mu^\omega(v))) = \delta_v(\mu^\omega(v)) = \gamma(v)$. For example, think of SUBVERSION: After a fresh working copy synchronization we have $\delta_v = id$ for all $v \in V$. Equation (8.6) ensures that every versioned working copy node corresponds to the correct repository node.

Lemma 9. *Let T be a fs-tree. If $\langle S, \omega, \Delta\rangle$ is a working copy in T, then $\langle S|_\pi, \omega, \Delta\rangle$ is a working copy in T for all fs-paths π in S.*

Proof. Given that $S|_\pi \sqsubseteq S \sqsubseteq T$ and (8.2) – (8.6) are universal, all conditions are inherited by $S|_\pi$, in particular, repository completeness. Consequently $\langle S|_\pi, \omega, \Delta\rangle$ is a working copy in T. □

Note that it is not all subtrees of working copies are working copies themselves since they may run afoul of directory completeness.

Example 3 (Working Copy). Let us consider the working copy structure depicted in Fig. 3. The *fs*-tree T constitutes a local file system tree and the subtree $S \sqsubseteq T$ a working copy $\langle S, \omega, \Delta\rangle$ in T at ϕ. Here we have $\omega_1(v) = \varrho$, $\omega_2(v) = \pi/\psi$ where $\psi \in \Sigma^*$ such that $S.\psi = v$ and $\mu^\omega(v) = \varrho(r).\pi.\psi$, and $\omega_3(v) = r$ for all $v \in \mathrm{V}(S)$. Hence $\langle S, \omega, \Delta\rangle$ is a working copy. Note, working copies are not uniformly versioned. Hence,

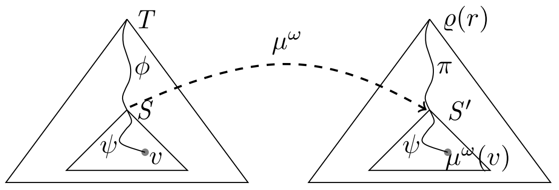

Figure 3: A working copy for $\varrho/\pi@r$ in T at ϕ.

switching repositories and/ or revisions within a versioned *fs*-tree is legal, however, invalid path labelings are restricted due to repository completeness and equation (1.4).

We are now in a position to understand the how to map common versioning commands like `add`, `update` , `remove`, and `commit` to *fs*-trees: Recall that for a file system entry f a SUBVERSION `add` command adds repository mapping information for the entry to the respective administrative directory. An `update` command on a versioned file f retrieves from the repository the difference δ^k_f between the working copy revision of f and the specified revision k and merges it into f. A `remove` command deletes the repository mapping information from the administrative directory. Finally, a `commit` command communicates the differences between the repository revision of f (cached in the administrative directory) and the potentially modified working copy entry to the repository, where it is incorporated into the data store and updates the meta-data in the administrative directory (for details about these commands see [Collins-Sussman *et al.*, 2004]).

Definition 10 (Version Commands on *fs*-trees)**.** Let $\langle S, \omega, \Delta\rangle$ be a working copy and $v \in V(S)$. The version control command "`add` v" extends the repository mapping ω by a new pair $(v,\ \omega_1(v)/\omega_2(v).a@\omega_3(v))$, where a is the name of v.

An "`update -r` k" changes $\omega_3(v)$ to k, and merges δ^k_f into $\Delta(v)$ so that (8.6) is maintained.

A "`remove` v" command deletes the pair (v, o) from ω.

A "`commit` v" command communicates $\Delta(v)$ to the repository $\omega_1(v)$ which extends the repository $\varrho' :=\omega_1(v)$ with a new pair (k, S'), where $k = \max(dom(\varrho))$ and S' is derived from $\varrho(k-1)$ by replacing $w :=\varrho(k-1).\pi$ with $\Delta(v, w)$.

The remaining versioning commands `copy`, or `switch` are modeled accordingly.

4 Version Control with Properties and Externals

The ability to mix various working copies with respect to corresponding repositories and/ or revisions gives us the notion of **nested working copies**. That is a working copy n inside another working copy w, but the respective *fs*-tree edge connecting the root node u of n to a leave node v of w is not versioned in the corresponding repository $\varrho(r)$ of w, i.e. $(\mu^{\omega}(v), \mu^{\omega}(u)) \notin \mathrm{E}(\varrho(r))$.

