A [insert XML Format] Database for [insert cool application]

Vyacheslav Zholudev, Michael Kohlhase, Florian Rabe
Computer Science
Jacobs University Bremen, Germany

January 29, 2010

Abstract

TNTBase is a versioned XML database; it combines XML fragment access techniques like XQuery with file system functionality and versioning features a la Subversion. We present an infrastructure how the generic TNTBase system can be specialized to include document format-specific services, such as validation, format-specific virtual files and human-oriented presentation.

1 Introduction and Related Work

In [ZK09] we have presented the TNTBase system, a versioned XML database, which combines XML fragment access techniques like XQuery [XQua, XQUb] with the file system functionality and versioning features of Subversion [PCSF08]. This system is intended as a generic storage solution for web applications based on collections of XML documents. However, most such applications are tailored to specific features of their respective document formats. For instance, DocBook [WM08] manuals rely on format-specific style sheets for web presentation and printing, which only work if all documents are valid instances of the DocBook format. The situation is similar for other applications; some even combine multiple document formats and thus need to support multiple validation schemata and presentation pipelines. Furthermore, even homogeneous collections may contain documents in various versions of the underlying formats: indeed the versioning capabilities of the TNTBase system should enable the management of long tails of legacy materials for which format conversion is not cost-effective.

In our experiments with the TNTBase system, it has turned out that many of the format-specific functionalities of the web application layer can be performed by the TNTBase system if we add format-specific interfaces for validation, aggregation, and presentation. We will call this layer TNTBase($\mathcal{F}$), since the services are parametric in the document format $\mathcal{F}$. This extension leads to a refined information architecture of TNTBase-supported web applications; see Figure 1, where the generic TNTBase($\mathcal{F}$) layer takes over format-specific functionalities and thus decreases the effort for developing web applications.

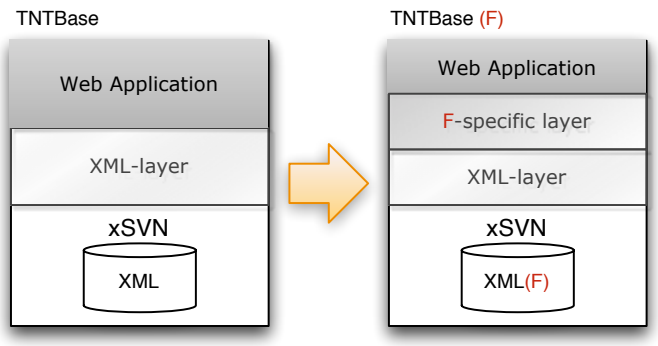

Figure 1: TNTBase($\mathcal{F}$): A Basis for Web Applications

In this paper we present the TNTBase($\mathcal{F}$) infrastructure and its APIs for validation (Section 3), format-specific virtual files (Section 4), and human-oriented presentation (Section 5). We will present much of the functionality of TNTBase($\mathcal{F}$) by using our OMDoc format [Koh06] ($F = OMDoc$) as a running example, since this drives the particular development and integrates many

of the structural prerequisites for virtual documents. The work presented in this paper is based on the experience of by managing a collection of more than 2000 OMDoc documents (ranging from formal representations of logics to course materials in a first-year computer science course) in TNTBase($OMDoc$).

This paper is a short version of [ZKR], which should be consulted for details we had to omit for space reasons.

2 Document-Format-Specific Features for TNTBase

To make this paper self-contained we briefly recap TNTBase (see [ZK09, TNT] for details). Then we take a closer look at the document-specific workflows in document-collection-centered applications that can be generically supported by a versioned XML storage system like TNTBase. This analysis serves as a motivation and basis for the implemented TNTBase($\mathcal{F}$) layer, which we will present in detail in the following sections.

2.1 TNTBase State of the Art

The core of TNTBase consists of the xSVN module, which integrates Berkeley DB XML [Ber] into a Subversion server. DB XML stores HEAD revisions of XML files; non-XML content like PDF, images or LaTeX source files, differences between revisions, directory entry lists and other repository information are retained in a usual SVN back-end storage. Keeping XML documents in DB XML allows us to access those files not only via any SVN client, but also through the DB XML API that supports efficient querying of XML content via XQuery and modifying that content via XQuery Update. As many XML-native databases, DB XML also supports indexing, which improves performance of certain queries.

The TNTBase system is realized as a web-application that provides two different interfaces to communicate with: an xSVN interface and a RESTful interface for XML-related tasks. The xSVN interface behaves like the normal SVN interface — the mod_dav_svn Apache module serves requests from remote clients — with one exception: If one of the committed XML files is ill-formed, then xSVN will abort the commit transaction. The RESTful interface provides XML fragment access to the versioned collection of documents:

Querying: xSVN extends DB XML XQuery by a notion of file system path to address path-based collections of documents. For instance, a part of a query `collection(//papers*/*.xsl)` will address all XSLT stylesheets in the child folders of directories having the `papers` prefix.

Modifying: It is also possible to modify XML documents via XQuery Update. In contrast to pure DB XML, modified documents are versioned, i.e., a new revision is committed to xSVN.

Querying of previous revisions: Although xSVN's DB XML back-end by default holds only HEAD revisions of XML documents, it is also possible to access and query previous versions by providing a revision number if they have been *cached* (by user request). Previous versions cannot be modified since once a revision is committed to an xSVN repository, it becomes persistent.

Virtual Files: These are essentially "XML database views" analogous to views in relational databases; these are tables that are virtual in the sense that they are the results of SQL queries computed on demand from the explicitly represented database tables. Similarly, TNTBase virtual files (VF) are the results of XQueries computed on demand from the XML files explicitly represented in TNTBase, presented to the user as entities (files) in the TNTBase file system API. Like views in relational databases TNTBase virtual files are editable, and the TNTBase system transparently patches the differences into the original files in its underlying versioning system. Again, like relational database views, virtual files become very useful abstractions in the interaction with versioned XML storage.

