XML Document Versioning and Revalidation*

Jakub Malý, Jakub Klímek, Irena Mlýnková, and Martin Nečaský

XML Research Group, Department of Software Engineering
Faculty of Mathematics and Physics, Charles University in Prague
Malostranské náměstí 25, 118 00 Praha 1
The Czech Republic
{klimek,maly,mlynkova,necasky}@ksi.mff.cuni.cz

Abstract. One of the prominent characteristics of XML applications is their dynamic nature. When a system grows and evolves, old user requirements change and/or new requirements accumulate. Apart from changes in the interface, it is also necessary to modify the existing documents with each new version, so they are valid against the new specification. The approach presented in this work extends an existing conceptual model with the support for multiple versions of the model. Thanks to this extension, it is possible to define a set of changes between two versions of a schema. This work contains an outline of an algorithm that compares two versions of a schema and produces a revalidation script in XSL.

Keywords: XML, schema, schema evolution

1 Introduction

Recently, XML [12] has become a corner stone of many information systems. It is a de facto standard for data exchange (i.e. Web services [4]) and it is also a popular data model in databases [3]. XML schemas are utilized in two scenarios: 1) to describe structure and check validity of internal documents, and 2) to define the interface of a component of the system for other components or of the system to the outer world.

Requirements change during the life cycle of the system and so do the XML schemas. Without any tools to help, the old and new schema need to be examined by a domain expert. Each change must be identified, analyzed and all the relevant parts of the system modified and the existing documents updated. This can be a time-consuming and error-prone process, but, in fact, a significant portion of the operations could be performed automatically.

The existing approaches work directly at the level of XML schemas, which leads to problems with recognizing the semantics of each change. Our approach utilizes the conceptual model for understanding the semantics. Also, most of the existing approaches require the user to either revalidate the documents after

* This work was supported in part by the Czech Science Foundation (GAČR), grant number P202/10/0573, and by the grant SVV-2011-263312.

every performed evolution operation, other require at least all the operations to be performed in one session and directly modify the old version (effectively replacing it with the new version). Our approach imposes no limits on the amount of operations performed between the versions and the amount of versions present in the systems.

The main aim of this paper is to simplify the whole process of system evolution and make the transition to the new version semi-automatic using our conceptual XML modeling framework with the support for multiple versions. We carry on by formally defining possible changes between versions. When changes are detected, we use this information to decide, whether the revalidation is necessary and if it is the case, we generate a script that revalidates existing XML documents against the new version.

1.1 Outline

The rest of this paper is organized as follows: in Section 2, we analyze the existing approaches. In Section 3, we introduce the XSEM conceptual model. In Section 4, we extend the model with versioning support. In Section 5, we formally define changes between versions. In Section 6, we outline how a revalidation script is generated. Section 7 concludes and lists the open problems and areas of future research.

2 Related Work

Approaches can be categorized on the basis of the level where the changes are made by the designer and detected by the algorithm as suggested in [10]:

- in XML schemas (in one of the XML schema languages), so-called *logical* level. At the logical level, changes are detected directly in the evolved schema.
- in diagrams visualizing the XML schema (diagrams directly visualizing constructs of a specific XML schema language).
- in a UML diagram used to model an XML format (usually annotated using comments or stereotypes to specify the translation to an XML schema).

X-Evolution, proposed in [5], is an example of a system built upon a graphical editor for creating schemas in the XML Schema language [14]. At first, the authors defined a set of *evolution primitives*, i.e. basic operations that modify the evolved schema (e.g. *insert_local_element*, *remove_type*). Impact of each primitive on the validity of the existing XML documents is examined. The presented *Adapt* algorithm takes as an input an evolution primitive, an XML schema and an XML document valid against the schema and returns an evolved document valid against the schema on which the evolution primitive is applied.

X-Evolution system provides the user with a narrow set of evolution primitives that can be performed upon XML schemas of a certain format. There are no links to the conceptual model which could be utilized when new content needs to be added during document revalidation; thus, it is up to the user to write the adaptation clauses or update the documents after revalidation. Some very common real-world evolution operations are not considered in the evolution primitives, e.g. moving content, adding a wrapping element for elements or transforming attributes to subelements and vice versa. All algorithms and operations expect a single evolution primitive as an input. And only a single change is supported in each evolution cycle, evolution with multiple changes is not elaborated.