However, as mentioned at the beginning, sometimes it is useful to construct working copies made out of a number of different subdirectories to come from different locations in a repository, or from different repositories altogether. One could set up such a structure of nested working copies by hand, but if this structure is important for everyone else using the respective repository, every other user would need to perform the same manual setup.

To support those structures of working copies without the need for manual reconstruction, we are going to formalize the well-known construct of external definitions (*aka.* externals). Basically, an external definition is a property defined on a working copy, mapping a directory to a URL and a particular revision of another versioned directory. Hence, externals themselves are nested working copies, but with the decided advantage of being versioned themselves. That is, once *defined* on a working copy, all other users benefit by updating their working copies to the latest revision. When we refer to a "nested working copy" we mean nested working copies defined by external definitions.

To capture the notion of defined nested working copies, we first extend *fs*-trees by properties to eventually adapt the definition of a repository and working copy, respectively, through to external definitions.

Definition 11 (*fsp*-tree)**.** Let P be an arbitrary set. A ***fsp*-tree** is a *fs*-tree T together with a property function $\beta \colon \mathrm{V(T)} \times \mathrm{P} \rightharpoonup \mathrm{D}$. We call elements of P **property names** and denote the set of all *fsp*-trees with FSP.

Property functions, or "properties" for short, are custom meta-data attached to an *fs*-tree. Note that we cannot map *fsp*-trees directly to file system trees. Version control systems solve this situation by administrative directories, though, we assume an equivalent file system structure. Regarding XML documents, the mapping is straightforward: properties become XML attributes in the obvious way.

Most version control systems provide a distinguished property for external definitions, we will assume the existence of a property name $\mathsf{ext} \in \mathrm{P}$ with a specific signature: a symbol is associated with a repository entry at a specific (remote) path and revision.

Definition 12 (External Definition)**.** Let T be an *fsp*-tree, and $v \in \mathrm{V}(T)$. An **external definition** is a property function $\xi(v) := \beta(v, \mathsf{ext})$, such that $\xi(v) \colon \Sigma^+ \rightharpoonup \mathrm{R} \times \Sigma^* \times \mathbb{N}$. We call elements of $\xi(v)$ **externals**.

We consider an external $\langle \pi.d, \varrho/\rho@r\rangle \in \xi(v)$ to be **well-defined** *iff* $\varrho(r).\rho$ is defined, otherwise **dangling**.

An external $\langle \pi.d, \varrho/\rho@r\rangle \in \xi(v)$ is **admissible** *iff* well-defined and there is a $u \in \mathrm{V}(T)$ such that $u = v.\pi$ and $u.d = \bot$. We call $\xi(v)$ admissible *iff* all $e \in \xi(v)$ are admissible.

The well-definedness criteria ensures that the repository referring to actually exists and admissible specifies $\pi.d$ is relative to v and the symbol d has not been versioned yet.

External definitions are not local to a working copy, but global in sense of all working copies of the respective repository obtain these properties. Thus external definitions have to be stored in repositories. Note, in case of SUBVERSION's external definitions, they extend the working copy they are defined on. That is, the underlying *fs*-tree of a working copy is extended by the underlying *fs*-tree of the externally defined working copy. In the following we naturally extend repositories and working copies to externals.

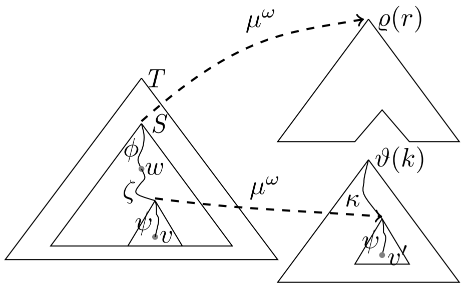

Figure 4: A working copy for $\varrho/\varepsilon@r$ in T with an external for $\vartheta/\kappa@k$.

Definition 13 (Repository with Externals). A **repository with externals** is a repository whose externals are admissible.

