A Wiki-based System for Schema and Data Evolution

Lorenzo Bossi, Dept. of Biology, Informatics and Communication (BI-COM) – Insubria University <lorenzo.bossi@uninsubria.it>

Alberto Trombetta, Dept. of Biology, Informatics and Communication (BI-COM) – Insubria University <alberto.trombetta@uninsubria.it>

Abstract

The community of users of a large data-driven web site may directly contribute to its management by feeding corrections and new additions, thus keeping “fresh” the information provided by the site. However, several issues may arise due to the fact that users may modify data in a more or less controlled way. Starting from a real-world scenario, we point out such issues and we present a simple and efficient framework. The proposed solution has been implemented in a XML-based prototype framework, that have been tested with large, real-world datasets.

Table of Contents

Introduction ................................................................. 1
  Our motivating scenario .................................................. 2
Related Works ............................................................... 3
Documents and Templates .................................................. 3
  Our scenario, continued .................................................... 5
  Validation of documents .................................................. 5
  Evolution of templates and documents ................................ 5
  Revision control support .................................................. 7
Our framework ..................................................................... 9
Experimental results ........................................................ 9
Conclusions ....................................................................... 10
Bibliography ..................................................................... 11

Introduction

Large electronic commerce sites allow users to actively participate in collecting feedback information about items and products available on such sites. Typically, users may add comments and rank or tag items according to their preferences. Typically, the information provided by e-commerce sites’ users is used for providing tailored recommendations to them and as the functionalities offered by to the users for adding and manipulating information are very limited: users typically can only add comments to the items and rate them, according to some ranking criteria. Following the recent trend of community-based information management [DRC06, Doa07, BCLM11] – according to which the information may be directly manipulated by end users – we propose to augment the “expressive power” of tools users employ in organizing and managing the semistructured (XML-based) information they provide about items, in order to allow an expressive and useful structuring of the information itself. As such, users may add and structure information about items (or even add new items) of their interest. We adopt an approach inspired by wikis, in which users may structure the data they provide in complex ways and manage it in a collaborative way.

Our application scenario (see the section called “Our motivating scenario”) deals with a vast number of XML documents containing information about items as shown on a large e-commerce site, and since such
information is not completely unstructured, our framework assumes that each document may (possibly in a loose way) adhere to one of a relatively small number of schemas (thereon called templates).

Henceforth, users may create and modify both documents and their corresponding templates. Furthermore users may interact by modifying documents and templates created by other users as well, thus adding and modifying information in a collaborative way (of course, such interactions have to be regulated by proper access policies and trust/reputation mechanisms stating which users may modify which data. In the present work we do not deal with such issues). Having the users such possibilities entails that our system has to take into account the interplay occurring among a modified template and the corresponding (unmodified) documents. More precisely the system, enforcing the constraints about the document structure, supports users in finding ill-formed documents by allowing them to improve their content.

While supporting a community-based approach for updating documents and templates offers the advantage in keeping information up to date (provided by users’ feedbacks), such approach poses several non-trivial questions about the correct management of such information. In particular, (i) when and how updates on templates are reflected on the corresponding documents? (ii) How to manage the rollbacks of unwanted (possibly malicious) updates without erasing subsequent licit ones? And how such rollbacks on templates affect the corresponding documents?

This may be non-trivial tasks, as we will argue in the following. As for point (i) above, the problems we incur in dealing with a community-inspired data management approach come from the existence of integrity constraints that are expressed in a template and that the corresponding documents have to satisfy; whereas for point (ii), the difficulties arise in guaranteeing the maximum possible number of updates without losing validity of documents with respect of their templates.

In fact classical wikis handle structured data through the use of infoboxes [INFWK], but they don’t support any kind of constraint enforcement which can guarantee the uniformity of pages which use them.

The framework we present in this work is based on XML Schema [TMBM04, MB04] for the definition of the basic structure of templates, on Schematron [Jel00] for the definition of more complex integrity constraints and on a XQuery Update-like language for the query language. As it is well known, updating XML documents in a consistent way is a far from trivial task [CRF09] and our work can be summarized as follows.