2.2 Validation and Interface Extraction

One of the salient features of XML as a representation format are its well-established methods for document validation. Indeed most document management workflows take advantage of this and check grammatical constraints on documents by schema validation to simplify further document processing. TNTBase($\mathcal{F}$) supports this practice by validating documents upon commit by default. However, sometimes schema validity is too strong a restriction in practice (e.g., during document authoring or format migration), so the level of validity is configurable. Moreover, many document format constraints — e.g., inter-document link consistency or the absence of link cycles — cannot be checked even by modern schema languages; therefore TNTBase($\mathcal{F}$) provides an API for custom validation modules that can verify that high-level document (collection) integrity constraints are met, if later processing steps require them.

Both high-level validation and further document processing often rely on specific information about other parts of the document collection. To support these processes incrementally, TNTBase($\mathcal{F}$) supports the extraction and indexing of such information upon commit. This approach is analogous to the "separate compilation" paradigm in programming, where declared methods and their types are extracted at compile-time into "signature" files that are used when compiling other files. A prominent example in the XML arena is the extraction of RDF triples for the use in query engines from RDFa-annotated documents. As the specific information is format and application specific, we speak of *interface extraction* for the purposes of this paper. TNTBase($\mathcal{F}$) enables the user to configure interface extraction at commit time; in our experience, this is a crucial ability to make high-level validation tractable. For details of the realization we refer to Section 3.

2.3 Compilation, Browsing and Cross-Referencing

Eventually, the purpose of most document collection applications is to enable humans and machines access to (processed forms) of the document collection. Technically, this usually involves the transformation into specialized presentation-oriented formats for machine (e.g., programming languages) or human (e.g., PDF generation) consumption. Depending on locality properties of the transformation and the rate of change of the document collection, it can be more efficient to execute the transformations when a document is committed to the collection (then we can employ a process analogous to interface extraction) or when the document is requested. Both workflows need to be supported in TNTBase($\mathcal{F}$), and in both cases these documents are automatically served in addition to their sources. If a post-processing of documents for human consumption is defined, TNTBase can provide an enhanced browsing interface. In the future, the TNTBase architecture will make it possible to provide higher-level presentational services such as navigation bar, cross-referencing, and in-place expansion of links, if it is told, which language features constitute document fragments and links to documents or document fragments (see Section 4.2). For details of the realization in TNTBase($\mathcal{F}$) we refer to Section 5.

2.4 Virtual Documents

TNTBase virtual files as introduced above have one great disadvantage: They are not instances of one of the XML formats in the application. For instance the example of a virtual file of section headings of the SVN Book [PCSF08] used in [ZK09] groups the `title` elements in special TNTBase elements, giving a mixed-namespace format specific to the TNTBase system. One generally wants to have virtual files that are "virtual documents", i.e. valid instances of a given format $\mathcal{F}$. This makes *virtual documents* into first-class citizens at the web application level. In theory, if a document format permits to be split into fragments, it becomes possible to recombine the same content fragments into multiple (virtual) documents. If virtual documents make their composition explicit, edited versions can be decomposed into edited fragments, and the changes can be propagated to the original document and other virtual documents using said fragments. But achieving this is surprisingly difficult; both in theory and in practice. In Sections 4.2 and 4.3,

we discuss the properties the target format $\mathcal{F}$ must have to allow this, and the TNTBase($\mathcal{F}$) implementation.

2.5 Realizing TNTBase($\mathcal{F}$)

As described in [ZK09], TNTBase has two major components: xSVN and Java-based Web-application that is built on top of the Java library for accessing xSVN's DB XML directly. Most of the material described in this paper involves both parts and interactions between them. For instance, when one commits a new version of some XML documents and wants their presentation to be cached, this involves committing files in xSVN, figuring out what files the presentation should be generated for, sending the corresponding request to the Java part of TNTBase, generating presentation and finally saving it into DB XML. We omit the technical details about the interactions, since they mainly concern the (painful) integration of C++ and Java. For our initial implementation, we utilize the standard SVN hook mechanisms for pre-commit and post-commit processing. In a nutshell, we use a pre-commit or post commit hook (depending on the processing purpose, e.g. validation or generation presentation) for figuring out what subset of committed files is subject to further processing.

Once this is done, a script sends requests to the TNTBase RESTful interface, where the actual processing is executed. The return codes and error stream are used to notify the committer about the result (e.g., in case of validation errors). For an example see Figure 2. This mechanism is surprisingly flexible and naturally fits into the xSVN approach since it is inherited from SVN. In the future, a tighter integration will add additional scalability and manageability, but our approach shows that that the workflows described in this paper naturally work inside the TNTBase architecture.

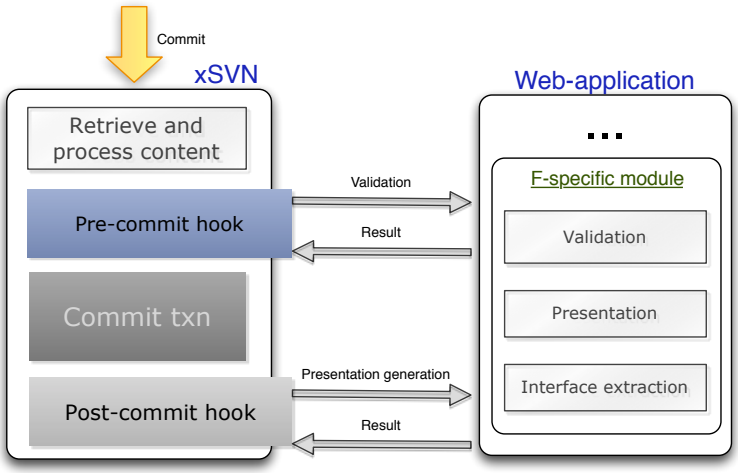

Figure 2: Interactions between xSVN and Web-application

3 Validation and Interface Extraction

TNTBase provides a powerful schema and format-specific validation infrastructure. When committing a set of changes to TNTBase, a validation method is selected for every affected file, that validation is performed, and the xSVN transaction is rejected if validation fails. That ensures that only well-formed documents are stored in TNTBase. The validation method is selected per file via the `tntbase:validate` property (and thus can easily be switched off temporarily). Successful validation may also return extracted information about the committed file and its dependencies, e.g., a document format-specific index, to be stored by TNTBase for later querying.