Definition 14 (Working Copy with Externals). Let $T = \langle \mathrm{V}, \mathrm{E}, \tau, \nu, \lambda, \beta \rangle$ be a *fsp*-tree. We call a working copy $\langle S, \omega, \Delta \rangle$ a **working copy with externals** if and only if

(14.1) $\beta(v, k) = \Delta(v, \beta(\mu^\omega(v), k))$ for all $v \in \mathrm{V}(S)$ and $k \in \mathrm{P}$.

(14.2) for all $v = S{\bullet}\varphi$ with $\langle \zeta, \Theta \rangle \in \xi(v)$ we have $\omega(S{\bullet}\varphi{\bullet}\zeta) = \Theta$.

Similar to (8.2) – (8.4), equation (14.1) ensures for each versioned node the property value is equal to the property value of the corresponding node modulo any local modifications with respect to property names. This restriction does not refer solely to external definitions, but to properties in general. In the following we will always assume that repositories and working copies, respectively, have externals without further mention.

Example 4 (Working Copy with External). Let us consider the working copy with externals depicted in Fig. 4. The *fsp*-tree T constitutes a local file system tree and the subtree $S \sqsubseteq T$ a working copy $\langle S, \omega, \Delta \rangle$. The node node $w = S{\bullet}\phi$ has an externals definition with $\xi(w) = \langle \zeta, \vartheta/\kappa@k \rangle$. Here we have $\omega_1(v) = \vartheta$, $\omega_2(v) = \kappa/\psi$, and $\omega_3(v) = k$ where $\vartheta(k){\bullet}\kappa{\bullet}\psi = \mu^\omega(v)$ and $v = S{\bullet}\phi{\bullet}\zeta{\bullet}\psi$. Hence $\langle S, \omega, \Delta \rangle$ is a working copy with externals.

Thanks to version control systems, external definitions are stored in administrative directories as well and therefore we do not have to modify the previously introduced mapping between file systems (with externals) and *fsp*-trees.

In case of XML documents, however, we have to perform some adaptations: for each XML element induced by a *fsp*-tree node, we have to identify the corresponding node in the respective repository to guarantee a fine-granular version control for XML fragments. To encode the information about the components of ω we propose to extend the elements in the respective XML format by three attributes:

- the `fsp:rep` attribute[3] specifies the repository root URL
- the value of the `fsp:path` attribute is the respective remote path, and
- the revision is stored in the `fsp:rev` attribute.

The XML attributes `fsp:rep` and `fsp:rev` are inherited by XML child elements, but may be individually overwritten. This enables authors to mix XML fragments within one XML document from various repositories and revisions. In this case XML fragments act like nested working copies. The `fsp:path` attribute has to be annotated to the XML root element and is implicitly defined for all descendants (relative to the XML root). XML elements defined by external definitions are specified by `fsp:ext` attributes in the respective XML parent element. The value of a `fsp:ext` attribute is a semicolon separated list of tuples. The first component represents the XPath position of the "external" XML child relative to the XML parent. The second component specifies the fully qualified repository URL and the respective revision. For example, Fig. 5 depicts our example XML document enriched with versioning meta-data. In the XML root element `paper` the repository root URL, the remote path, and the revision are presented. The `body` element is extended by an external definition. In contrast to XInclude statements externals are copied into the XML document. The `fsp:ext` attribute in the `body` element assures the identification of the `example` element to be an external. We assume, without loss of generality, that XML elements externally linked do not corrupt XML documents with respect to XML validity.

```
<paper fsp:rep="https://kwarc.info/repos/locutor"
       fsp:path="/doc/mkm08" fsp:rev="@HEAD">
  <header><title>fs-trees</title>...</header>
  <body
   fsp:ext="2 https://kwarc.info/repos/MiKo/impl.xml#ex1@512">
    <xi:include href="𝒰"/>
    <example xml:id="ex1"> ... </example>
  </body>
</paper>
```

Figure 5: An XML document with externals.

We are now in a position to implement the version control commands from Definition 10 to XML files to achieve the seamless operation of version control across the file/file system border.

Definition 15 (Version Control on XML files). Let X be a versioned XML file that admits the `fsp` attributes and e an element in X. Then the version control command "`add` e" annotates e by `fsp` attribute `fsp:rev` `="k"`, where k is current revision. Note that the `fsp` `:rep` and `fsp:path` attributes are inherited from the parent. Dually, a `remove` command just removes the `fsp` attributes.

