INFORMATION SOCIETY TECHNOLOGIES
(IST)
PROGRAMME

Project IST-2001-33562 MoWGLI

Report n. D5.a
Distributed Knowledge Management and Version Control

Main Authors:
Serge Autexier, Frederick Eberhardt, Dieter Hutter, Michael Kohlhase, Romeo Anghelache

Project Acronym: MoWGLI
Project full title: Mathematics On the Web: Get it by Logic and Interfaces
Proposal/Contract no.: IST-2001-33562 MoWGLI

Abstract

We propose an infrastructure for collaborative content management and version control for structured mathematical knowledge. This will enable multiple users to work jointly on mathematical theories with minimal interference.

We describe the API and the functionality needed to realize a CVS-like version control and distribution model. This architecture extends the CVS architecture in two ways, motivated by the specific needs of distributed management of structured mathematical knowledge on the Internet. On the one hand the one-level client/server model of CVS is generalized to a multi-level graph of client/server relations, and on the other hand the underlying change-detection tools take the math-specific structure of the data into account.

Contents

1 Overview

The MoWGLI project is concerned with representation formalisms for mathematical knowledge, such as MathML [8], OpenMath [7] or OMDoc [16], mathematical content management systems [13, 1, 2], as well as publication and education systems for mathematics. The perceived interest in the domain of general knowledge management tools applied to mathematics is that mathematics is a very well-structured and well-conceptualized subject. The main focus of the mathematical knowledge management (MKM) techniques is to recover the content/semantics of mathematical knowledge and exploit it for the application of automated knowledge management techniques, with an emphasis on web-based and distributed access to the knowledge.

In this report, we extend the focus of MKM techniques from the distributed *access* to mathematical knowledge to the management and *creation process* of mathematical knowledge, which is — for the most part — a distributed and collaborative process. After all, even if mathematicians often develop individual contributions alone (e.g. in single-authored papers), the progress of a mathematical theory or sub-field involves a multitude of authors — communicating via meetings, messages and publications. Moreover, in contrast to the "knowledge access" scenario, where the mathematics is relatively static, the "knowledge creation" scenario involves managing the change of resources. We claim that MKM techniques have the potential of supporting this scenario as well, and that the "knowledge creation/management" scenario is potentially even more important for applications, as knowledge can only be accessed after it has been created.

The main claim of this report is that distribution of mathematical knowledge bases on the web cannot be solved without understanding the dynamics of the collaborative creation/-management problem for mathematical knowledge, and that a variation of the collaborative version control CVS system [12] already provides a good paragigm for building a distributed mathematical knowledge management system out of the heterogeneous MKM systems we have today.

In the long run, we expect the implementations of techniques like the ones presented in this report, to play a similarly facilitating role in the development of open repositories of formal mathematical knowledge as the code management systems like the CVS system have had for the creation of the wealth of open-source software we know today.

We will discuss the overall architecture of a distributed collaborative management system for mathematical knowledge (section 2), the basics of the distribution and version control protocol for interaction between the MKM systems (section 4), fundamentals of the differencing and merging algorithms needed for this architecture (section 3). Finally, we will discuss ways of extending existing MKM systems in the MoWGLI environment to components of the envisioned distributed system for collaborative managment of mathematical knowledge.

1.1 MKM, Distribution and Collaboration

Currently, MKM systems either support simple monotonic addition of mathematical content or are specialized to particular applications, e.g. the Maya system [5] which is specialized to formal software engineering and verification. The "development graph" model for a management of theory change [14] employed in this system uses a rich set of relations among theories to trace logical dependencies among mathematical objects and propagate/limit the effects of changes to the theories.

The MBase [13, 17] system is currently a member of the first class, but it can communicate with the Maya system via the joint interface language OMDoc [16]. As an effect, MBase/Maya support theory management on the fragment of OMDoc that corresponds to the Maya development graph. In fact, in [17] we have proposed to distinguish two kinds of MBases, different in their data changing policies.

- An archive MBase, which is epitomized by the Journal MBase MJ in our scenario below, archives unchanging mathematical knowledge and is referenced by many other MBases.
- A scratch-pad MBase like the personal MBases MR and MR', that do not have any dependents and are primarily used for theory development.

To get a feeling of the requirements for the functionality addressed in this paper, let us take a look at a likely research communication scenario: We will first describe the communication pattern in a neutral way — say as it could have happened in the era of mathematics done with pen and paper (around 2001), and then model it using distributed MKM (about 2010).

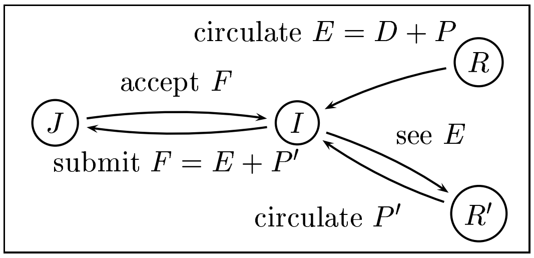


Figure 1: Classical Research Cooperation

classical, see Figure 1 Researcher R works on theory T together with his colleague R' at institute I. The theory T is a body of mathematics laid down in an article A published in journal J. Now, R extends theory T by a new definition D (say for a mathematical object O), proves a set P of theorems about O, and calls the resulting extended theory E. After that, R tells her colleague R' at I about D and P (say by circulating a memo in I), who gets interested and proves a set P' of useful properties of O. Together, R and R' put the theory E into final form F, and submit it to journal J. This accepts F and publishes it.

with MKM, see Figure 2 In 2010, the publisher of journal J has established an MBase server MJ for J which now contains theory T. Furthermore, the institute has its own departmental MBase MI and the researchers R and R' have the personal MBases MR and MR'. Now R develops the formalization FD of O, stores it in MR and formalizes the set P of theorems by formalizing them and formally proving them[1] (yielding FP in MR). Instead of sending around an internal note about D and P in I, R moves their formalizations FD and FP into the institute MBase server MI, from where R' can

[1]To do so, R may need to revise the initial version of D several times in order to be able to prove the desired theorems (reproving the already obtained results that depended on a previous version of D every time). This process is supported by MBase/Maya based on techniques presented in [5]

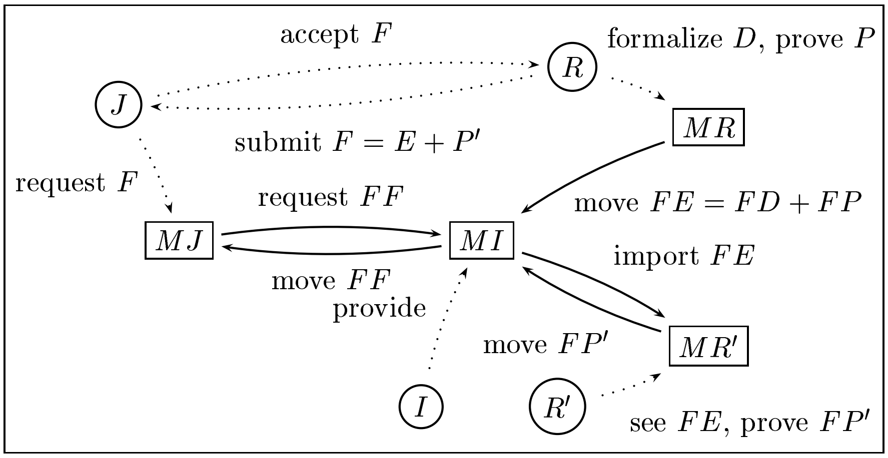


Figure 2: Research Cooperation with distributed MKM

import them into his personal MBase MR'[2]. On this basis R' formally proves FP', and adds it to theory FE, yielding FF the formal version of theory F. Then R and R' submit F to journal J, who evaluates it (possibly via his own personal MBase) and finally accepts F. To publish F on MJ, it requests FF from MI, which moves it there.

1.2 Contribution of this Report

As we have seen in the scenario above, a strict division into archive and scratch-pad knowledge bases is unrealistic, since it does not reflect the current and anticipated nature of scientific communication and publication: Collaboration and theory change occurs at every level and should be supported by an infrastructure for collaborative content management and version control which enables multiple users to work jointly on mathematical theories with minimal interference.