**Our Contributions.** In this paper, we focus on the following main topics:

i. describe the prototype for a wiki focused on store and manage semistructured data,

ii. define ad-hoc methods for automatic documents evolution upon template change,

iii. propose a new kind of revision control system that is centered on how to minimize loss of new data and maximize data coherence in case of template rollback,

iv. define a simple, ad-hoc, XML-based data model to store data and an XQuery like update language for it.

We remark that we do not address relevant issues concerning access control policies to documents and templates (and the corresponding enforcement). We are aware of the paramount relevant of such aspect and we plan to work on them in the near future.

**Our motivating scenario**

As a real-world example of a large repository of relatively small and relatively uniform documents, we consider the price comparison service http://www.shoppydoo.it. Such site holds the large majority of the market share in Italy with more than 2 million users and it has a very significant presence in other European
countries (e.g. Spain, France, Germany, Netherlands) and non-european as well (e.g. Brazil). More details on data volumes are detailed in the section called “Experimental results”.

The site stores information about more than 1 million items, grouped in roughly one hundred categories. Items are described in pages containing their technical details. Such information is displayed in concise and tabular form for letting users quickly find and compare items.

For example, the information about technical details about digital camera must specify brand and model, as well as camera resolution and memory support. Other less relevant – but still useful information – may comprise the presence of features like an image stabilizer or a face detector, etc.

The users of such site form an online community that may create, update and share information about items by interacting with the site itself. At the present time, the site does not allow its users to actively participate in the management of the displayed information. Our long-term goal is to provide community-based information capabilities to a large, e-commerce-based web site.

Thus, for example, in the case that the user Alice notices that the page describing her preferred camera reports incorrect information about its resolution, she can correct it. Further, Bob is a more active user and notes that almost every digital camera is able to connect to a PC and thus explicitly specifying such information is useless. As such, he decides to modify the digital camera template in order to remove such information from every corresponding document.

In what follows we present methods and techniques that allow users to directly manage such updates and to control the interplay between documents and templates. We will present in a more detailed way the actions performed by such users, as we unfold our motivating scenario in the following sections.

**Related Works**

To the best of our knowledge, the most mature work closest to ours is present in [BCLM11] in which the authors describe a prototype for a wiki for structured data. The main difference between our project consist that they manage only one (usually fairly big) XML document. It turns out that the considered schemas are simpler than ours: for example, they do not allow to specify the type of the data but only the tree structure. Rather, the author focus on query language issues. They are developing a powerful query language which let to select only a fragment of the XML based on various constraints which can involve also annotation on nodes. Finally they support only two types of schema updates: insertion that happen automatically when inserting a data which require a schema extension and deletion which delete also the corresponding data subtree.

Other works deal only with the schema evolution. We think that the most important for our context are [Kle07] in which the authors propose a conceptual model for XML Schema evolution. They use a graphical environment to define schemas and schemas update. Then some normalizations are performed on updates to minimize their. After schema update, they recheck document validity and perform a document update. But there’s no way to update node values on documents. Another interesting work is [GMR05], where the authors define a set of update primitives for XML Schema. They study which evolution primitives are known not to compromise documents validity. Then, they use a labeling process to keep track of the document portions whose validity might have been compromised so they can revalidate only subtree to speedup the process. Their approach to documents evolution consist on the detection of the minimal modifications required to make the documents valid for the evolved schema. But only document structure can evolve, not document data.

**Documents and Templates**

We customarily represent a document as an XML tree. For example, a document containing information about a given digital camera may be structured in the following – rather conventional – way: the camera’s
model is stored in an alphanumeric string, its megapixel capacity is an integer number, the supported memory is a single value chosen from a set of alphanumeric strings. Further, we add a section element which is used to group related elements under a section name. Each element name is unique in its section.

Templates are defined in a way similar to what proposed by Examplotron [VDV03]. We have chosen such approach for its ease of use (see the section called “Validation of documents”). Thus, we define a template as an instance of a empty document, where each element has an additional boolean attribute specifying whether it is mandatory or not in the documents.