A "`commit`" command computes the XML-differences δ_e (see [mdpm, 2008] for a XML-aware differencing algorithm) between e and and the cached repository copy e'[4] communicates them to the repository in `fsp:rep`, which creates a new repository revision by adjusting the `fsp` attributes in X' and its original in the repository.

[3]As most XML applications allow foreign namespaces, we place the new attributes in our own name space `fsp`, which we map to `http://kwarc.info/projects/locutor`.

[4]For this, we assume that the underlying version control stores a cached repository copy X' of X in the administrative directory. From this, we can take the element e' that is the cached repository copy for e.

~/stex/	https://kwarc.info/repos/stex	root@512
~/stex/LaTeXML/	https://mathweb.org/repos/LaTeXML/trunk	ext@815
~/stex/stex/		local
~/stex/stex/lib/	https://kwarc.info/repos/stexc/slides/lib	ext@4711
~/stex/stex/omdoc/	https://kwarc.info/repos/stexc/slides/omdoc	ext@4711
~/stexc/	https://kwarc.info/repos/stexc	root@4711
~/stexc/slides/		local
~/stexc/slides/lib/		local
~/stexc/slides/omdoc/		local
~/stexc/stex/	https://kwarc.info/repos/stex	ext@512
~/stexc/stex/LaTeXML/	https://mathweb.org/repos/LaTeXML/trunk	ext@815
~/stexc/stex/stex/lib/	https://kwarc.info/repos/stexc/slides/lib	ext@4711
~/stexc/stex/stex/omdoc/	https://kwarc.info/repos/stexc/slides/omdoc	ext@4711
~/stexc/www/	https://kwarc.info/repos/www	root@16676

Figure 6: Instance Structure of Working Copies

An "`update -r` k" changes `fsp:rev` to k, and merges[5] δ_f^k into e. The property versioning commands like `propset`, `propdel` are modeled by the obvious `fsp` attribute movements.

5 Redundancy Resolution

Version control systems allow us to manage files and directories that change over time. This allows authors to access older versions of files and examine the history of how and when data changed. Unfortunately, all features are restricted to individual working copies at a time, so to keep several working copies in sync with central repositories, authors have to update each one by its own.

For example let us consider the structure of working copies depicted in Fig. 6. The first column represents the local working copy directories, the second one the respective fully qualified remote working copy URL, and the last one describes the checkout type and revision. If a directory is the root of a working copy checkout the type is `root`, if a directory is an external definition it is `ext`, otherwise `local`. The revision is suffixed to the type separated by an `@` character. For instance, the directory `~/stex` constitutes the root directory of a working copy checked out in revision 512 from the SUBVERSION repository at `https://kwarc.info/repos/stex`. The subdirectories `/LaTeXML`, `/stex/lib`, and `/stex/omdoc` are external definitions. The directory `~/stex/stex` is local. Note, as to Definition 14 subdirectory `/www`, is not an external definition but constitutes the root directory of a working copy manually checked out in revision 16676 from the SUBVERSION repository at `https://kwarc.info/repos/www`. In the following we call the first working copy root sTEX and the second one sTEX$\mathcal{C}$. Apparently, in order to keep sTEX as well as sTEX$\mathcal{C}$ in sync with the central repositories, we have to update both by invoking the respective version control `update` command in each root directory. A simple approach to solve this problem could be a shell script updating all working copies within the respective directory. Redundant externals (`LaTeXML`, `stex/lib`, `stex/omdoc`, and `stex` in our example), are updated more than once, which wastes time, bandwidth, and space. Moreover, we also have to perform local commits to propagate changes. Otherwise, for example, changes to sTEX are not immediately available in `~/stexc/stex`, a constant source of errors and confusion in practice. Managing a structure of related working copies is a complex task, and automating this would foster logical separations of multiple repositories via external definitions.