We will develop a general architecture for a collaborative content management extending the CVS architecture and specialize it for mathematical knowledge by taking into account the structure of mathematical documents. For the second task we will build on both the work on structural `diff/patch/merge` utilities in Xml, as well as on the semantic management of change in the Maya system [5].

Even though the work reported here is motivated by the MBase system, it is much more general, since it only depends on the communication format used by the system. The methods are not even specific to the OMDoc format, we will only assume that the knowledge base systems use a similar Xml-based format for communication and provide a way to re-create the original interface documents. This would for instance cover the the HELM system [2], which employs a lightweight infrastructure based mainly on Xml documents and XslT stylesheets for MKM.

[2]Alternatively, R could leave FD and FP in MR and tell R' personally about them, allowing him to import them from MR into MR'; but this is a matter of institute policy, which we will not address here.

2 An Infrastructure for Managing distributed mathematical knowledge cooperatively

The proposed infrastructure for collaborative theory management is largely based on the CVS (**C**oncurrent **V**ersions **S**ystem [12]) architecture. This system is widely used to support collaborative software development, since it combines software versioning with controlled concurrent access to the resources under CVS control. We will briefly review the basic notions of CVS, and describe our multi-level architecture with reference to it.

2.1 Cooperative Version Control in CVS

CVS is a server-based system for concurrent version control, used mainly for software development. The CVS server provides a so-called CVS repository R, which keeps a representation of all committed versions (called *revisions*) of the software together with logging information.

A CVS client C can then *check out* a *working copy* of the software and work on it. Let us for simplicity assume that C checks out the most recent revision in the repository, the so-called *head*. After completing the development task, C can *commit* the changes Δ to the repository, creating a new (current) revision in the repository. She will usually accompany the commit with a short description of the changes; this is also logged in the repository, eventually adding up to a changelog for the software development.

It is a distinguishing feature of CVS that the repository is not locked when a working copy is checked out. So another client C' can also have active working copies of the software and work on them. When C commits, the working copy of C' which was based on the (old) head, is no longer *up to date* with the repository. As a consequence, the changes C' has made to the software cannot be committed to the repository. C' can not simply check out a new working copy from R, since she would lose her work; therefore (upon C's request) CVS merges the changes Δ into C's working copy to keep it up to date with respect to the head of R. Now (after resolving any conflicts introduced by the merge) C' can commit her changes to the repository. Even though conflicts can occur in the merging operation, they are sufficiently infrequent in practice.

We have seen above that version control in CVS protocol is based on the computation, communication and management of differences (changes) to files. CVS uses the `unix` utilities

`diff` for determining the changes in a working copy to be committed to the repository

`patch` for updating old revisions

`merge` for merging changes into a working copy to keep it up to date with the repository.

To facilitate the functionality described above, the CVS server represents committed non-head revisions of files internally as reverse `diffs` from the head revision (which is stored explicitly). Thus the head revision can be served immediately, whereas older revisions can be computed by applying the respective reverse diffs. In this model, a version can be represented as a specific sequence of transformations (edit scripts).

2.2 A multi-level Client/Server Architecture

CVS has a one-level client-server architecture, i.e. all the CVS clients can only communicate with a dedicated CVS server. In the distributed MKM settings like in Figure 2, we have a knowledge base MI that acts both as a repository for MR and MR' and as a client for MJ.

We will say that a knowledge base A is *downstream* from a knowledge base B, iff A is a CVS client of B or any knowledge base C that is downstream from B. The relation of being *upstream* is the converse relation to *downstream*. In Figure 2, MR, MR', and MI are downstream from MJ. Note that commit actions push information upstream and update actions pull information downstream.

A multi-level client-server architecture has inherent advantages: it can, for instance simulate CVS branching: In CVS a branch is used, if a set of clients want to make changes to software that are either too disruptive or too extensive for the usual update/commit cycle. In essence a branch acts as a virtual repository for the development and allows controlling revisions without disturbing the main development (the so-called *trunk*).

In our multi-level architecture, a branch in repository A for clients $S^1, \ldots, S^n$ can be simulated by creating a new knowledge base B downstream from A and upstream from the S^i. B is initialized by checking out a working copy from A, and the S^i can track their revisions in B and eventually commit the result to B. Closing the branch corresponds to deleting the knowledge base B and updating the S^i.

2.3 An Atomized Version Control Relation

The CVS protocol is based on the file system hierarchy for grouping and anchoring user interaction. For instance, update and commit commands issued without reference to a particular file will be applied to all registered files in the current directory.

The file system hierarchy is replaced with a document-centered (given by `omgroup` in the OMDoc representation) or semantic hierarchies (given by theories or development graphs). The notion of a file (or equivalently of an OMDoc document) is only a secondary concept — if present at all — in the conceptual hierarchy of mathematical knowledge management systems. In particular, the level of a file is not the lowest level of an object under version control. This role is taken up by the notion of a mathematical object represented by a top-level OMDoc element. As a consequence, the client/server relation is atomized to mathematical objects instead of files. We speak of the *version control relation* that relates working copies of mathematical objects with their repository instances. Of course this relation must be acyclic.

Just as a file system can contain working copies from multiple repositories, a knowledge base can contain objects that are working copies checked out from different repositories, though for each mathematical object, the version control relation is a tree, i.e. every object has at most one server it can be committed to and updated from. Intuitively, a math object is — for the parent element — as a file for a directory; and files have attributes like creation time, modification time, permissions; so should the math objects, which can be stored in the Dublin Core metadata of OMDoc elements.

2.4 Interaction of Version Control with Distribution and Knowledge Base Consistency

In [17] we have identified four tasks necessary for distributing mathematical knowledge bases: caching, moving, changing, and deleting mathematical objects. Before we give them interpretations in our architecture, let us re-examine the assumptions we based the analysis on; they include (paraphrased):

A3 all mathematical elements have a unique "*defining*" realization in the network of knowledge bases.

A4 mathematical objects are never changed.

Assumption **A3** is directly related to distribution: Every object has a unique description, which is a pair consisting of the URL of the knowledge base and the unique identifier of the object there. All other copies of the object are just cached copies of it.

Assumption **A4** was useful for distribution, since it makes caching and maintenance very simple. Relaxing **A4** — which is the task at hand in this report — has two aspects: How do we ensure consistency in situations where e.g. a definition or theorem that other mathematical objects depend on are changed in mathematically significant ways[3]. We will not deal with this problem here, since it is already studied in great detail in the development graph model [14].

The question we will address in this paper is purely at a protocol level: it can largely be framed in terms of the interaction between **A3** and version control. We will study this with respect to the three distribution tasks identified in [17].

Caching Mathematical Objects: We used assumption **A4** to allow trivial caching. In the new architecture, we identify the caching relation to be the version control relation: to cache a copy of a mathematical object, it is simply checked out from the repository as a working copy. Note that objects that are working copies can never be defining instances of mathematical objects in our model. In the new model cache-consistency is a well-understood problem, since an object can always be updated from its repository. The ensuing conflicts can be resolved by the standard three-way merge methods described e.g. in [20].

Moving Mathematical Objects: One of the most basic procedures is that of moving objects between knowledge bases, e.g. of the theory FF from MI to MJ after the submission described in our scenario. This action can be modeled by adding FF as a defining instance to MJ, deleting FF in MI, and checking out FF from MJ to MI, which acts as a CVS client for MJ for this object. Note that with this construction, we can only move mathematical objects upstream, which is the natural direction.

Deleting and Changing Mathematical Objects: Since we leave the question of maintaining knowledge base consistency to the development graph techniques which entail re-examining mathematical objects that depend on the changed ones, augmenting the "pull" technology of our CVS-like architecture with a "push" component seems advantageous. Note that mathematical objects are always upstream from ones that logically depend upon them. Therefore a knowledge base $\mathcal{M}$ keeps a record of all the downstream knowledge bases, so that these can be notified of any changes and trigger propagation of the change. Apart from notification of dependents this information can be used for optimizations like the following: Whenever $\mathcal{M}$ moves the defining instance of an object $\mathcal{O}$ to some knowledge base $\mathcal{M}''$, then it can send the new location of $\mathcal{O}$ to all downstream knowledge bases, asking them to update their reference objects and thus shielding itself from future requests to $\mathcal{O}$.