More formally, a document is composed by data nodes and elements nodes where: a data node contains only a string value; an element node is a tuple associated with a name, a type and a set of children nodes; Elements may have simple or complex types, where complex types are in \{enum, values, section\} and simple types are in \{int, float, str, bool\}. The type of an element node defines restrictions on the set of children nodes. That is,

i. simple type elements and enum elements children set must contain only a data node;

ii. values elements children set must be a non empty set of distinct data nodes;

iii. section element children sets can be only a non empty list of children elements with distinct names (in this way, elements can be found without ambiguity).

As such, the document is an unordered tree of section and value elements, where the root is a section element.

A template is composed by template elements which are similar to documents’ one. In addition, they have an extra boolean mandatory attribute. Every template has to satisfy the following constraints:

i. template elements of simple type must have an empty set of children;

ii. template elements with type values or enum must have a non empty set of distinct data nodes;

iii. template elements with type section must have a non empty set of children elements with distinct names.

As for documents, a template is an unordered tree where root element is a template section element.

A document is valid, of course we assume the well-formedness, if all data nodes contain values that:

i. an integer, if the type of parent node is int;

ii. a float, if the type of parent node is float;

iii. a boolean, if the type of parent node is bool;

iv. a non empty string, if the type of parent node is str.

A document is valid with respect to a template if and only if:

i. it is valid (see above);

ii. all document’s elements are also present in the template with the same name and the same type;

iii. all template’s elements with mandatory set has a corresponding element in the document;

iv. the children of enum, values and section elements of the document are also children of the corresponding template’s elements.
Our scenario, continued

As one may suspect, documents and templates are stored in XML files. Document elements are serialized in XML nodes where the tag name defines the node type and the name is stored in an attribute. For example an element *Model* of type *string* is serialized as `<str name="Model">Z80</str>`. Templates enforce the information type of complex element, in such a way that the *Viewfinder* can be one of *optical* or *LCD*, while *Extra feature* must be one or more between a list of valid values. Templates XMLs are very similar to documents’ one, the main difference is that simple elements are empty and that every element has an attribute *mandatory* which contain a boolean value. For example the element *Model* in a template is `<str name="Model" mandatory="true"/>

Validation of documents

We use XML Schema [MB04, TMBM04] to validate documents and templates as serialized XML documents. This first step validates the overall structure of XML documents. Regarding documents, XML Schema is used to check that they are structured in sections containing the named values and the correct types of simple elements’ values. Regarding templates, XML Schema is used to check sections, the uniqueness of name into their sections and the presence of a mandatory boolean attribute for each element.

As said before, users specify templates as an empty document. As such, in order to validate a document with respect to some template, templates themselves have to be rewritten in some suitable XML schema language.

We use Schematron [Jel00] since its assertion rule validation style makes error reporting clearer and the usage of XPath constraints allows the definition of constraints over unordered sets of context-dependent elements. Furthermore, Schematron checks the presence of mandatory elements, the absence of illegal elements and the correctness of *values* and *enum* children elements. The conversion from a template to its corresponding Schematron schema is performed by an XSLT stylesheet [Cla99].

As already pointed out, the main advantage in writing templates in the above presented XML format is its compact and easily readable syntax, that can be promptly deployed by the community users.

Evolution of templates and documents

Returning to our motivating scenario, since Bob reputes very important to know if a digital camera is able to record videos, he adds a new boolean mandatory field called *video recording* in the camera template. We note that this kind of update invalidates all the documents associated to the corresponding template. We patch this problem adding a special page showing all invalid documents to let active users of the community perform updates on such documents, to restore their validity again.

Interacting with templates and documents

Community users can read, create and modify documents and templates. After a document is updated, the following steps are performed: (i) check whether the document is valid according to the corresponding XML Schema, if this is not the case, reject the update; (ii) otherwise get the associated template and translate it into a Schematron document; (iii) validate the document with the corresponding Schematron to check if it complies with the template and return the validation results.

A template update can (i) leave all associated documents valid (for example, the insertion of a new value in an enumeration); (ii) invalidate all associated documents (for example, adding a mandatory element in a mandatory section); (iii) require a necessary update and consequent re-checking of all associated documents (for example, deleting an element); or, finally, (iv) leave the documents in an unpredictable
state, regarding their validity (for example, an optional value becomes mandatory). In this case, the only way to discern the documents’ validity is to re-check them all. Since cases (iii) and (iv) are similar, because (iv) is like (iii) with an empty update, we treat them in the same way.