An important property of TNTBase is that committing a change to a `tntbase:validate` property will result in automatic revalidation of all affected files. Thus, TNTBase guarantees that files with a certain `tntbase:validate` property are well-formed.

Selecting a Validation Method Every directory or file may have the `tntbase:validate` property whose value is a string. If the property is absent, it is assumed to be empty. For

a file its value is a string that defines the validation methods to be applied. For a folder the `tntbase:validate` property is a whitespace-separated list of strings.

If this list is of the form $e_1, m_1, \ldots, e_n, m_n$, the every e_i is treated as a file name extension and every m_i as a validation method. The semantics is that files with the extension e_i are to be validated using method m_i. If the length of the list is odd, the last entry is treated as a default validation method that applies to any file with an extension that was not met before in the list.

When validating a file, the entry in its `tntbase:validate` property determines the validation method (or methods). If there is no matching entry, one is searched in the corresponding property of the parent directory, and so on until a matching entry is found. If the root directory is reached without finding an entry, the predefined validation method `none` is chosen, which does nothing and always succeeds.

Defining Validation Methods Every TNTBase repository comes with a special top-level directory called `admin`. This folder contains all configuration files for that repository so that configurations can be easily made via SVN (and are automatically versioned themselves!). The configuration of validation behavior is done in the `process` subfolder. It may contain a file `methods.xml`, which defines validation methods. If it is not present, no methods are defined.

The content of this file is of the form

1 <methods>$M_1 \ldots M_n$</methods>

where every M_i is of one of the following forms:

<schema name="n" type="t" location="l"/>
<java name="n" class="c"/>

In the first case, validation is performed by schema validation. l is a URL giving the location of the schema, and t is its type: Currently, a user can choose between the `rnc` and `rng` for the text and XML syntaxes of RelaxNG schemata correspondingly. Other XML schema language are planned to be added in the future, e.g. Schematron [sch]. Also, DTD and XML Schema validation is supported automatically by DB XML, and the schemas should be provided as XML files themselves. n is the name of the validation method that occurs in the `tntbase:validate` property.

While the first case, already permits simple validation through different XML schemata, the second case offers full customizability. Here, n is as for schema validation, and c gives the qualified name of the Java class implementing the validation interface. (See the project website `http://tntbase.mathweb.org` for documentation of the interface.) These implementation classes must be in the Java `classpath` provided when starting TNTBase.

Applying Validation and Interface Extraction Every m_i in the `tntbase:validate` property is of the form $s_i?(+v_i)?$, where s_i is the name of an XML schema validation method and v_i is the name of a Java validation method. The latter may generate any serializable content. The intuition here is that the validation automatically produces information that is desirable to store for later use. These can be transformations of the committed file into human- or machine-readable formats such as XHTML or RDF.

These results are stored as XML documents in TNTBase and every such document is associated with a real XML file in TNTBase. They are not shown in the TNTBase file system and can be queried via a separate TNTBase interface. For that purpose, the `method` element from above receives one additional optional attribute `output`. If the value of this attribute is *true* TNTBase will cache the files containing the extracted information and will make them accessible via dedicated RESTful methods.

In particular, the output of validating the file `PATH` is available at `http://tntbase.your.host.org/restful/integration/output/METHOD/PATH`, and XQuery access to all output files is possible via `http://tntbase.your.host.org/restful/integration/query/METHOD?query=XQUERY_EXPRESSION`. In both cases, `METHOD` is the name of the validation method.

As an example, we give the current setup of the OMDoc repository used in the Latin project [LAT]. Here the top-level folder of the repository carries the property `tntbase:validate=omdoc mmt-rng+mmt`. This means that all files with the `omdoc` extension are to be validated using first the `mmt-rng` and then the `mmt` method. A subfolder `inprogress` contains work in progress and carries the property `tntbase:validate=none` to avoid validation.

The `methods.xml` file looks as follows:

```
<methods>
  <schema name="mmt-rng" type="rnc" location="/var/www/tntbase/schemata/mmt.rnc" type="rnc"/>
  <java name="mmt" class="info.kwarc.jomdoc.frontend.TNTValidate" output="true"/>
</methods>
```

Thus, the method `mmt-rng+mmt` consists of calling first the schema validation using the schema `mmt.rnc`, followed by a custom java implementation in the class *TNTValidate*.

The extraction method of the class *TNTValidate* returns a set of RDF triples that represent the `omdoc` file according to the `omdoc` ontology. Because the value of the `output` attribute is `true`, the generated RDF theory (ABox) is accessible via the RESTful TNTBase interface. For example, the RDF triples of the file `math/algebra/algebra1.omdoc` are accessible as `http://tntbase.your.host.org/restful/integration/output/mmt/math/algebra/algebra1.omdoc`.

4 Virtual Documents

In this section, we will take a closer look at virtual documents: As they are surprisingly complex to understand, we introduce the concept and its realization by an example in Section 4.1 before we develop a precise terminology to capture all aspects of the concept: Sections 4.2 and 4.3 develop a set of prerequisites for the document format to make use of virtual documents and show how they can be achieved generically.

4.1 Virtual Documents by Example

To fortify our intuition about virtual documents let us consider the following situation: We have a set of exercises with problem statements and solutions in our document collection. Say they are marked up in the following way:

Listing 1: An Exercise with Solution

```
<exercise>
  <problem>P</problem>
  <solution>S</solution>
</exercise>
```

Now we want to make them available in two forms: without solutions to students and with (master) solutions to the teaching assistants. In this situation, virtual documents are ideal. Generally, for a virtual document we specify an XML file like the one in Listing 2. It consists of a skeleton and a list of external queries.