3 Computing Differences and Managing Change

In this section we will describe the computational utilities underlying our collaboration architecture. CVS uses the line-based `diff/patch/merge` utilities to compute differences between

[3]Of course changes like correcting typos or changing explanatory text are unproblematic from a consistency point of view.

versions, update files, and merge differences into modified working copies. In applications like ours, where we know more about the structure of the data, we can do better, and arrive at more compact, less intrusive edit scripts[4]. For instance, if we know that whitespace carries no meaning in a document format, two documents are considered equal, even if they differ (with respect to the distribution of whitespace characters) in every single line; as a consequence, the computed difference would be empty.

We will look at different document models and their impact on computing differences between documents in this section. Before we do this, let us briefly clarify what we mean by a document model by comparison to mathematical models. In mathematics, when we define a class of mathematical objects (e.g. vector spaces), we have to say which objects belong to this class, and when they are to be considered equal (e.g. vector spaces are equal, iff they are isomorphic). For document models, we do the same, only that the objects are documents. XML supports the first task by allowing us to specify a document type definition (DTD) or an XML Schema, which can be used for mechanical document validation, but leaves the second to be clarified in the (informal) format specifications.

Listing 1: An OMDoc definition.

```
<definition id="comm-def" for="comm">
  <CMP xml:lang="en">An operation <OMOBJ id="op"><OMV name="op"/></OMOBJ>
     is called commutative, iff
    <OMOBJ id="comm1">
      <OMA><OMS cd="relation1" name="eq"/>
        <OMA><OMV name="op"/><OMV name="X"/><OMV name="Y"/></OMA>
        <OMA><OMV name="op"/><OMV name="Y"/><OMV name="X"/></OMA>
      </OMA>
    </OMOBJ> for all <OMOBJ id="x"><OMV name="X"/></OMOBJ>
    and <OMOBJ id="y"><OMV name="Y"/></OMOBJ>.
  </CMP>
  <CMP xml:lang="de">
    Eine Operation <OMOBJ xref="op"/> heißt kommutativ, falls
    <OMOBJ xref="comm1"/> für alle <OMOBJ xref="x"/> und <OMOBJ xref="y">.
  </CMP>
</definition>
```

Of course, the stronger the equality modulo which differences are computed, the better the edit scripts become. The conceptual core of the MBase data model is given by the OMDoc format [16, 22], which is also used as an interface representation for communication between MBases and their clients. We will base our discussion in this section concretely on the OMDoc document model, building up to it by discussing the underlying XML document model. We will discuss generalizations to other document formats for MKM in section 3.4.

Let us call two documents $\mathcal{M}$-equal, iff they are equal with respect to the document model $\mathcal{M}$, analogously we will call an algorithm an $\mathcal{M}$-`diff` algorithm, iff it computes differences modulo $\mathcal{M}$-equality. In the rest of this section, we will use the OMDoc element in Listing 1 as a running example.

3.1 Using the tree structure of XML Documents

As OMDoc is an XML application, we can make use of the generic tree structure of XML documents. For instance, XML specifies that the order of attribute declarations in XML

[4] Compactness of edit scripts is important for storage and query efficiency in MKM systems, while minimal intrusiveness (patching does not disrupt document structure) is important for humans to track and understand changes.

elements is immaterial, double and single quotes can be used interchangeably for strings, XML comments (<!−−...−−>) are ignored, and whitespace characters in the UniCode serialization is only meaningful in text nodes. As a consequence, the serialization in Listing 2 is XML-equal to the one in Listing 1, but not to the one in Listing 4.

Listing 2: An XML-equal serialization for Listing 1

```
<definition  for="comm"             id="comm−def" >
  ...
  <CMP xml:lang='de'> <!−− note the unabbreviated empty element −−>
   Eine Operation <OMOBJ xref="op"></OMOBJ> heißt kommutativ, falls
   <OMOBJ xref='comm1'/> für alle <OMOBJ xref="x"/> und <OMOBJ xref='y'>.
  </CMP>
</definition>
```

There is a large body of work on using the XML tree structure to compute differences of XML documents modulo XML-equality (see e.g. [24]). The algorithms (see [9] for an introduction) compute partial tree matchings[5] and express these as so-called "edit scripts" that add and delete XML elements and attributes in the source tree to arrive at the target tree. The work has been mainly concerned with finding algorithms for optimal (least-cost) edit scripts and complexity issues. Formats like `XUpdate` [19] (see Listing 3 for an example) use XPATH [11] expressions to identify the elements the instructions act upon.

The central problem of finding corresponding nodes in trees critically depends on the notion of tree-similarity employed. If the document is strongly keyed (e.g. all elements have unique `ID` attributes, which cannot be changed by the user[6] or the knowledge management system employs some node numbering system like the one proposed in [10]), then the key structure gives a very natural notion of node correspondence, and differencing becomes relatively simple. For the un-keyed case, only the notion of structural isomorphism and of ordered and un-ordered trees has been considered e.g. in [9].

Listing 3: An `XUpdate` edit script (partly) updating Listing 1 to Listing 4

```
<xu:modifications xmlns:xu="http://www.xmldb.org/xupdate">
  <xu:variable name="c" select="definition/CMP[0]/OMOBJ[@id='comm1']"/>
  <xu:remove select="definition/CMP[0]/OMOBJ[@id='comm1']/@xref"/>
  <xu:append select="definition/CMP[0]/OMOBJ[@id='comm1']" child="1">
    <xu:value−of select="$c"/>
  </xu:append>
  <xu:remove select="definition/CMP[0]/OMOBJ[@xref='comm1']/*"/>
  <xu:update select="definition/CMP[0]/OMOBJ[@xref='comm1']/@xref">
   <xu:value−of select="'comm1'"/>
  </xu:update>
</xu:modifications>
```

3.2 The OMDOC Document Model

Let us now take a look at how the OMDOC document model can be used for more *semantic* differencing (OMDOC-`diff`[7]).

[5]Which nodes correspond to each other modulo a given notion of tree similarity?

[6]The action of changing keys in the data, can lead to un-intuitive and computationally sub-optimal edit scripts, but does not compromise the method per se.

[7]Note that we are *not* proposing to use mathematical equality here, which would make the formula $X+Y = Y+X$ (the `OMOBJ` with id="comm1" in Listing 4 instantiated with addition for `op`) mathematicallly equal to the trivial condition $X+Y = X+Y$, obtained by exchanging the right hand side $Y+X$ of the equality by $X+Y$, which is mathematically equal (but not OMDOC-equal).

Listing 4: An OMDoc-equal representation for Listings 1 and 2

```
<definition id="comm-def" for="comm">
  <CMP xml:lang="de">Eine Operation <OMOBJ xref="op"/> heißt kommutativ, falls
    <OMOBJ id="comm1">
      <OMA><OMS cd="relation1" name="eq"/>
        <OMA><OMV name="op"/><OMV name="X"/><OMV name="Y"/></OMA>
        <OMA><OMV name="op"/><OMV name="Y"/><OMV name="X"/></OMA>
      </OMA>
    </OMOBJ> für alle <OMOBJ xref="x"/> und <OMOBJ xref="y">.
  </CMP>
  <CMP xml:lang="en">An operation <OMOBJ id="op"><OMV name="op"/></OMOBJ>
    is called commutative, iff <OMOBJ xref="comm1"/> for all
    <OMOBJ id="x"><OMV name="X"/></OMOBJ> and
    <OMOBJ id="y"><OMV name="Y"/></OMOBJ>.
  </CMP>
</definition>
```

The OMDoc document model extends the Xml document model in various ways. For instance[8], the order of `CMP` children of an `omtext` element does not matter, and the distribution of whitespace is irrelevant even in text nodes. More generally, as OMDoc documents have both formal and informal aspects, they can contain *data-set-based* as well as *document-structured* information. At one extreme an OMDoc document contains a formalization of a mathematical theory, as a reference for automated theorem proving systems. There, logical dependencies play a much greater role than the order of serialization in mathematical objects. We call such documents *data set based* and specify the value `DataSet` in the `Type` element of the OMDoc metadata for such documents. On the other extreme we have human-oriented presentations of mathematical knowledge, e.g. for educational purposes, where didactic considerations determine the order of presentation. We call such documents *document-structured* and specify this by the value `Text`. Note that since OMDoc allows to specify Dublin Core metadata [23] at many levels, document-structured and data set based parts can interleave in the same document, allowing OMDoc-`diff` algorithms to take this into account.