XML Evolution Management, XEM [6] is an approach to manage schema evolution where DTD is used as a schema language. It deals both with changes in DTD and XML documents (*instance documents*).

The proposed primitives are divided into two categories: 1) changes in DTD 2) changes in instance documents. It is proven that the proposed set of primitives is complete, in the sense that any possible change in DTD can be expressed as a sequence of application of the primitives. However, the method used in the proof in many common cases degenerates in removing a large part of the tree and creating it from scratch (e.g. when an element is renamed, it must be removed and then added under new name; if the root element is removed, the whole XML document is first deleted and its structure recreated again). In this process, the structure is created properly by the algorithm, but the data is lost. Again, as in the case of *X-Evolution*, the conceptual model is not considered and thus cannot be utilized when new content is added in the instance documents.

Unlike the previous approaches, *CoDEX*, Conceptual Design and Evolution of XML schemas [7], allows the user to perform a sequence of evolution operation before the documents are revalidated. Each user's single action is recorded by the logging component. When the user is satisfied with the new version of the model, the recorded design steps are minimized and normalized. The proposed conceptual model is closer to a visualization of the XML Schema language than to a platform-independent model and the lack of a conceptual model is directly responsible for problems declared as in-built (e.g. the algorithm can not detect such a change where the same structural element acquires different semantics).

3 XSEM Conceptual Model

Conceptual modeling is a top-down approach to system and data design, which concentrates on correct depicting the problem domain – defining the particular concepts comprising the system and relationships between these concepts.

XSEM is an approach to modeling XML data using *Model Driven Architecture* (MDA) [8]. A prototype implementation of XSEM in a CASE[1] tool called XCase [1] is available. The model has two interconnected layers – platform-independent model (PIM) which utilizes UML [11] class diagrams and platform-specific model (PSM) which again utilizes UML class diagrams but extended with so-called *XSEM profile*. In the rest of this paper, we will use the formal definitions of PIM (Definition 1) and PSM (Definition 2) levels.

[1] Computer-aided software engineering

3.1 PIM Level

A schema in a platform-independent model (PIM) models real-world concepts and relationships between them without any details of their representation in a specific data model (XML in our case). As a PIM, we use the classical model of UML class diagrams in our work. This model is widely supported by the majority of tools for data engineering and there exists the XMI [2] standard for exchanging diagrams between them. Therefore, it is natural to use UML in our approach as well. We introduce PIM schemas formally in Definition 1.

Definition 1. *A* platform-independent schema (PIM schema) *is a triple* $\mathcal{S} = (\mathcal{S}_c, \mathcal{S}_a, \mathcal{S}_r)$ *of disjoint sets of* classes, attributes, *and* associations, *respectively. In addition, PIM schema defines functions* name, card, type, class *and* participant*:*

- Class $C \in \mathcal{S}_c$ *has a name* $name(C)$*.*
- Attribute $A \in \mathcal{S}_a$ *has a name, data type and cardinality* $name(A)$, $type(A)$ *and* $card(A)$, *respectively.* A *is associated with a class from* $\mathcal{S}_c$ $class(A)$*.*
- Association $R \in \mathcal{S}_r$ *is a set* $R = \{E_1, E_2\}$ *where* E_1 *and* E_2 *are called* association ends *of* R*.* R *has a name* $name(R)$*.* $E \in R$ *has a cardinality* $card(E)$ *and is associated with a class from* $\mathcal{S}_c$ $participant(E)$*. For* R, $participant(E_1)$ *and* $participant(E_2)$ *are called* participants *of* R*.*

For simplicity, we will also use the notation $attributes(C)$ *to denote the set* $\{A \in \mathcal{S}_a : class(A) = C\}$ *and* $associations(C)$ *to denote the set* $\{R \in \mathcal{S}_r : (\exists E \in R)(participant(E) = C)\}$ *for each class* $C \in \mathcal{S}_c$*.*

PIM schema components have a usual semantics. A class models a real-world concept. An attribute of that class models a property of the concept. And, an association models a kind of relationships between two concepts modeled by the connected classes. We display each PIM schema as a UML class diagram.

Example 1. Figure 1 shows a sample PIM in XSEM modeling a component accepting purchases. Classes (`Address`, `Purchase`, ...) are depicted as boxes with the list of attributes (`street`, `city`, ...), associations as labeled lines connecting classes (`bill-to`, `ordered`, ...).