Listing 2: A Virtual Document for Practice Exercises

```
<tnt:virtualdocument xmlns:tnt="http://tntbase.mathweb.org/ns">
  <tnt:skeleton xml:id="exercises">
    <omdoc xmlns="http://omdoc.org/ns" xmlns:dc="http://purl.org/DublinCore">
      <dc:title>Exercises for GenCS</dc:title>
      <dc:creator>Michael Kohlhase</dc:creator>
      <omdoc>
        <dc:title>Acknowledgements</dc:title>
        <omtext>The following individuals have contributed material to this document</omtext>
        <tnt:xqinclude query="distinct-values(collection(/gencs/*/*.omdoc)//dc:author)">
          <tnt:return><omtext><tnt:result/></omtext></tnt:return>
        </tnt:xqinclude>
      </omdoc>
```

```
        <omdoc>
          <dc:title>SML Exercises</dc:title>
          <tnt:xqinclude>
16          <tnt:query name="xq.SMLex"/>
            <tnt:return><exercise><tnt:result/></exercise></tnt:return>
          </tnt:xqinclude>
        </omdoc>
        <omdoc> ...</omdoc>
21    </omdoc>
    </tnt:skeleton>
    <tnt:query name="xq.SMLex>
      for $i in collection(/gencs/SML/*.omdoc)//exercise
      return $i/*[local-name() ne 'solution']
26  </tnt:query>
    ...
</tnt:virtualdocument>
```

Conceptually, the skeleton consists of the top-level parts of the intended exercises document, where some document fragments have been replaced by embedded XQueries that generate them. In general, queries have the form

Listing 3: A XQuery Fragment Reference

```
<tnt:xqinclude>
2   <tnt:query>q</tnt:query>
  <tnt:return>R</tnt:result>
</tnt:xqinlcude>
```

where q is an XQuery and R consists of a result expression where the `tnt:result` element will be replaced with the results of q. In line 7-9 we have such a query in a syntactic variant where <tnt:xqinclude query="q">R</tnt:xqinclude> was used to abbreviate the primary query form above for queries that do not contain embedded elements. The result of this particular query would be a list of author names wrapped in an `omtext` element. The query in lines 15-18 uses another useful syntactic feature: it refers to the query by reference to enable reuse and sharing for virtual documents (see below). [1] The result of this query would be a list of exercises of the form

```
<exercise>
  <problem>P</problem>
</exercise>
```

Note that the default and the `dc` namespace within the query are inherited from the dominating `omdoc` node.

If we want to edit a virtual document, then we should tell TNTBase to send it to us in a special editing mode. Then the relevant parts of our virtual document will be of the form

```
<exercise>
2   <problem tnt:srcfile="/gencs/SML/prob1.omdoc" tnt:xpath="(omdoc/exercise/problem)[2]">P</problem>
</exercise>
```

Note that the result fragments have been decorated with `tnt:srcfile` and `tnt:xpath` attributes that specify the origin of the text fragments. These attributes are syntactical variants of the source references in virtual files (see [ZK09]) and enable editability and transparent commit in the same way.

Thus, the virtual document in Listing 2 indeed expands to the desired exercises document after XQuery processing. In TNTBase($\mathcal{F}$), virtual documents are created simply by committing a *virtual document specification* (the XML file in Listing 2) to the xSVN repository and executing an additional method of TNTBase that takes as an input the path of a virtual document specification and a path of a new virtual document itself. Thus, we can create multiple virtual documents based on a single virtual document specification. For instance, the above virtual document specification's

[1] In the future, we intend to integrate the XQuery module system into virtual documents so that individual XQuery function and variable declarations can be shared between queries.

XQueries can be changed so that they select problems in the directory where virtual document resides. This is done by for $i in collection(./*.omdoc)//exercise return $i/*[local−name() ne 'solution'], where the first "." in the query stands for the *current directory*. Thus we write a specification once, and may use it for creating virtual documents without solutions in different folders separated by topic.

The reference-based setup caters for a wide variety of reuse scenarios. For instance the document for the GenCS teaching assistants could be specified by the following virtual document specification, which reuses the skeleton from Listing 2:

```
<tnt:virtualfile xmlns:tnt="http://tntbase.mathweb.org/ns">
  <tnt:skeleton href="../exercises.vf#exercises"/>
  <tnt:query name="xq.SMLex>
    for $i in collection(/gencs/SML/*.omdoc)//exercise return $i
  </tnt:query>
  ...
</tnt:virtualfile>
```

Note that the new query does not remove the solutions. It is lexically captured by the named reference in the skeleton. One may think of method overloading in Java or C++. It is also possible to have an empty `tnt:query` element is the specification. Then it becomes an *abstract* specification, and can be compared to abstract methods in Java or pure virtual functions in C++.

4.2 Document Formats and Document Fragments

In the example above we have been very informal; to fully understand the concept of virtual documents we have to work a little harder and pin down the relevant concepts. We will provide the relevant definitions and explain them by examples that show the problems we have glossed over above.

Definition 1 (Document Format)**.** A *document format* consists of

1. a formal language, whose words we call *documents*,
2. a formal language of *fragments* over the same alphabet,
3. for every document D a set of pairs (p, F), where F is a fragment occurring as a sub-word of D and p is some expression, called the *position* of F in D.

We write $D(p) = F$ to retrieve the fragment at position p.

The intuition of fragments is that fragments occur as components of documents. Document formats are typically given as a context-free grammar that subsumes the language with some non-terminal symbols designated as the fragment-producing ones. For example, we define the format XML/XPath as follows: The documents are the XML documents, and the fragments of an XML document D are the element nodes occurring in D; the fragment positions are given by XPath expressions [BBC+07]. Another XML-based document format is XML/id: Here, only the nodes with `xml:id` attributes [MVW05] are fragments, they are addressed by the values of of `xml:id` attributes which serve as positions. For text documents, we can use substrings as fragments together with line and column numbers as positions (or paragraphs and paragraph numbers if we use empty lines as paragraph separators). Furthermore, any formal language trivially becomes a document format if every document has itself as the only fragment.

Any particular XML language becomes a document format in the sense of Definition 1 by treating it as a special case of XML/XPath. However, it is often useful to define fragments differently, in particular to restrict the fragments to ones that carry self-contained meaning in the document format. After all, documents are composed of fragments that represent logical structuring units.