Moreover OMDoc uses a variant of OpenMath objects [7] that can be represented as directed acyclic graphs (DAGs; using `ID/IDREF` links) rather than regular trees: an empty element with an `xref` attribute is OMDoc-equal to the element that carries the corresponding `id` attribute. As a consequence, the representations in Listings 1 and 2 are OMDoc-equal to the one in Listing 4, and an OMDoc-`diff` algorithm must generate the empty edit script between all three, while an Xml-`diff` algorithm should generate an extension of the `XUpdate` script in Listing 3.

In particular, the process of exploding the DAG to a tree representation or sharing a tree to a DAG should not result in a different computation. The same applies to the OMDoc representation of proofs, where an additional level of structure sharing is possible. A case where the underlying structure of the data is not tree-like, that is, not based on structure-sharing, is the development graph itself, which can even be cyclic. Here, first steps for defining a correspondence relation and for determining changes have been taken in [3] and implemented in the Maya system.

3.3 Challenges for OMDoc-`diff` Algorithms

As we have shown, taking advantage of OMDoc-equality in computing differences leads to more concise edit-scripts, which is essential in an environment where document processing

[8]As an introduction to the OMDoc format is beyond the scope of this paper, we will assume a basic knowledge of [16] and the material at [22].

applications manipulate mathematical content by acting on internal data structures and generate target documents from these. In such situations, it is impossible to predict which of the possibly many OMDoc-equal representations will be generated. Since in a CVS-like collaborative protocol any `diff` can lead to a conflict that will require human intervention for resolution, the availability of such algorithms will be crucial for adoption.

Of course extending Xml-equality to OMDoc-equality in computing differences breaks the underlying assumptions of the algorithms described in section 3.1. For instance, the DAG-nature of OMDoc documents requires the differencing algorithms to (virtually) expand the objects to tree form while processing them[9].

It seems that techniques from [6] can be used to get around the obvious computational difficulties involved in differencing modulo equality. [6] trivialize the tree matching problem by assuming that all tree representations are "strongly keyed", employing a generalized notion of data base keys to determine element correspondence in Xml documents. They claim that sensible data formats are almost always strongly keyed up to data in Xml text nodes. We have not verified this for OMDoc yet, but for instance even though `CMP` nodes do not have `ID` attributes, they are keyed, since they have `xml:lang` attributes, which must be unique among their siblings. However, `CMP` content however is not keyed, since it is generic text data (which is trivially un-keyed) mixed with representations of mathematical objects represented as content MathML or OpenMath objects (this also caused some addressing problems in the `XUpdate` script in Listing 3). Note that in OMDoc documents managed by MKM systems (as opposed to directly written by hand), the OMDoc `mid` attributes can be used for keying, alleviating the higher computational costs of the un-ordered algorithms somewhat.

Obviously, we need a combination of the Xml tree-based un-keyed algorithms with key-sensitive techniques for our application; such algorithms have been requested, but to the author's knowledge not been reported on so far.

3.4 Modular $\mathcal{M}$-`diff` Algorithms

Given that most of the OMDoc document model is rather standard (DAGs vs. trees, sets vs. lists of children, etc.), it is appealing to develop general $\mathcal{M}$-`diff` algorithms, where the notion of $\mathcal{M}$-equality is specified externally, e.g. by extending the document schema to a full document model.

Note that Xml Schema so far only specifies full document models (i.e. including equality) for so-called *data types* (e.g. "`100`" and "`1.0E2`" are equal as members of the data type `float`). Thus we could define the notion of Xml-Schema-`diff`, which would take these into account, but this is only marginally relevant for our problem here, since it only concerns the leaves of the trees we are dealing with.

Listing 5: Specifying Order in Xml Schema using `xs:appinfo`

```
<xs:complexType name="omtextType">
  ...
  <xs:sequence>
    <xs:annotation><xs:appinfo><mdiff:unordered/></xs:appinfo></xs:annotation>
    <xs:element name="CMP" type="inCMPtype" maxOccurs="unbounded"/>
    <xs:element name="FMP" type="FMP" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  ...
</xs:complexType>
```

[9] In the file system metaphor, this would correspond to following symbolic links.

A more promising avenue seems to be to make use of the `xs:appinfo`[10] element to specify document models for complex types in XML Schemata — as opposed to just content models for validation. Based on the examination of the OMDoc document model in section 3.2, it seems plausible to assume that we could go a long way by specifying

document order e.g. by an element `mdiff:unordered` in Listing 5, and

link semantics e.g. as in Listing 6 where we specify that the `xref` attribute of an OpenMath object means that it represents a copy of the object that carries the corresponding `id` attribute.

Listing 6: Specifying DAG attributes in XML Schema using `xs:appinfo`

```
<xs:attributeGroup name="DAG.attrib">
  <xs:attribute name="xref" type="xs:anyURI" use="optional">
    <xs:annotation><xs:appinfo><mdiff:dag-source/></xs:appinfo></xs:annotation>
  <xs:attribute>
  <xs:attribute name="id" type="xs:ID" use="optional">
    <xs:annotation><xs:appinfo><mdiff:dag-target/></xs:appinfo></xs:annotation>
  <xs:attribute>
</xs:attributeGroup>
```

So, if an XML document (fragment) is an instance of a schema that contains document model specifications like the ones in Listings 5 and 6, then a modular `diff` algorithm can read the schema and customize — multiple times during the parsing process if necessary — the comparison criteria used by the algorithm.

4 XML-CVS Client Sever Protocol

In this section we specify the protocol between the CVS server described in section 2.2, it is loosely based on the CVS protocol, but adapted to the particular architecture proposed here.

Communication between the client and the server can only be initiated by the client. All initial requests therefore are sent from the client and result in a response from the server. The following requests and responses should be seen as the core protocol to which several features may be added at a later stage.

In the following "`path+`" stands for a string specifying a file by its name and location within the directory which is being version-controlled, e.g. `/home/toplevel/documents/myfile.xml`. In the case of directories, "`path`" just specifies the location, e.g. `/home/toplevel/documents/`.

4.1 Requests

`checkout(path+)` This call requests a copy of the complete file in the repository. It takes the `path+` as an argument and changes the client into a state expecting the transmission of the file.

`update(path+, version-number)` This call takes the `path+` and the file's version number as its arguments. The version number specifies the last version of the file that was synchronized between client and server. An update call expects a response containing the `XUpdate` corresponding to the changes that occurred to this file in the repository since the last synchronizing event.

[10] The `xs:appinfo` is introduced in XML Schema expressly for such purposes.

`commit(path+, version-number, XUpdate)` Commit takes three arguments, the first again the `path+`, the second the version number of the file, which changes are to be added to and the third an `XUpdate` corresponding to the structural changes that occurred within the client copy of the specified file since the last synchronization, i.e. the version specified by the version number. The version number is necessary in case there have been changes in the repository since the last synchronization, which need to be merged with the client copy before the `XUpdate` can be committed.

`codir(path)` This call is a checkout call for a whole (sub-)directory. Its argument specifies the location of the (sub-)directory in the repository.

`updatedir(path,[(p1+,v1),(p2+,v2),...(pn+,vn)])` Corresponding to the update call for individual files, `updatedir` updates a whole directory tree. The path argument specifies the (sub-)directory that is supposed to be updated. The second argument is a sequence of n-tuples, referring to the files stored in this subdirectory. Each tuple specifies the `path`, i.e. the location and filename of a specific file in the subdirectory, relative to the subdirectory, i.e. `path+ = path^p5+` for the fifth file in the subdirectory. The second tuple entry contains the version number of this file. `updatedir` is equivalent to a sequence of update requests.

`commitdir(path,[(p1+,v1,XUp1),(p2+,v2,XUp2),...(pn+,vn,XUpn)])` Analogously to `updatedir`, `commitdir` is the commit call for a directory. It is equivalent to calling commit on each file in the directory. The first argument is the directory location, the second argument is a sequence of 3-tuples specifying the commit operation for each file in the directory. In particular, the first entry of the tuple specifies the file location relative to the directory location, the second tuple entry specifies the

4.2 Responses

`newfile(path+, file, version-number)` This is the response to a checkout request. The response contains the `path+` and the complete requested file in a default formatting (as present in the repository) together with a version number so that later references to this version identifies the correct version of the file in the repository.