3.2 PSM Level

A schema in a platform-specific model (PSM) models how a part of the reality modeled by the PIM schema is represented in a particular data model. In our case, we consider XML as the data model and our PSM schema specifies representation in a particular XML format. Our PSM allows for specifying more different XML formats on the base of the same PIM schema. As a PSM we also use UML class diagrams extended for the purposes of XML data modeling. We introduce PSM formally in Definition 2.

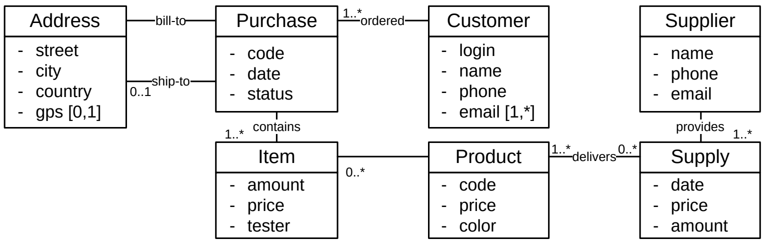


Fig. 1. Sample PIM schema

Definition 2. *A* platform-specific schema (PSM schema) *is a 5-tuple $\mathcal{S}' = (\mathcal{S}'_c, \mathcal{S}'_a, \mathcal{S}'_r, \mathcal{S}'_m, \mathcal{C}'_{\mathcal{S}'})$ of disjoint sets of* classes, attributes, associations, *and* content models, *respectively, and one specific class $\mathcal{C}'_{\mathcal{S}'} \in \mathcal{S}'_c$ called* schema class. *$\mathcal{S}'$ must be a forest with one of its trees rooted in $\mathcal{C}'_{\mathcal{S}'}$. In addition, PSM schema defines functions* name', card', type', xform', cmtype' class' *and* participant'*:*

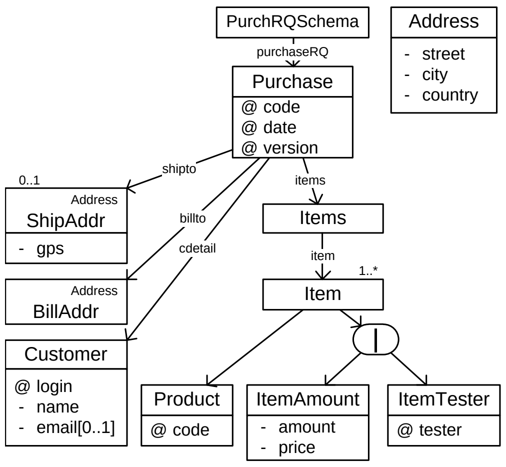
– Class $C' \in \mathcal{S}'_c$ *has a name denoted name$'(C')$.*
– Attribute $A' \in \mathcal{S}'_a$ *has a name, data type, cardinality and XML form denoted name$'(A')$, type$'(A')$, card$'(A')$ and xform$'(A') \in \{e, a\}$, respectively. Moreover, it is associated with a class from $\mathcal{S}'_c$ denoted class$'(A')$ and has a position denoted indexOf(A') within the all attributes associated with class$'(A')$.*
– Association $R' \in \mathcal{S}'_r$ *is a pair $R' = (E'_1, E'_2)$ where E'_1 and E'_2 are called* association ends *of R'. $E' \in R'$ has a cardinality denoted card$'(E')$ and is associated with a class from $\mathcal{S}'_c$ denoted participant$'(E')$. participant$'(E'_1)$ and participant$'(E'_2)$ are called* parent *and* child *of R and denoted parent$'(R')$ and child$'(R')$, respectively. R' has a name denoted name$'(R')$ and has a position denoted indexOf(R') among associations with the same parent.*
– Content model $M' \in \mathcal{S}'_m$ *has a content model type cmtype$'(M') \in \{$`sequence`*,* `choice`*,* `set`$\}$.*

For simplicity, we will also use the notation attributes(C') to denote the sequence $(A'_i \in \mathcal{S}'_a : class(A') = C' \wedge i = indexOf(A'))$ for each class $C \in \mathcal{S}_c$.