The OMDoc1.6 [2] format has been designed around the notion of "semantic fragment". These are represented by tags such as *omdoc* (used both for documents and document sections), *theory*,

[2] The OMDoc1.6 format is the radically redesigned successor of OMDoc1.2 [Koh06] which is the first draft of the OMDoc2 format. It is based on the semantic data and referencing model of [RK09], which makes it an interesting case study for our purposes here

symbol, *axiom*, *notation*. Other element nodes, such as the ones representing individual mathematical expressions, are not considered fragments. In the OMDoc1.6 design the fragment nodes all have `name` attributes: Any element with a name yields a fragment, and any tag that is worthy of being extracted as a semantic fragment gets a name. Contrary to `xml:id`s, the `name` attributes have to be unique only within the scope given by their parent fragment, which gives rise to a hierarchical naming scheme that is more adequate for mathematical practice. For example, the relative URI `algebra.omdoc/groupoids?Monoid?comp` [3] refers to the declaration of the composition operation in the theory of monoids occurring in the section on groupoids in the document `algebra1.omdoc`. The general form of an OMDoc fragment position is $sec_1/\ldots/sec_m?mod_1/\ldots/mod_m?sym$, where the sec_i are section names, the mod_i are the names of nested modules in the OMDoc module system [RK09], and sym is a symbol name. Clearly, the set of fragments of an OMDoc document can be naturally included into the set of fragments of the same document considered as an XML/XPath document.

We use names instead of `xml:id`s as fragment positions because the latter have to be document-unique, whereas in virtual documents, we want to be able to flexibly recombine existing fragments into new documents. Already the next definition could lead to name clashes if we used `xml:id`s to identify fragments.

Definition 2. We say that a document format $\mathcal{F}$ supports *reference expansion* if it provides

1. for every document D in $\mathcal{F}$, a distinguished set of fragments called *fragment references*,
2. for every fragment reference R in $\mathcal{F}$, a list $(d_i, p_i)|_{i=1}^n$, called *results*, where d_i is a document URI and p_i is a fragment position in the document referenced by d_i,
3. an operation $exp(d, p)$ that given a URI d of an $\mathcal{F}$ document and a position p in it returns a document fragment in $\mathcal{F}$,

such that D is semantically equivalent to the document that arises from replacing a fragment reference R with the concatenation $exp(d_i, p_i)|_{i=1}^n$. A document is called *fully expanded* if it contains no fragment references.

Technically, the precise definition of semantic equivalence must be given by the document format as an equivalence relation between documents. The right choice of the semantic equivalence relation can be quite difficult, and often different notions of equivalence must be employed in different contexts. For example, replacing relative URIs in an XHTML document with appropriately resolved absolute ones will usually not change the semantics of the document. However, in certain situations, it is still desirable to forbid such a transformation, e.g., when a directory of inter-linked documents is often moved to different locations. For simplicity, we will not go into details and simply assume such an equivalence relation to be present.

The intuition behind reference expansion becomes clear if we define it for the document format XML/XPath from above: It becomes a document format XML/XPath/XInclude with reference expansion by using XML inclusions via XInclude [MOV06], i.e., the fragment references are the `xi:include` elements. XInclude (essentially) requires that processors obtain the document fragment(s) referenced by the `href` and `xpointer` attributes of the `xi:include` element and replace the `xi:include` element with them, which we take as semantic equivalence. Thus, we can define that for every `xi:include` element, the result list is $(d, p_1), \ldots, (d, p_n)$ where d is given by the `href` attribute, $p_1, \ldots, p_n$ are the normalized XPath expressions identifying the nodes matched by the `xpointer` attribute.

At first it might seem that one should always put $exp(d, p) := D'(p)$ where D' is the document at URI d. However, it is useful to permit the document format to perform an arbitrary operation instead – typically this operation consists of computing $D'(p)$ and then applying some post-processing to it. For example, `xi:include` computes $exp(d, p)$ by adding an attribute `xml:base` to $D'(p)$ because $D'(p)$ might inherit its `xml:base` from an ancestor node within D'. Similarly, XML namespace attributes inherited from an ancestor may have to be added to $D'(p)$.[4]

[3] The use of an additional ? in the query is unusual but legal. It leads to a very compact notation, which is important in OMDoc.

[4] The latter is usually not necessary for XInclude, which technically acts on info sets rather than documents.

OMDoc1.6 integrates a custom syntax for reference expansion. Every tag designating a fragment may alternatively occur as an empty element node with an `href` attribute instead of a `name` attribute. The value of this attribute is of the form d/p where d is a URI without query or fragment and p is an OMDoc fragment position. The list of results always has length 1 and is given by (d, p). (Note that since both d and p may contain slashes, the computation of (d, p) has to be carried out by a server providing the resource d). To define reference expansion, let D' be the document at d. Then $exp(d, p)$ returns $D'(p)$ with an added attribute `base` whose value references the parent fragment of $D'(p)$ in D'. For example, assume the following OMDoc document D' at URI d:

```
<omdoc>
  <omdoc name="div_1">
    <theory name="th_1"><constant name="c_1"/></theory>
  </omdoc>
</omdoc>
```

A document D may contain a fragment reference to the theory `th_1` in the section `div_1` of the form <omdoc href="d/div_1?th_1"/>. Then $exp(d, \texttt{div}_1?\texttt{th}_1)$ yields

```
<theory name="th_1" base="d/div_1"><constant name="c_1"/></theory>
```

The added `base` attribute serves two purposes. Firstly, it functions as a base address relative to which all semantic references occurring inside `th_1` are resolved. (There are none in this example, but practical theories contain large numbers of mathematical expressions containing semantic references to theories and constants.) Secondly, it remembers the origin of the theory `th_1` so that a link between the theory defined in D' and its copy in the fully expanded version of D is maintained. This link is important for several applications, and below we will use it in particular to propagate changes made to `th_1` from D to D'.