`updresp(path+, version-number, XUpdate)` This is the response to an `update` request. It contains the `path+` and version number to which this update should be applied to ensure the client copy is XML-equivalent to the current copy on the server, and it contains the corresponding `XUpdate`.

`ciresp(path+, version-number)` or `ciresp(path+, version-number, XUpdate)` This response can take two forms, depending on whether the update in the repository that was requested by the preceeding `commit` call could be completed or whether a merge has to occur first. In the first case, when the commit was successful `ciresp` has two arguments, the `path+` and the version number of the file in the repository. This version number will be stored by the client as the last point of synchronization. If the `commit` was unsuccessful, `ciresp` returns three values, the `path+`, a version number and an `XUpdate`, essentially mimicking a response to an `update` request. The `XUpdate` specifies the changes that occurred since the last synchronization, the version number specifies the point of synchronization and will therefore be equal to that sent in the `commit` request.

5 Realisation of full `OMDOC-Diff` and `OMDOC-Merge` in MoWGLI

The main functionalities required inside the OMDOC-CVS servers are the computation of *differences* between two OMDOC documents and the *merge* of two OMDOC documents either to update a branch of a mathematical development to the actual development or to merge two diverting documents from different sources, for instance different CVS-servers. The function *Diff* takes two specifications S_1 and S_2 and computes a patch that applied to S_1 will result in S_2. Patches are sequences of operational instructions. Suppose, there is a distributed development of OMDOC-documents by two developers A and B. Both sides start with a common document S. While A has refined his document to S', B was lazy and hasn't changed his version of the document. Thus, he can immediately update his document S to S' by applying the patch computed by $\mathit{Diff}(S, S')$, i.e. by replaying the operations A performed on S to obtain S'. Now suppose, B himself has worked on document S which resulted in a new version S''. In this case A and B have to *merge* their documents to a common version S'''. The problem of merging documents is to detect and deal with arising conflicts when both developers worked on the same part of the documents and changed it in different ways.

Both functionalities, *Diff* and *Merge* rely on a specific notion of equality between two OMDoc documents, that takes into account (part of) the semantics of OMDoc language constructs. The same semantical object can have different syntactical representations and we do not want to care about syntactical differences which do not cause any changes in the semantics. For instance comparing Xml-documents, like OMDoc, we want to ignore redundant whitespaces. Concerning the formal parts of OMDoc, the order in which axioms are specified inside a theory can be mostly ignored. Thus, shuffling the sequence of axioms inside a theory should not be considered as a real change of the specification.

To reason about the equality of OMDoc-parts that have a formal semantics we need a logical representation of these parts. Concerning the OMDoc-`diff` procedure for the formal parts in OMDoc-documents, for instance, it is necessary to determine which theories have been added or removed, which symbols have been newly declared or removed, the added or deleted axioms, and how those changes affect existing proofs. We consider two theories as equal if their logical representation is equal. We denote by *structured formal* OMDoc the sublanguage of OMDoc that consists of those OMDoc elements that have a formal semantics, such as theories, imports, symbol declarations, FMPs in definitions, axioms, etc.

For the non-formal parts in OMDoc-documents, as for instance `omgroup`-elements, it needs to be determined whether these elements have been removed or added, and if they have been preserved, which differences there are in the content of these elements, which can either be other OMDoc-elements or plain text. The latter can be realised with a normal Xml-diff procedure, which builds upon a standard CVS-diff procedure for the pure textual parts. However, the former is more complex, since not only the differences in declarations (such as theories, imports, symbols, definitions, and axioms) need to be computed, but also the effect of these changes onto proofs of theorems and existing decompositions of `theory-inclusion` and `axiom-inclusion` elements.

In Section 5.1 we present the architecture that realises an OMDoc-`diff` and in Section 5.2 the realisation of the OMDoc-`merge`.

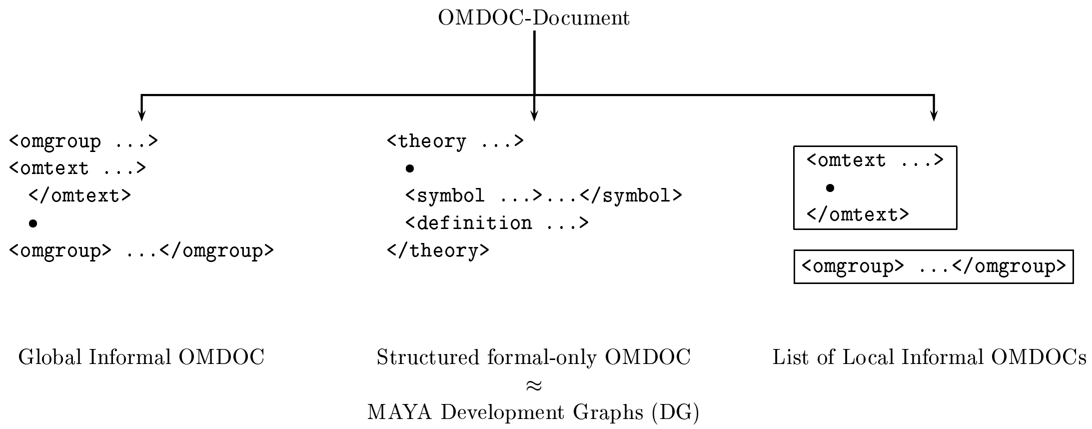


Figure 3: Decomposition of OMDOC-Documents

5.1 OMDoc-Diff

The structured formal sublanguage of OMDoc corresponds to development graphs [15] for which support is implemented in the Maya development graph manager [4]. The Maya tool incorporates a difference analysis algorithm for that sublanguage and especially deals with the effect of changes in declarations onto decomposition information of theory-inclusions and on existing proofs of lemmas. Therefore, we envision to reuse that functionality provided by Maya for the implementation of a difference analysis procedure for full OMDoc.

It remains to develop a methodology for the extension of the difference analysis for structured formal OMDoc provided by Maya to a difference analysis for full OMDoc. To this end consider the following sketch of an OMDoc document:

```
<omgroup ...>
<omtext ...>
  </omtext>
  <theory ...>
    <omtext ...>
      <symbol ...>...</symbol>
    </omtext>
    <definition ...>
    <omgroup ...>...</omgroup>
  </theory>
<omgroup> ...</omgroup>
```

The main problem for the envisioned difference analysis is due to the interleaving of structured formal OMDoc elements with the other OMDoc elements. To remedy that problem and allow for the use of the Maya difference analysis, we separate the structured formal parts from the other OMDoc parts. The decomposition process has two stages of separation, since informal parts occur inside formal parts, for instance the `omtext`-element wrapped around the formal `symbol`-element occurs inside the formal `theory` element. For the above sketched OMDoc document the decomposition is as follows:

```
<omgroup ...>
<omtext ...>
 </omtext>
 •←----------------------<theory ...>
                           •←------------------<omtext ...>
                                                 •←--------------------------<symbol ...>...</symbol>
                                               </omtext>
                          <definition ...>
                          •←-------------------<omgroup ...>...</omgroup>
                         </theory>
<omgroup> ...</omgroup>
```

During the decomposition references with unique identifiers are introduced as placeholders for the removed parts, which allows to recompose the complete OMDoc document from the different parts. After the decomposition, we recompose the structured formal parts only to obtain that part of the complete OMDoc document which belongs to that sublanguage handled by Maya. Thus, any OMDoc document can be decomposed into these parts as depicted in Figure 3. It results in (1) an outer non-formal OMDoc document shell, denoted by *global informal* OMDoc, (2) the pure structured formal OMDoc which is equivalent to Maya development graphs, and (3) a list of informal parts that occurred inside these structured formal parts.