Definition 16 (Redundancy)**.** Let T be a *fsp*-tree, $w = \langle S, \omega, \Delta \rangle$ a working copy in T and $v \in \mathrm{V}(T)$. We call v **redundant** to S ($v \prec S$) *iff* there is a $w \in \mathrm{V}(S)$ such that $\omega(w) = \omega(v)$. We call a working copy $m = \langle S', \omega, \Delta \rangle$ redundant to w ($m \prec w$) *iff* $v \prec S$ for all $v \in \mathrm{V}(S')$.

Lemma 17. *Let T be a fsp-tree and $w = \langle S, \omega, \Delta \rangle$ a working copy in T.*

(i) We have $S \prec v$ for all $v \in \mathrm{V}(S)$.

(ii) If $m = \langle S', \omega, \Delta \rangle$ is a working copy, S, S' are uniform, and r is root of S' with $r \prec S$ then we have $m \prec w$.

Proof. Choose $w = v$ for (17.i) and (17.ii) is a consequence of Lemma 9. □

Back to our example in Fig. 6, we can now identify redundant working copies: the external definition `stex` on `~/stexc/` is redundant to `~/stex` due to equivalent repository root URL, remote path and revision. Note that this task is less trivial than it seems at first glance. For instance, replacing redundant entries with symbolic forced us to introduce *update scopes* to avoid infinite loops inside SVN commands. Indeed the example in Fig. 6 already shows such a dangerous loop.

In the *`locutor`* system we have implemented redundancy identification as well as resolution on file system level via an XML registry. That is, redundant externals are transformed to symbolic file system links (`transex`) during an `update` or `checkout` versioning command. To emphasize *raison d'être* of transformed externals we performed the following case study: one

[5]Note that we are not specifying here how the merge actually works and appeal to the intuition of the reader. In fact, we are currently working on an extension of XML merge algorithms to include format equality information and *fs*-tree property attributes. Note that given suitable merging algorithms, we can execute the merge on the client, since SUBVERSION (and thus *`locutor`*) caches base revisions in a private part of the working copy.

of the authors has mirrored all of his local working copies in two directories: `~/svn` and `~/locutor`. The command `du -sh ~/svn` returned 6.8G and `du -sh ~/locutor`, however, returned only 3.7G after redundancy resolution.

6 Conclusion & Outlook

We have presented an abstract theory of collaborative content management and version control for collections of semi-structured documents. In particular, we have defined version control algorithms on our *fs*-tree model and extended version control to arbitrary XML formats that allow foreign-namespace attributes thus extended them to seamlessly work across the file/-file system border. Furthermore, we have shown how to control redundancy and confusion induced by duplicate links and externals.

Currently we are developing the *mdpm* system, a collection of model-based `diff`, `patch`, `merge` algorithms. These are document format dependent algorithm, comprising respective notions of *dependency*, *dependency* and *change types*, and *propagation* based on corresponding equality theories. The *mdpm* components will identify INFOMs for fine-granular change identification, compute less intrusive edit-scripts, and compute "long-range effects" of changes based on the *fsp*-trees. The currently most finished `mdiff` component is for LaTeX and XML. In the former case various LaTeX coding styles are identified to be equal, e.g. whitespace, line breaks and empty lines. The latter one is an improvement of [Radzevich, 2006].

As the transformation of externals may also be performed offline, we think of another sub-command like `transex`. This sub-command would analyze all externals of local working copies and if applicable transform these to symbolic links.

Acknowledgements Our special thanks goes to the KWARC research group for all their assistance, especially for providing us with their huge and complex SUBVERSION repositories utilizing virtually all possible — at least popular — versioning features.

References

[Allen *et al.*, 2002] Stuart Allen, Mark Bickford, Robert Constable, Richard Eaton, Christoph Kreitz, and Lori Lorigo. FDL: A prototype formal digital library – description and draft reference manual. Technical report, Computer Science, Cornell, 2002. `http://www.cs.cornell.edu/Info/Projects/NuPrl/html/FDLProject/02cucs-fdl.pdf`.

[Asperti *et al.*, 2001] Andrea Asperti, Luca Padovani, Claudio Sacerdoti Coen, and Irene Schena. HELM and the semantic math-web. In Richard. J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics: TPHOLs'01*, volume 2152 of *LNCS*, pages 59–74. Springer Verlag, 2001.