Definition 3 (Document Contraction)**.** In a document format that supports reference expansion, we speak of *document contraction* if a document D is transformed into a semantically equivalent document D' by replacing a fragment of D with a fragment reference.

A fragment that is not simply a fragment reference is called *fully contracted* if no further contraction is possible. A fully contracted fragment is called *atomic* if it is also fully expanded. S is called the *skeleton* of a document D if S arises from D by contracting exactly the atomic fragments. Finally, we say that a document format supports document contraction if every document has a skeleton.

Both XML/XPath/XInclude and OMDoc support document contraction. However, for XML/XPath/XInclude, the atomic fragments are the empty elements, and thus the skeleton is not interesting because it is isomorphic to the original XML document. But for document formats with custom definitions of fragments such as OMDoc, the skeleton yields an interesting decomposition into medium-sized chunks.

A good example of a skeleton is the table of contents with every node representing one entry. This can also serve as a guiding intuition to define document formats: An XML node should count as a fragment if it could occur in a table of contents. This motivated our choice of fragments for a document format for DocBook: The skeleton of a DocBook document is just its table of contents. More precisely, we can say the following.

Theorem 1. *Let $\mathcal{F}$ be a document format arising from XML/XPath/XInclude by restricting the documents to a sub-language of XML and by restricting the fragments of a document to element nodes with certain tags. Further assume that the $\mathcal{F}$-documents may contain `xi:include` fragments and that the specification of $\mathcal{F}$ imposes semantic equivalence under reference expansion as defined above for XML/XPath/XInclude. Then $\mathcal{F}$ supports reference expansion and document contraction.*

As an example, consider the left OMDoc document in Figure 3 with URI d. It can be contracted to the one on the right: Here all fragment references are relative to the base URI d, which receives a trailing slash so that, e.g., div_1/div_1?theory_1 resolves to d/div_1/div_1?theory_1. Note how the

```
<omdoc base="d/">
 <omdoc name="div_1">
  <omdoc name="div_1>
   <theory name="th_1">
     <constant name="c_1">
      <type>T</type>
      <definition>D</definition>
     </constant>
   </theory>
  </omdoc>
  <omdoc name="div_2">
   <theory name="th_1">
    ...
   </theory>
   <theory name="th_2">
    ...
   </theory>
  </omdoc>
 </omdoc>
</omdoc>
```

```
<omdoc base="d/">
 <omdoc name="div_1">
  <omdoc name="div_1>
   <theory name="th_1">
    <constant
      href="div_1/div_1?th_1?c_1"/>


   </theory>
  </omdoc>
  <omdoc name="div_2">
   <theory href="div_1/div_2?th_1"/>


   <theory href="div_1/div_2?th_2"/>


  </omdoc>
 </omdoc>
</omdoc>
```

Figure 3: Contracting Documents

contraction status is different between the fragments at position div_1/div_1 and div_1/div_2: The former is skeletal, the latter is fully contracted. Such an OMDoc document typically arises after a fully contracted document has been served and the user has interactively expanded some of the fragment references.

4.3 Virtual Documents

Contraction permits to decompose a document into its fragments, and by expanding fragment references this process can be inverted. This is useful in itself, but it also opens the door to a powerful application: New documents can be composed partially or completely out of fragments from existing documents. If we follow this idea systematically, we can conceptually consider all documents to be skeletons that pick fragments from a shared reservoir. In particular, there is no conceptual distinction needed between the document in which a fragment is created and the documents, from which that fragments is referenced.

Virtual documents are simple as long as they are created or read but not changed. But assume a document D' that contains a reference to a fragment F of D, and assume that the reference has been expanded. Further assume, we changed F to F' within this expanded version of D'. When these changes are committed, we have two options:

A The changes to F are propagated to D. D' is contracted to its original form, which means D' did not change at all.

B D' is stored in its expanded form, which means that a new atomic fragment is stored for F', and a new revision of D' is stored where the fragment reference now points to F'.

Clearly, both options are desirable under different circumstances. The following definition is strong enough to handle both cases.

Definition 4 (Virtual Documents)**.** *Assume a document format $\mathcal{F}$ such that 1. all documents are XML documents, 2. all fragments are XML element nodes, 3. reference expansion and document contraction are supported. We say that the format supports* virtual documents *if it provides a partial function $contr(F')$ that takes a document fragment F' and (if defined) returns (d, p, F) such that $contr(exp(d, p)) = (d, p, D(p))$ where D is the document at URI d.*

The intuition is that $contr(F') = (d, p, F)$ holds iff F' arose from a reference expansion where (d, p) is the URI-position pair used to identify the referenced fragment. In that case, F arises by inverting whatever modification $exp(d, p)$ made to $D(p)$. Then F represents a new version of F' that can be propagated to the document at URI d.

To see more clearly how this works, we give a document format VXML for virtual XML documents. We define it as follows:

- The documents are XML documents.
- The fragments are the XML element nodes with XPath expressions as their positions.
- The fragment references are of the form of Listing 3. where Q is an XQuery returning a list $(d_i, p_i)|_{i=1}^{n}$ of results, and R is arbitrary XML in particular using an element node <tnt:result/>. The namespaces of the query Q are inherited from the `tnt:query` node.
- For every result, the node $exp(d_i, p_i)$ is defined as follows:
  1. Take the node list R and add to all top-level nodes an attribute tnt:virtual="q" where q is the XPath expression of the tnt:xqinclude element. Let this yield L.
  2. Let N_i be the node list arising from replacing the node <tnt:result/> in L with: the node $D_i(p_i)$ (including its in-scope namespace attributes) where D_i is the document at URI d_i with the following additional attributes:
     - an attribute xml:base as in the XInclude specification,
     - an attribute tnt:added="xml:base" if an xml:base attribute was added,
     - the attributes tnt:srcfile="d_i" and tnt:xpath="p_i".
  3. Return the concatenation $N_1 \dots N_n$.
- For every fragment F' the function $contr(F')$ is defined whenever F' has exactly one descendant node N with an attribute of the form tnt:origin="d#xpointer(p)"; and in that case it returns (d, p, F) where F arises from N by removing tnt:origin attribute and if applicable the xml:base attribute.