From the obtained pieces of the original OMDoc document, a complete OMDoc difference analysis procedure can be designed which extends the difference analysis provided by Maya. Thereby the results of the difference analysis on the structured formal OMDoc part guide the difference analyses between the elements of the list of informal OMDoc parts. That difference analysis is a normal difference procedure on Xml-documents, which is based on a standard CVS-like difference procedure for pure textual contents. Finally, the same difference analysis procedure is used on the outer OMDoc document shell, to determine the top-level informal differences. Then the difference analysis for complete OMDoc documents is based on these three difference analyses. Its resulting list of differences is assembled from the results of the three difference analyses to a common list of differences by replacing the occurrences of references introduced during the decomposition with their original parts. The overall structure of the global difference analysis procedure is depicted in Figure 4.

5.2 OMDoc-`Merge`

OMDoc-`Merge` again makes use of the separation of formal and informal parts of OMDoc. While the informal parts can be treated by commonly known approaches to merge text files or either Xml-files, special care has to be taken to merge the corresponding formal parts.

In order to merge two formal OMDOC documents we start with their common orginal version S. Using the `Diff`-approach described above we are able to compute the differences d_1 (or d_2, respectively) between the original specification S and the actual document S' (or S'' respectively). Thus, $patch(S, d_1) = S'$ and $patch(S, d_2) = S''$ holds. Thus to merge two formal documents, we have to compute two new patches d'_1 and d'_2 such that $patch(S', d'_2) = S''' = patch(S'', d'_1)$ holds. Obviously there are various trivial solutions to this problem since, up to now, we did not specify any constraints on the shape of the resulting document S''', which could be empty in the worst case. Hence, the first task is to specify constraints about the resulting document S'''. Obviously, the changes of d_1 should be independent of d_2 or, if both have a common domain, should coincide. From a technical point of view, we introduce the notion of a domain $Dom(d)$ for a list of differences d. We use this notion to split d_1 (and

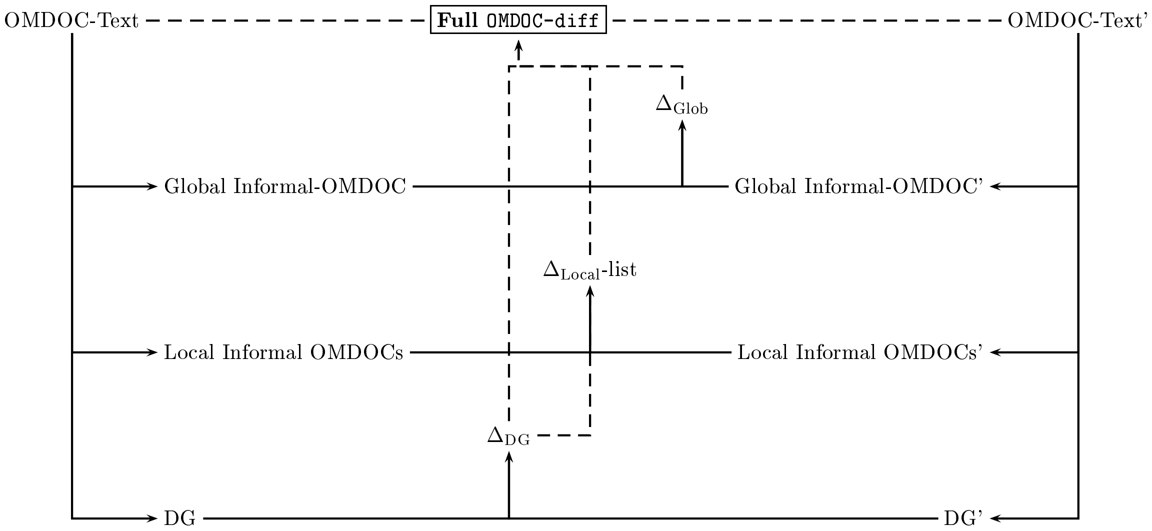


Figure 4: Incremental OMDOC-Diff

d_2, respectively) into two sets of differences d_1' and d_1'' (and d_2' and d_2'', respectively) such that $Dom(d_1') \cap Dom(d_2') = \emptyset$ and $d_1'' = d_2''$ holds. As a result, we apply differences d_2' to S' and d_1' to S'' and compare the resulting documents. A merge of S' and S'' is possible if both documents are (semantically) equal which presumes that we were able to split the differences d_1 and d_2 in such a way that the condition mentioned above holds.

6 A Document-Sensitive XML-CVS Client

One of the remaining problems is to support document-oriented activities in this setting. Consider the following scenario:

1. David writes a large XML document in emacs; checks it into the repository.
2. Stephen notices a typo, fires up his XML editor, corrects the typo, and commits it into the repository.
3. David merges the changes into his local copy.

In the ASCII-diff scenario used by CVS today, the ASCII-diff is computed upon commit in 2.. Since emacs and the XML editor use different indentation, linebreaking algorithms, there is a diff in every single line. The edit script is roughly twice the size of the document (one add- and one delete line per line in the original). In step 3. David gets a totally reformatted document and is exasperated (unlike Stephen he really sees the formatting in emacs).

In the XML-diff scenario, the XML-diff is computed upon commit in 2. There is a diff in only one element, so the diff script is small (independent of the document size). Moreover, in step 3. when David merges in the changes, only the element in question is changed, so David gets to keep his original linebreaking and indentation (except maybe in the changed element, this only makes a difference, if the changed node is a text node, and there the XML editor cannot have reformatted anyway, since it would break XML-equality).

So, we have two large advantages in the XML-diff case:

technical The ASCII-diff repository grows at a rate $O(n*(d+x))$, where n is the number of changes, d is the mean document size, and x is the mean change size. The XML-diff repository grows with $O(d+n*x)$, which makes a large difference for large d and small x (the general case).

ergonomical Users keep their preferred instance of the XML-equivalence class as much as possible. Think of the encoding headaches that you avoid.

communicational All maintainance operations (update, merge, commit) between the client and the MBASE are accomplished solely on the basis of XML-infoset edit scripts. This vastly reduces the amount of bandwidth required since there is no more need to send whole documents over the network (except for the first checkout where the edit script contains the whole file and generates a generically formatted version of the file for the client).

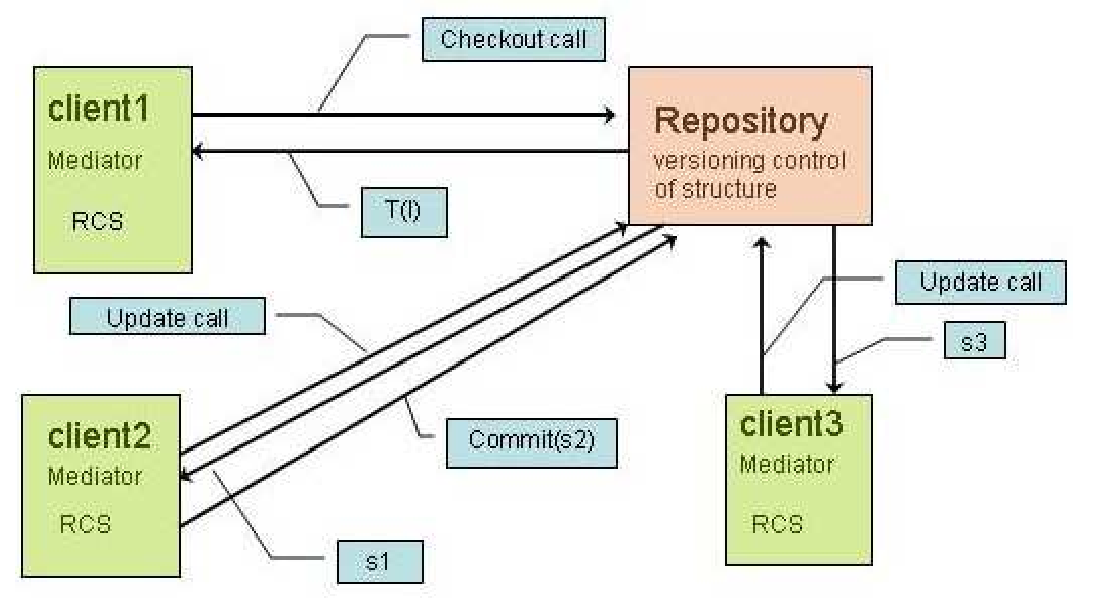
6.1 Architecture

The key to an XML-infoset versioning control while maintaining several (individualized)XML-document presentations (in as far as that is possible in the light of any changes that are made to the underlying XML-file) is the clean separation of individualized presentation (XML document view) and the XML-file structure (XML infoset view). The versioning that we are interested in here is focused on infoset equality, so the concern is to limit versioning to the structural changes in a given document while NOT messing with the layout an individual researcher working on an XML-file uses to view it. The aim is to respect his/her formatting and therefore allow for different users to use different editors to work on the same XML-file without risking the loss of all the work that has been put in to make a local version readable in, say, emacs as opposed to an XML editor. Consequently, in the light of these concerns, a versioning based on a generic document view would be insufficient.