The update language we propose allows community users to create and update elements’ types, their name, mandatory fields, add, modify and delete elements.

Since the community users may perform updates on templates and documents defined by the previously defined data model, we do not need the full expressive power of XQuery Update Facility [CRF09] and, thus, our update language is basically a simplified version of XQuery Update. First, we define the element selector as a string that allows to find an element in unambiguous way. It is formed by the element name preceded by all sections name separated by the slash sign (e.g. `/Digital camera/Brand`). We also define a data selector as the selector of its container followed by a slash sign followed by the data text value wrapped by brackets (e.g. `/Digital camera/Supported memory/`[Compact-Flash]). Those selectors can be easily translated into XPath [BBC10] expressions. The latest example in XPath is written `/*[@name='Digital camera']/[@name='Supported memory']/[text()='CompactFlash']`.

### Evolution of documents

In this section we describe our document update language and how each command can be translated in an XQuery Update statement.

Every command that we had defined take a selector as parameter. Whether it is an element selector or a data selector is first transformed into an XPath selector while the command is recognized and translated into a valid XQuery Update statement.

A user may delete either a data or element node with the command `delete node <selector>`. In this case we need only to translate the selector to have a valid update.

Add new data into a document let a user to fill values list. This operation is performed by the command `insert data(text) into <elementSelector>` which is converted into `insert node <value>text</value> into <xpathSelector>`.

Since the most of elements node need a data child to be valid (all simple and `enum` types) we provide the command `insert node(type, name, isMandatory) into <elementSelector>` to insert both of them. When translated into XQuery Update, such command becomes `insert <type name='name' mandatory='isMandatory'/> into <xpathSelector>`.

The last command we describe is for replacing the value of a data node. Its syntax is `replace value of <selector> with <newval>` and is translated into XQuery Update in replace `replace <xpathSelector>/text() with 'newval'`.

### Evolution of templates

Template evolution is more difficult because for every defined command we need not only to update – of course – the template but also to decide whether the documents associated have to be modified as well and, in the affirmative case, perform such updates. It is important to note that our node selectors (and the translated XPath equivalents) are valid both on documents and templates.

To add a new field on template specify we should specify the name, the type and if this information is to be mandatory. This operation is performed with the instruction `insert node(type, elName, isMandatory) into <elementSelector>`, which we translate it into XQuery Update with insert `<type name='elName' mandatory='isMandatory'/>` into `<xpathSelector>`. If the
inserted node is not mandatory, after this update documents are still valid. If the inserted node is mandatory
and all ancestors sections are mandatory too, after this update all associated documents are no more valid.
Otherwise all associated document must be re-checked to define if they are still valid.

Another useful command is similar to the last one but lets user to specify a default value for the new
elements. The template update is equal to the last one, but in this case we have also the following document
update insert <type name='elName'>defaultVal</type> into <xpathSelector>.

New data nodes can be added to an enum or values element with insert data(val) into <elementSelector>. This update preserves validity of documents and can be written in XQuery Update
as insert <value>val</value> into <xpathSelector>.

The command change type <elementSelector> with newType is useful to change the type
of an element. The template and document XQuery Update is rename node <xpathSelector>
as newType. Is important to note that, depending on the update and on the old data, document validity
could be preserved after this update.

In a similar way, to change the name of an element we use changename <elementSelector>
with newName and we apply to the template and the corresponding documents the instruction replace
<elementSelector>/@name with newName.

The command changemandatory <selector> with (true|false) is used to change the
mandatory value of an element. It is translated in replace <elementSelector>/@mandatory
with (true|false). After this operation, if the updated element is not mandatory all documents
are still valid. If it is mandatory, as well as all the section ancestors, all documents is marked as invalid.
Otherwise, all documents have to be revalidated.

The simplest operation is the deletion of a node, that is performed in the same way both on templates and
documents by the XQuery delete node <xpathSelector>.