After retrieving VXML documents from TNTBase they can be dynamically expanded – e.g., by using an interactive browser or a TNTBase-supporting editor – in which case they will have tnt:virtual="q" attributes. When committing the document, TNTBase (i) replaces all elements with such attributes with the original fragment reference, which can be retrieved from the database using the XPath expression q, and (ii) propagates the respective changes to the origin document using the $contr(-)$ operation. This corresponds to option A above. The user can choose option B by removing the `tnt:virtual` attribute.

The above definition of VXML assumes that all results of the XQuery correspond to physical nodes in some documents rather than being constructed during the XQuery. This is a necessary prerequisite for the propagation of changes. In general, this may only hold for some result nodes, in which case only changes to those nodes can be propagated. TNTBase is able to handle that situation as well, but we avoid the description here for simplicity. Note that in our example above, the OMDoc format supports virtual documents because the `base` attribute mentioned in the definition of the OMDoc reference expansion determines the origin of a fragment.

5 Presentation: Compilation and Browsing

The presentation support of TNTBase is targeted at providing document format-specific browsing interfaces. We use the term *rendering* for the process of transforming documents into human-oriented presentation formats such as XHTML.

Rendering can be done at commit or at view time. Commit-time rendering is generally preferable because it makes viewing faster. However, it is not suitable for interactive editing of documents and for browsing virtual documents which are created on the fly using dynamically executed XQueries. Furthermore, the presentation of a document can be affected by changes in other documents, e.g., if a document contains fragment references that are expanded before rendering. Therefore, TNTBase supports both commit- and view-time rendering.

Both types of *rendering* are controlled by a configuration file `browsers.xml` in the folder `admin/viewing`. Its content is of the form

---

<browsers>$B_1 \dots B_n$</browsers>

---

where every B_i is of one of the following forms:

<xslt name="n" location="l"/>
<java name="n" class="c"/>

The semantics is very similar to the one of the corresponding entries of the `methods.xml` file. The main difference is that n is not referenced in a subversion property but directly in a URI.

For *view-time rendering*, if `PATH` is a path in a repository with the URL REPOS, then the URL REPOS/restful/presentation/render/n/PATH yields the version of that file rendered using the method n. In particular, PATH need not refer to a file but can also contain queries that generate virtual documents on the fly. Currently two types of rendering methods are supported: XSLT transformations and custom Java rendering. In the latter case, a user has to provide an implementation of certain TNTBase interfaces that return rendered documents.

For *commit-time rendering*, the situation is very similar. The main difference is that a user has to generate (via an ad-hoc TNTBase script) the post-commit hook that internally will invoke the necessary presentational caching method via the RESTful interface. When creating such a hook, the user has to provide the name of a browser, say n, that is declared in `browsers.xml`. Once this is set up and rendered documents have been generated at commit time, one uses the URL REPOS/restful/presentation/view/n/PATH to view the cached presentation. In this case not every PATH will yield a document because the commit could have occurred before the commit-time rendering was configured. Furthermore, it is possible to force a refresh of the cached presentation by retrieving the URL REPOS/restful/presentation/cache/n/PATH. (In fact, this is what the above-mentioned post-commit hook does.)

6 Conclusion and Future Work

We have presented TNTBase($\mathcal{F}$), a format-specific extension of the TNTBase system that allows the (web) application developer to simply configure common format-specific workflows like validation, compilation, and presentation within the TNTBase system. This considerably reduces the application logic of (web) applications that manage large, changing XML-based document collections. The TNTBase($\mathcal{F}$) framework is implemented as an extension of TNTBase and can be obtained from the project website `http://trac.mathweb.org/tntbase`. TNTBase($\mathcal{F}$) is under active development and the project website should be consulted for details and the evolving state of implementation.

To get a better intuition for the power of the TNTBase($\mathcal{F}$) framework, let us look at the presentation of an ontology in Figure 4. This example from [Lan10] shows a heavily cross-referenced XHTML+MathML rendering of the underlying OMDoc sources, where much of the structural relations have been annotated in RDFa. The rendering and RDFa annotation process made use of the extracted interface information (see Section 3), which is then picked up by JOBAD [GLR09], a JavaScript library that instruments the XHTML for interactivity. For instance, JOBAD can use CSS properties to collapse a proof or embedded MathML `maction` elements or pick a different (pre-generated) notation variant. But JOBAD can also use AJAX-style callbacks to the TNTBase($OMDoc$) system for definition lookup: For any symbol a right-click menu item will query TNTBase for the definition of the corresponding constant, which will then be served by TNTBase($OMDoc$) and displayed in a popup by JOBAD. This example uses only the techniques from Sections 3 and 5: Pre-existing validation, annotation, rendering, and interaction functionalities were integrated into TNTBase($\mathcal{F}$), which also serves as a web application platform [KGLZ09]. The work reported here was influenced by discussions of how to integrate TNTBase into the systems of the KWARC research group, in particular the SWiM system. The latter is an OMDoc-based semantic Wiki for scientific/technical documents [Lan08, Lan10], in which workflows similar to the ones described in this paper had to be established for the integration of the OMDoc format into the underlying IkeWiki platform.