The idea our implementation rests on is the realization that all concerns regarding the XML-document view have to be maintained locally with the client. Furthermore, the aim is to reduce communications between the client and the MBASE to consist exclusively of `XUpdate`-edit scripts, as opposed to sending whole documents and/or ASCII edit scripts as is done by CVS.

The work-horse in this line of communication is the XML-Mediator. In the direction from the MBASE to the client it translates the `XUpdates` into the appropriate ASCII edit scripts in order to update a client copy, without destroying the local formatting. In the other direction it picks out the XML-infoset changes from among the changes a researcher makes on his local copy (which may include presentational changes which are a zero operation in terms of the XML-infoset) and sends these to the MBASE in an commit operation as an `XUpdate`. It is in this latter direction of communication that knowledge of the grammar defining the strucutre becomes relevant.

On the large scale the set-up involves one MBASE that centrally maintains the repository of the XML-file. It is connected to several clients that have their local copy of the XML-file that is synchronised with one particular version of the file in the repository (see Figure 5). Each developer may use their individual document presentation to view the file. The client stores information about the client copy in an RCS, in particular its last synchronised form and all changes made since the last synchronisation. The RCS receives a copy of the XML-file with a synchronisation reference at each synchronisation with the MBASE. This facilitates the



Several client machines work concurrently on files stored in the repository.

Figure 5: The Client Server Architecture

computation of the edit scripts for merge and commit operations. Communication between the MBase and any one of the clients is reduced to checkout, update and commit commands, which result in the transmission of structural edit scripts. No documents or presentational information are sent around.

In the following there is a description of each action triggered by the three commands (See Figure 6).

6.2 Initial Checkout

In an initial checkout from the central repository (MBase), the client machine will obtain a complete structural edit script (s(comp)) from the MBase. This edit script describes the latest version (T(I)) of the Xml-file completely and is basically an edit script of how to get from an empty file to the current file. This edit script is translated into an ASCII edit script and from there a generically formatted copy of the current Xml-file is generated - D1(I). Note that the formatting may be different for each client depending on what editors they are using. The client stores this version of the file together with its synchronisation number (the Roman numbers in the drawings) that indicates when the last synchronisation with the MBase occurred. At this stage a developer may start editing and formatting the Xml-file on the client machine. These changes will be stored in terms of ASCII edit scripts in an RCS on the client machine. At the same time changes may well occur independently in the Xml-file in the MBase.

Pseudo-code

client: checkout(filename, version)

MBase: send(s(comp))

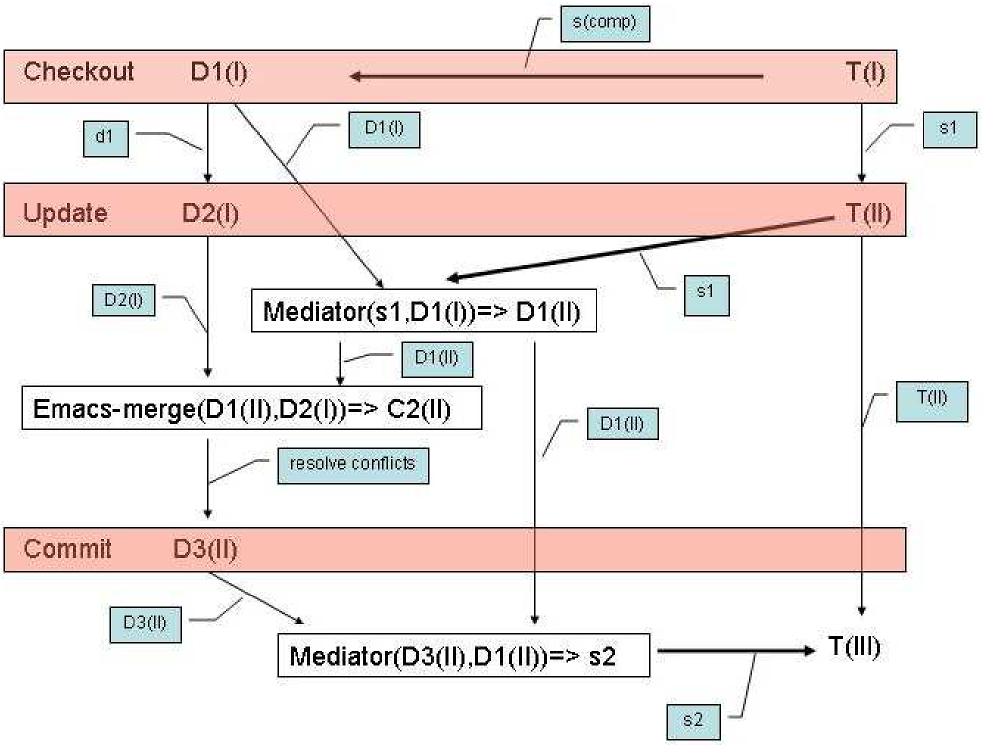


Figure 6: The Architecture

client: defaultFormat(s(comp)) $\rightsquigarrow$ D1(I)

client: currCopy = D1(I)

client: writeToRCS(currCopy, lastSync = I)

6.3 Update

The general situation in which an update call might occur is one where the client and MBase have been synchronised in the past, but changes have occurred in both copies of the Xml-file and those changes may well conflict. So the client may have made changes to D1(I) that are described by the ASCII edit-script d1 and result in D2(I), i.e. an Xml-file that is synchronised with T(I) but has changed from D1(I). Similarly changes might have occurred in the MBase. These have to be structural changes since the MBase is only concerned with structural changes. These changes to T(I) can be described by the `XUpdate`-script s1 and result in T(II), a structurally changed version of the Xml-file (see Figure 6). Note that D2(I) might contain structural changes but needn't. They might just be presentational changes (i.e. indenting) which are structurally a zero-operation.

As a response to the update call, the MBase sends the `XUpdate`-script s1, which contains all the changes on the MBase since the last synchronization with this particular client. On the client machine, s1 and D1(I)(which was stored at the last synchronization) are fed into the Mediator that generates D1(II). This is a version of the Xml-file that contains the changes that occured on the MBase, but not those on the client. In this process the Mediator has to translate the `XUpdate`-script into an ASCII `diff`-script and process it on the Xml-file.

D1(II) and D2(I) are then given to a regular ASCII text merger. Since the changes that occurred locally and those that occurred on the MBase may conflict, the resulting file C2(II) may contain conflicts. However, it contains all the changes that occured on the MBase as well as those made by the client.

The developer is then presented the Xml-file C2(II) containing conflicts and is asked to resolve them. The file resulting from this resolution is D3(II) - its last point of synchronisation with the MBase is II, however, D3(II) may well be structurally different from both D1(II) and D2(I) due to conflict resolution. At this point the update is completed.

Note that in the case where there have not been any changes on the client copy, this procedure becomes almost trivial. Conflicts will not occur update is completed as soon as the `XUpdates` from the MBase are processed into the client copy.

Pseudo-code

client: update(file, lastSync)

MBase: send(s1, v-No) where s1 is the `XUpdate` since lastSync and v-No is the new version number of the file

client: callRCS(lastSync) $\rightsquigarrow$ D1(I), d2

client: Mediator(s1, D1(I)) $\rightsquigarrow$ D1(II)

client: writeToRCS(d2, lastSync = II)

client: textMerge(D1(II),D2(I)) $\rightsquigarrow$ C2(II)

client: resolve conflicts in C2(II) by hand $\rightsquigarrow$ d3

client: writeToRCS(d3)

client: update completed $\rightsquigarrow$ D3(II)

6.4 Commit

It is strongly recommended that a `commit` call follows the completion of a conflict resolution. There is little point in resolving conflicts if one isn't going to let the MBase know about it since it will just mean that these conflicts will have to be resolved again at a later stage.