Let us denote $\mathcal{N}' = \mathcal{S}'_c \cup \mathcal{S}'_m$. The graph $\widehat{\mathcal{S}'} = (\mathcal{N}', \mathcal{S}'_r)$ with classes and content models as nodes and associations as ordered edges must be a directed forest with one of its trees rooted in the schema class $C'_{\mathcal{S}'}$. Members of $\mathcal{S}'_c$, $\mathcal{S}'_a$, $\mathcal{S}'_r$, and $\mathcal{S}'_m$ are called components *of $\mathcal{S}'$. The set of all PSM schemas will be denoted $\mathcal{S}^{*\prime}$. Finally we define the set of all constructs (PIM and PSM): $\mathcal{M} = \mathcal{S}_c \cup \mathcal{S}_r \cup \mathcal{S}_a \cup (\bigcup_{\mathcal{S}' \in \mathcal{S}^{*\prime}} (\mathcal{S}'_c \cup \mathcal{S}'_r \cup \mathcal{S}'_a))$.*

The PSM constructs in a concrete PSM schema are linked to the PIM schema constructs (we will say that the PSM schema is derived from the PIM schema and the PSM constructs have an interpretation in the PIM). We will not include formal definition of interpretation in this paper and use these terms intuitively instead.

Example 2. Figure 2(a) shows a sample PSM schema modeling an XML document with a purchase. Again, classes, associations and attributes are depicted



(a) PSM schema

```
<purchaseRQ code="#"
  date="#" version="#">
  <shipto gps="#">
    <street>#</street>
    <city>#</city>
    <country>#</country>
  </shipto>
  <billto>
    <street>#</street>
    <city>#</city>
    <country>#</country>
  </billto>
  <cdetail login="#">
    <name>#</name>
  </cdetail>
  <items>
    <item code="#" tester="#"/>
    <item code="#">
      <amount>#</amount>
      <price>#</price>
    </item>
  </items>
</purchaseRQ>
```

(b) XML document

Fig. 2. Sample PSM schema of a purchase order request with a sample XML document

similarly as in PIM schemas, the difference is in associations, which are now directed. The diagram also contains one content model of type `choice` (parent node of classes `ItemAmount` and `ItemTester`). The XML document depicted in Figure 2(b) illustrates how PSM schema models XML documents. Briefly, named associations model element-subelement nesting, unnamed association inline the content to the element above (see association `Item-Product`), associations starting at the schema class model allowed root elements. Attributes model XML attributes or elements with simple content (depending on the value of *xform'* for the attribute, value `a` is depicted by `@` before the name of the attribute). Structural representatives are used for content reuse (see `ShipAddr` and `BillAddr` reusing content of `Address`). The in-depth description of the relation of PSM schemas and modeled documents can be found in [9].

4 Evolution

For the goal of determining whether XML documents were invalidated due to schema evolution, the system must recognize and analyze the differences between the old ($\mathcal{S}'$) and new ($\widetilde{\mathcal{S}'}$) version of the schema. There are two possible approaches: 1) recording the changes as they are conducted during the design process or 2) comparing the two versions of the diagram. Table 1 compares both approaches, for the stated reasons we chose the second approach for our work.

As a motivation for our further progress, take a look at Figure 3. It contains two versions of a fragment of a PSM schema. In the old version, class `Item` has an attribute `price`, in the new version, class `Purchase` has an attribute `price`.

recording changes	**schema comparison**
the recorded set should be normalized to eliminate redundancies (repeated changes in the same place etc.).	No redundancies; the set of changes is always minimal.
Once the evolution process is started, the old version can not be easily changed.	Both versions and new can be edited without limitations.
Possibility to interrupt the work and continue in another session later. The sequence of recorded changes would have to be stored and recording resumed later.	The process of evolution can be arbitrarily stopped and resumed.
To retrieve the sequence for reverse process, the user will have to either start with the new version and record the operations needed to go back to the old version again, or the system will have to be able to create inverse sequence for each sequence of operations.	The reverse operation can be easily handled by the same algorithm, only with the two schemas on the input swapped.
For a schema from an outer source, the sequence of operation changes can not be retrieved directly; the user must start with the old version of the schema and manually adjust it to match the new schema.	A schema from an outer source can be imported[2] and serve as an input to the change detection algorithm.

Table 1. Approaches to Evolution: Recording Changes vs. Schema Comparison

Without any further information, there can be two interpretations of the change: 1) attribute `price` was moved from `Item` to `Purchase` or 2) the attribute was removed from `Item` and a new attribute was added to `Purchase`, coincidentally having the same name. Revalidation of the documents is even more difficult to decide. In the second case, a correct value must be assigned to the new attribute[3]. In the first case, the value of attribute `price` should be set to the sum of the values of `price` in all the items in the first version.