We are using TNTBase($OMDoc$) in daily practice exploring the possibilities of TNTBase($\mathcal{F}$) for a semantic document format that is designed to support document aggregation and to stretch the limits of document modularization. The newly implemented virtual documents allow to aggre-

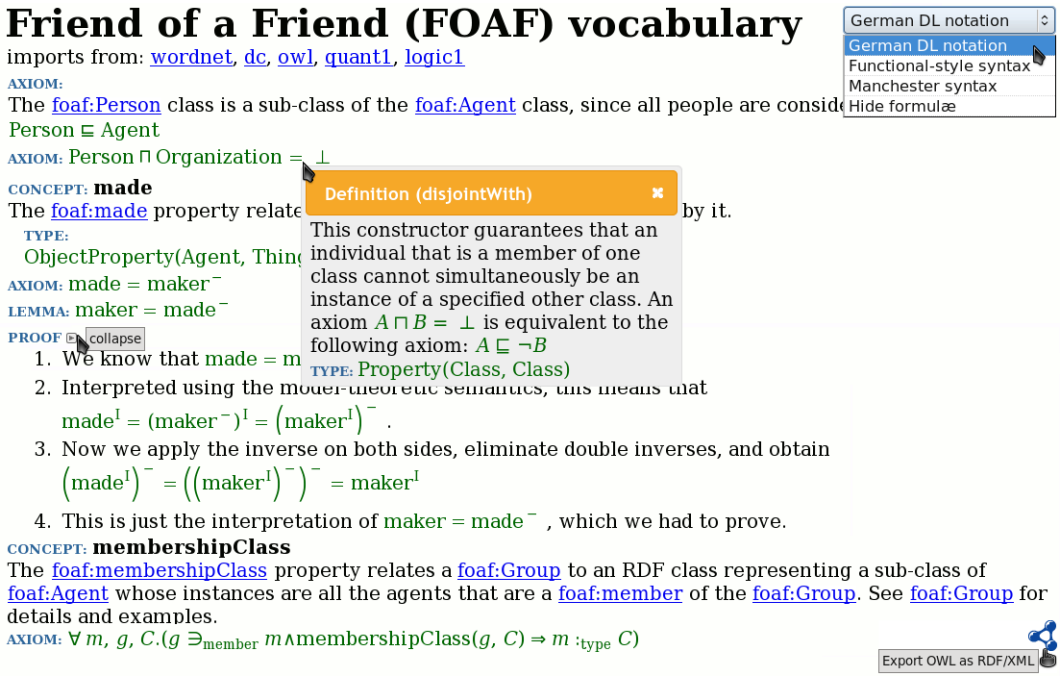

Figure 4: A complex presentation of a documented Ontology

gate fragments of the existing document base and access, edit, and commit them, much like regular documents, if the XML format supports document aggregation in the sense we have defined in this paper. We are also looking into the use of TNTBase(OWL) for ontologies, which is semantically structured document format, but does not (directly) support document aggregation with the intention of developing a suitable generic XML extension that generalizes XInclude [MOV06] to queries.

Even in our limited experience with virtual files, they have turned out to be an enabling technology. Take for instance the example in Section 4.1. There we can use the virtual document in Listing 2 to give students access to fragments of the original XML documents (the problems) while hiding others (the solutions[5]). Here, virtual documents promise to enable (some) fine-grained access control in TNTBase; we plan to study the consequences of this idea in the near future.

The next larger step in the development of TNTBase will be the introduction of distribution facilities for versioned XML document storage supporting both push and pull-based workflows. We hope to gain not only distributed document management functionalities for TNTBase, but also to offer offline capabilities for web applications, which can then simply integrate the TNTBase library for transparent caching. In such applications, the TNTBase content would take the function of a subversion working copy with the additional ability of offline commits.

References

[BBC+07] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path Language (XPath) 2.0. W3C recommendation, World Wide Web Consortium (W3C), January 2007.

[Ber] Berkeley DB XML.

[5]Assuming of course that we have configured TNTBase's file-level authentication and authorization to restrict access to the course materials to instructors.

[GLR09] Jana Giceva, Christoph Lange, and Florian Rabe. Integrating web services into active mathematical documents. In Jacques Carette, Lucas Dixon, Claudio Sacerdoti Coen, and Stephen M. Watt, editors, *MKM/Calculemus 2009 Proceedings*, number 5625 in LNAI, pages 279–293. Springer Verlag, 2009.

[KGLZ09] Michael Kohlhase, Jana Giceva, Christoph Lange, and Vyacheslav Zholudev. Jobad – interactive mathematical documents. In Brigitte Endres-Niggemeyer, Valentin Zacharias, and Pascal Hitzler, editors, *AI Mashup Challenge 2009, KI Conference*, 2009.

[Koh06] Michael Kohlhase. OMDoc – *An open markup format for mathematical documents [Version 1.2]*. Number 4180 in LNAI. Springer Verlag, 2006.

[Lan08] Christoph Lange. SWiM – a semantic wiki for mathematical knowledge management. In Sean Bechhofer, Manfred Hauswirth, Jörg Hoffmann, and Manolis Koubarakis, editors, *ESWC*, volume 5021 of *Lecture Notes in Computer Science*, pages 832–837. Springer, 2008.

[Lan10] Christoph Lange. *Semantic Web Collaboration on Semiformal Mathematical Knowledge*. PhD thesis, Jacobs University Bremen, 2010. submission expected in January 2010.

[LAT] Latin: Logic atlas and integrator.

[MOV06] Jonathan Marsh, David Orchard, and Daniel Veillard. XML inclusions (XInclude) version 1.0 (second edition). W3C Recommendation, World Wide Web Consortium (W3C), November 2006.

[MVW05] Jonathan Marsh, Daniel Veillard, and Norman Walsh. `xml:id` version 1.0. W3C recommendation, World Wide Web Consortium (W3C), September 2005.

[PCSF08] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control With Subversion*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2 edition, 2008.

[RK09] Florian Rabe and Michael Kohlhase. A web-scalable module system for mathematical theories. Manuscript, to be submitted to the Journal of Symbolic Computation, 2009.

[sch] schematron.

[TNT] Tntbase trac.

[WM08] Norman Walsh and Leonard Muellner. *DocBook 5.0: The Definitive Guide*. O'Reilly, 2008.

[XQua] XQuery: An XML Query Language.

[XQUb] XQUpdate: XQuery Update Facility 1.0.

[ZK09] Vyacheslav Zholudev and Michael Kohlhase. TNTBase: a versioned storage for XML. In *Proceedings of Balisage: The Markup Conference 2009*, volume 3 of *Balisage Series on Markup Technologies*. Mulberry Technologies, Inc., 2009.

[ZKR] Vyacheslav Zholudev, Michael Kohlhase, and Florian Rabe. A [insert xml format] database for [insert cool application] (extended version).