[Ausbrooks *et al.*, 2008] Ron Ausbrooks, Bert Bos, Olga Caprotti, David Carlisle, Giorgi Chavchanidze, Ananth Coorg, Stphane Dalmas, Stan Devitt, Sam Dooley, Margaret Hinchcliffe, Patrick Ion, Michael Kohlhase, Azzeddine Lazrek, Dennis Leas, Paul Libbrecht, Manolis Mavrikis, Bruce Miller, Robert Miner, Murray Sargent, Kyle Siegrist, Neil Soiffer, Stephen Watt, and Mohamed Zergaoui. Mathematical Markup Language (MathML) version 3.0. W3C working draft of march april 9., World Wide Web Consortium, 2008. Available at `http://www.w3.org/TR/MathML3`.

[Baraniuk *et al.*, 2002] R.G. Baraniuk, C.S. Burrus, B.M. Hendricks, G.L. Henry, A.O. Hero III, D.H. Johnson, D.L. Jones, J. Kusuma, R.D. Nowak, J.E. Odegard, L.C. Potter, K. Ramchandran, R.J. Reedstrom, P. Schniter, I.W. Selesnick, D.B. Williams, and W.L. Wilson. Connexions: DSP education for a networked world. In *Acoustics, Speech, and Signal Processing, 2002. Proceedings. (ICASSP '02). IEEE International Conference on*, volume 4 of *ICASSP Conference Proceedings*, pages 4144–4147. IEEE, 2002.

[Buswell *et al.*, 2004] Stephen Buswell, Olga Caprotti, David P. Carlisle, Michael C. Dewar, Marc Gaetano, and Michael Kohlhase. The Open Math standard, version 2.0. Technical report, The Open Math Society, 2004. `http://www.openmath.org/standard/om20`.

[CNX, 2007] CONNEXIONS. Project homepage at `http://www.cnx.org`, seen February 2007.

[Collins-Sussman *et al.*, 2004] Ben Collins-Sussman, Brian W. Fitzpatrick, and Michael Pilato. *Version Control With Subversion.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004. online version available at `http://svnbook.red-bean.com/nightly/en/svn-book.html`.

[DARCS, 2008] darcs, seen January 2008. available at `http://darcs.net/`.

[Fogel and Bar, 2003] Karl Franz Fogel and Moshe Bar. *Open Source Development with CVS.* Paraglyph Press, 2003.

[Franke and Kohlhase, 2000] Andreas Franke and Michael Kohlhase. System description: MBASE, an open mathematical knowledge base. In David McAllester, editor, *Automated Deduction – CADE-17*, number 1831 in LNAI, pages 455–459. Springer Verlag, 2000.

[GIT, 2007] Git - Fast Version Control System, seen September 2007. available at `http://git.or.cz/`.

[Kohlhase, 2006] Michael Kohlhase. OMDOC – *An open markup format for mathematical documents [Version 1.2].* Number 4180 in LNAI. Springer Verlag, 2006.

[locutor, 2007] *locutor*: An Ontology-Driven Management of Change, seen June 2007. system homepage at `http://www.kwarc.info/projects/locutor/`.

[mdpm, 2008] *mdpm*: A Collection of Model-based DIFF, PATCH, MERGE Algorithms, seen March 2008. system homepage at `http://www.kwarc.info/projects/mdpm/`.

[Müller and Wagner, 2007] Normen Müller and Marc Wagner. Towards Improving Interactive Mathematical Authoring by Ontology-driven Management of Change. In Alexander Hinneburg, editor, *Wissens- und Erfahrungsmanagement LWA (Lernen, Wissensentdeckung und Adaptivität) conference proceedings*, pages 289–295, 2007.

[Müller, 2006] Normen Müller. An Ontology-Driven Management of Change. In *Wissens- und Erfahrungsmanagement LWA (Lernen, Wissensentdeckung und Adaptivität) conference proceedings*, pages 186–193, 2006.

[Radzevich, 2006] Svetlana Radzevich. Semantic-based Diff, Patch and Merge for XML-Documents. Master's thesis, Universität des Saarlandes - Saarbrücken, 2006.

[reiser4, 2008] Reiser4. `http://en.wikipedia.org/wiki/Reiser4`, seen July 2008.