The last operation is replace value of <selector> with newVal that uses XQuery functions
to compute newVal. This operation is used to perform an update to a field on all documents associated
to the current template.

Is important to note that particular updates sequence can leave the system in an inconsistent state. For
example we have a template with a not mandatory element, then some documents will contain this element
while some other will not. All documents and templates are valid. This is the initial state. A user decide that
the element must be mandatory and update the template. As described previous, after the template update,
all documents are revalidate. So we have some valid documents (those have the mandatory elements) and
some invalid ones (all the other). In this state another user see the new template and change back the same
element to be optional. An unskilled or heedless user can perform this operation as a normal update, instead
using the correct feature of the versioning system. So the template is updated however no operation on
document is performed because this kind of update don’t compromise their validity. But all documents that
was marked as invalid is now valid again and in this final state we have all templates and documents valid
but with some document marked as invalid. Therefore we need also a process scheduled for re-checking
at regular intervals the validity of documents to achieve an eventual consistency of data.

Revision control support

Revision control is a very relevant feature for wikis. It is useful for monitoring a page evolution and to
deal with modifications performed by malicious users.

Implementing a revision control system for our wiki-based repository is not trivial, since documents are
represented as XML trees with complex constraints occurring among their elements, as specified by their
corresponding templates.
In order to illustrate the problems in building a revision control system fulfilling the above mentioned conditions, we consider the scenario in which a user wishes to undo a template update, but – in the meanwhile, after the template update – the documents associated to such template have been modified, so they are no more valid with the old template.

In this scenario we can operate in two different and completely opposite way: (1) we can revert all documents to their valid versions with the old template or (2) we can leave all documents to the last revision and revert only the template. Both solutions have pros and cons: The former maintains consistency between documents and templates but it can potentially loose many useful document updates. The latter does not loose document updates, but it can potentially leave all documents in an invalid state.

There is no single best solution to the problem of how to retain document updates while satisfying template consistency too, since it may depend very heavily from the context. For example, if one wishes to undo revert a malicious template update (possibly the outcome of a deliberate act of vandalism), it is pointless to save the corresponding document updates because documents contain information corrupted by the vandalism act, as well. In this case, Solution (1) appears to be the best fit. On the other side, inconsistencies between a template and the corresponding documents arising from a minor change in the template structure (e.g., an element may become optional) seem to be viably managed by Solution (2).

Given the extreme sensitiveness to the context of the chosen solution, our aim is to define helpful techniques and tools that support the user in adopting the best possible solution that fits its needs.

In particular, we define a hierarchy of possible solutions formed by the two scenarios introduced above, plus other two intermediate levels that offer a sort of “interpolation” between such two extremes. In more detail, when a user has to “revert” a template to a previous version, we can choose to:

1. revert all the corresponding documents to their previous versions, in agreement with the reverted template;
2. revert only the structure (as dictated by the reverted template) of the documents but not the content;
3. revert only the document involved by the template update;
4. leave the document unmodified.

The following examples show a real-world scenario the four options just introduced.

As a deliberate act of vandalism, consider the following: an user deletes a large part of a template and renames the elements of the remaining part with unrelated content. A document update performed with the new template, can only try to limit the damages, but can not improve the document quality. In this case the best solution is ignore the document updates and revert documents as they were before the template update.

As for the second option, consider a car radio template, describing its features. In the Car Radio template, a user rename the Audio section in Speaker system. The semantic of the section does not change. So every document update is legit. But all the other templates which describe consumer electronics stuffs contain an Audio section. So, to uniform the templates, is better revert this update undoing the rename but without loose any update to the documents (Level 2).

Consider a Notebook template which contains a Memory and a HD sections. Both of them contains two integer Total size and Max size. An unskilled user, who does not know the difference between RAM memory and Hard Disk, can rename the Memory section in Disk sizes section hoping to make the template better. This update changes the semantic of the section. So every document update which involve this section is compromised. When someone reverts the template update, the best solution should revert all documents updates which involved this section, but should keep all the others (Level 3).