The `commit`- call can only work if the Xml-file is synchronised with the latest version of the Xml-file in the MBase. If this is not the case, then the `commit` will return with `update` information that needs to be merged into the client copy before the `commit` can be completed.

Here we will assume that the client copy is synchronized with the current copy in the MBase. Consequently, the commit call is very simple: The Mediator, this time operating in the reverse direction is given the Xml-file containing the changes to be committed and the file as it was at the last point of synchronization (stored in the RCS), i.e. in the Figure 2 this is D3(II) and D1(II). Note that these might be structurally different to each other due to the conflict resolution after the merge in the update call. The Mediator calculates the `XUpdate` from these two versions of the Xml-file and sends it to the MBase so that a structural `commit` can be completed there, resulting in T(III). On the client machine, D3(II) then changes to D3(III) - since synchronization has occurred - a copy is stored in the RCS and lastSync is updated to III.

Pseudo-code

assumption here: Xml-file containing changes to be committed is synchronised with the latest version of the Xml-file of the MBase. (this can be done by an update call prior to committing)

client: commit(D3(II))

client: callRCS(lastSync) $\rightsquigarrow$ D1(II)

client: Mediator(D3(II), D1(II)) $\rightsquigarrow$ s2

client: send(s2)

MBase: newStructure(s2, T(II)) $\rightsquigarrow$ T(III)

client: currCopy = D3(III)

client: writeToRCS(currCopy, lastSync = III)

7 Conclusion

We have laid down first ideas for a distribution model for MKM systems that is based on collaborative version control. We have developed and overall architecture, and determined some of the requirements for OMDoc-`diff` algorithms that come from the respective structural invariants of the data in MKM systems. We have exhibited an implementation of a concrete semantic `diff` system based on the Maya system, and addressed document-centered problems of the whole approach by a mediator approach.

We have seen that the proposed distribution architecture can be kept quite close to that of the well-known CVS system[11], and interacts well with the requirements for distribution identified in [17], which is encouraging from an implementation point of view. In particular, we are currently experimenting with the idea to annotate all information necessary for a CVS-like file-based formalism in the `metadata` elements of mathematical objects. We could for instance use the existing Dublin Core `Date` and `Identifier` element for timestamping, and keeping version information. Further information, such as pointers to the repository in working copy objects can be kept in the `metadata/extradata` element provided by OMDoc expressly for this purpose. We will experiment with a HELM [2]-like setup based on OMDoc files on web-servers and implement merging by server-side XslT processing.

The main item for further research is an OMDoc-`diff` algorithm as described in section 5.1. In the literature on version management in Xml, we often hear the argument that difference-computation is not needed in practice, since documents are generated by Xml structure editors, but this only moves the burden from an independent postprocess (implement once) to a module in every editor. Moreover, this would penalize authors for using general Xml editors, since they could only incorporate Xml-`diff` algorithms. Finally, the actual editing process employed by the user may not correspond to the optimal edit script.

Given a good difference computation algorithm, merging can be obtained by relatively simple extensions, especially since our CVS-like architecture allows the usage of the so-called three-way merge (see [20]), where two revisions are compared with respect to a known base revision, from which they have been created. Here, edit scripts for the changes from the base can be computed for both revisions. These can be analyzed and combined to a joint edit

[11] Actually, [6] propose a repository organization that is not `diff`-based, which would be interesting to experiment with, but integrating it into a *collaborative* version control environment is not trivial

script which updates the base revision to the merged revision. [21, 20] present algorithms for three-way merge of XML documents and there are even commercial implementations (e.g. the one described in [18]). Since the merge operation only depends on the edit scripts which act on the generic XML structure, and not on the particular structure of the OMDoc format, we can use these algorithms and implementations off the shelf.

References

[1] 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`.

[2] Andrea Asperti, Luca Padovani, Claudio Sacerdoti Coen, and Irene Schena. HELMand the semantic math-web. pages 59–74.

[3] Serge Autexier, Dieter Hutter, Heiko Mantel, and Axel Schairer. Towards an evolutionary formal software-development using CASL. In C. Choppy and D. Bert, editors, *Proceedings Workshop on Algebraic Development Techniques, WADT-99*. Springer, LNCS 1827, 2000.

[4] Serge Autexier, Dieter Hutter, Till Mossakowski, and Axel Schairer. The development graph manager MAYA. In Hélène Kirchner and Christophe Ringeissen, editors, *Proceedings 9th International Conference on Algebraic Methodology And Software Technology (AMAST'02)*, volume 2422 of *LNCS*. Springer, September 2002.

[5] Serge Autexier, Dieter Hutter, Till Mossakowski, and Axel Schairer. The development graph manager maya (system description). In Hélene Kirchner, editor, *Proceedings of 9th International Conference on Algebraic Methodology And Software Technology (AMAST'02)*. Springer Verlag, 2002.

[6] Peter Buneman, Sanjeev Khanna, Keishi Tajima, and Wang Chiew Tan. Archiving scientific data. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2002.

[7] Olga Caprotti and Arjeh M. Cohen. Draft of the Open Math standard. The Open Math Society, `http://www.nag.co.uk/projects/OpenMath/omstd/`, 1998.

[8] David Carlisle, Patrick Ion, Robert Miner, and Nico Poppelier. Mathematical Markup Language (MathML) version 2.0. W3c recommendation, World Wide Web Consortium, 2001. Available at http://www.w3.org/TR/MathML2.

[9] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 493–504, 1996.

[10] Shu-Yao Chien, Vassilis J. Tsotras, Carlo Zaniolo, and Donghui Zhang. Storing and querying multiversion XML documents using durable node numbers. In *Proc. of The 2nd International Conference on Web Information Systems Engineering (WISE)*, 2001.

[11] Xml path language (xpath) version 1.0. W3c recommendation, The World Wide Web Consortium, 1999. available at `http://www.w3.org/TR/xpath`.

[12] Concurrent versions system: The open standard for version control. Web site at `http://www.cvshome.org`.

[13] 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.

[14] Dieter Hutter. Management of change in structured verification. In *Proceedings Automated Software Engineering (ASE-2000)*. IEEE Press, 2000.

[15] Dieter Hutter. Management of change in structured verification. In *Proceedings of Automated Software Engineering, ASE-2000*. IEEE, 2000.

[16] Michael Kohlhase. OMDoc: An open markup format for mathematical documents. Seki Report SR-00-02, Fachbereich Informatik, Universität des Saarlandes, 2000. `http://www.mathweb.org/omdoc`.

[17] Michael Kohlhase and Andreas Franke. MBase: Representing knowledge and context for the integration of mathematical software systems. *Journal of Symbolic Computation; Special Issue on the Integration of Computer algebra and Deduction Systems*, 32(4):365–402, September 2001.

[18] Robin La Fontaine. Merging XML files: A new approach providing intelligent merge of xml data sets. In *XML Europe 2002 - Conference Proceedings*, 2002.

[19] Andreas Laux and Lars Martin. `XUpdate` - XML update language. Working Draft of the XML:DB Initiative, 2000. `http://www.xmldb.org/xupdate/xupdate-wd.html`.

[20] Tancred Lindholm. A 3-way merging algorithm for synchronizing ordered trees – the 3DM merging and differencing tool for XML. Master's thesis, Helsinki University of Technology, 2001. `http://www.cs.hut.fi/ ctl/3dm/`.

[21] Gerald W. Manger. A generic algorithm for merging SGML/XML-instances. In *XML Europe 2001 - Conference Proceedings*, 2001.

[22] The OMDoc repository. web page at `http://www.mathweb.org/omdoc`.

[23] S. Weibel, J. Kunze, C. Lagoze, and M. Wolf. Dublin core metadata element set, version 1.1: Reference description. DCMI Recommendation, 1999. `http://dublincore.org/documents/1999/07/02/dces/`.

[24] Jin-Yi Cai Yuan Wang, David J. DeWitt. X-diff: An effective change detection algorithm for xml documents. Submitted, `http://www.cs.wisc.edu/ yuanwang/xdiff.html`.