Suppose that someone add a new data Autofocus under the multi-values Extra features in a camera template. This is a useful information, but it is redundant since there is an optional boolean field Auto focus
in the section *Lens system* yet. In this case document updates happen after the template update contains useful information but in the wrong place. The better solution consist to revert the template but not the documents so no information is loose (Level 4). In this way we can encourage the community to correct the documents moving the information in the right place.

**Our framework**

Our framework, as seen in Figure 1, “Framework schema”, is structured in four components: the XML repository, the query/update processor, the document and template validity checker and the user interface.

**Figure 1. Framework schema**

For the XML repository we need a system that can efficiently handle many XML files and that supports XQuery Update. We choose eXist [EXDB] because, as of now, is the most standards-compliant and extensible among free native XML databases. This component provides the persistence of data, the XQuery Update engine and the XML validation.

The query processor component takes as input the user queries and translates them into XQuery Update. It communicates with the database by sending updates, asking for validations and performs commit or rollback depending on validation results. Algorithms described in the section called “Documents and Templates” are implemented in this component.

The user interface provide a set of features which let the user to intuitively interact with the prototype without the requirement of a formal training. Since the purpose of the prototype is to build a collaborative system, the user interface is a web application. It lets the user to (i) easily find documents and templates based on different criteria, (ii) read a rendered version of templates and documents and (iii) edit templates and documents in an interactive way.

**Experimental results**

Since the relatively small size of documents and templates and since updates and validations are performed on a native XML database, we can safely assume that a single update operation is performed in a relatively short, constant time roughly equal to 10 ms.

Thus, the performance issues of our framework depends on the number of documents that have to be checked, depending on the kind of update operation that has been issued.
All tests have been performed on a PC with an Intel® Core™ 2 Duo P8700 CPU and 4 GB of RAM running Windows Vista™. The framework is implemented in C# 3.0 and deploys eXist 1.4 [EXDB] as the XML native database.

As already mentioned in the introduction, the dataset used in the experiments come from the ShoppyDoo online price comparison service which is visited by more than two million of unique user per month and compare prices of four million offers from fifteen hundred merchants. Every document describes the technical details of a single product and there is one template for product category. In total, our dataset contains 9605 documents and 72 templates, totaling about 65 MB of XML files. The less populated category (Digital photo frames) contains only 3 documents, while the most populated one (LCD, LED and plasma TVs) contains 798 documents. We want remark that the current dump of Wikispecies [WIKISPC] is approximately 370 MB of XML data. It’s interesting for us show that our test dataset is big about the 18% of a small Wikimedia Foundation project.

As explained in the section called “Interacting with templates and documents”, a template update can fall into three different categories, depending upon its impact on the associated documents: it can leave them all valid, it can invalidate all of them or it can make necessary to update and recheck all of them. As such, we performed editing templates tests with different quantities of documents and with the three different kinds of update. The results are shown in Figure 2, “Template update performance test”.

The first type of update is very fast and it is not affected by how many documents are associated to the modified template. Since the algorithm needs only to update one (typically) small XML document (the template itself) it executes in about 100 milliseconds.

We point out the in our current prototype, all documents have a metadata field that records their validity with respect to their templates. In this way, the second type of update performs in a time linear in the number of documents. Since updating the validity to given value is very fast, all our tests ended within 150 milliseconds.

The last type of update is the worst because, after the template update, the prototype has to perform an update to all the documents and they all have to be revalidated. The time is linear to the number of documents, and since validation process is slower than the update execution, the worst case takes about two minutes to execute.

**Figure 2. Template update performance test**

![Template update performance test](image)

**Conclusions**

Although our framework is still under development, the first experimental results show that updates of templates associated with small – yet significative – sets of documents execute in reasonable time; still, there is ample space for optimizing the proposed procedures. The optimization process may involve as well a phase of logical redesign of large XML schemas into smaller, more manageable ones. Along with
optimization issues, we are dealing with the realization of a suitable web application that allows users to easily interact with the repository.

Finally, the other major topic deserving further investigation is – of course – how to regulate the access of users to data, allowing them to modify items created by other users. To this end, we are investigating the integration of classical access control techniques with incentive-based mechanisms borrowed from reputation systems, in order to elicit a collaborative behaviour from users but keeping an eye – at the same time – on what they can do.

Bibliography