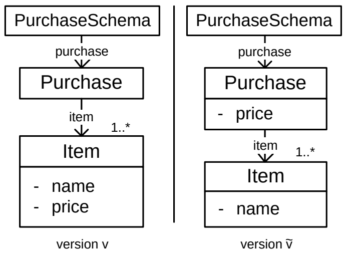


Fig. 3. Evolved PSM Schema

To achieve correct identification of “addition/removal” from “move” (as opposed to existing approaches), we need to extend the XSEM framework with a support for having multiple version in the model and possibility to link constructs in different versions.

Definition 3 (ver, version link, *getInVer*). *Let $\mathcal{V}$ be the set of versions for the model $\mathcal{M}$. Function ver : $\mathcal{M} \cup \mathcal{S}^{*\prime} \rightarrow \mathcal{V}$ returns the version a given construct belongs to. We require a natural condition of all the constructs of a PSM schema $\mathcal{S}'$ to belong to the same version, i.e. $(\forall \mathcal{S}' \in \mathcal{S}^{*\prime})(x \in \mathcal{S}'_{all} \leftrightarrow ver(x) = ver(\mathcal{S}'))$. An equivalence relation of* version links $\mathcal{VL} \subset \mathcal{M} \times \mathcal{M}$

[3] This is an open problem for our future work, see Section 7.

change	**description**
classAdded	new class was added to the schema
classRemoved	class was removed from the schema
classRenamed	existing class was renamed
classMoved	class is moved to the PSM tree
attributeAdded/Removed/Renamed	attribute was added/removed/renamed
attributeTypeChanged	attribute type changed
attributeIndexChanged	attributes were reordered in the class
attributeCardinalityChanged	cardinality of an attribute was changed
attributeMoved	attribute was moved from one class to another
associationAdded/Removed/Renamed	association was added/removed/renamed
associationIndexChanged	associations were reordered in the class
associationCardinalityChanged	cardinality of an attribute was changed
associationMoved	association is moved in the PSM tree

Table 2. Changes Identified in XSEM-Evo between two Versions of the same Schema

contains pairs of constructs that are different versions of the same construct. Partial function getInVer : $(\mathcal{M} \times \mathcal{V}) \rightarrow \mathcal{M}$, *getInVer*$(e, v) = \tilde{e} \leftrightarrow \exists\ \tilde{e} \in \mathcal{M}$: $(e, \tilde{e}) \in \mathcal{VL}\ \wedge\ ver(\tilde{e}) = v$ *returns a construct in a desired version (if* e *does not exist in version* v*, getInVer*(e, v) *is undefined). If* $(x, \tilde{x}) \in \mathcal{VL}$*,* x *and* $\tilde{x}$ *must both be constructs of the same kind (e.g. both classes, attributes, PSM associations...).*

In the following text, we will assume $|\mathcal{V}| = 2$, unless explicitly stated otherwise (i.e. we expect that there are two versions in the system - an old version ($v \in \mathcal{V}$) and a new version ($\tilde{v} \in \mathcal{V}$)). Values of *ver* function form the input of the change detection algorithm (and so does the relation $\mathcal{VL}$). In XCase, values of *ver* are assigned automatically by the system during and maintained As the user edits the schema (either the old or the new version).

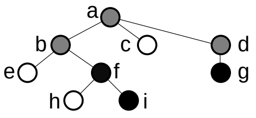
5 Changes

Based on Definitions 1 and 2, we can identify all possible changes that can occur between two versions. Each change is formally defined as a predicate with certain amount of parameters characterizing the change. Detecting changes means finding the constructs that match each respective predicate. Table 2 lists all recognized changes.

Describing each change and it's revalidation is beyond the scope of this paper. Therefore we will describe only *attributeMoved* change in detail (an example of this change is depictend in Figure 3), other changes are treated in a similar manner. Change *attributeMoved* means that the value of *class'*(A') is changed i.e. an attribute is moved from class C'_o to another PSM class $\widetilde{C'_n}$ (and the new index of $\tilde{A}'$ is $\tilde{i}'$). Formally *attributeMoved* is defined as a ternary predicate:

$$\begin{aligned} &\mathit{attributeMoved}(\tilde{A}', \widetilde{C'_n}, \tilde{i}') \leftrightarrow \tilde{A}' \in \widetilde{\mathcal{S}'_a} \wedge \widetilde{C'_n} \in \widetilde{\mathcal{S}'_c} \wedge \tilde{i}' \in \mathbb{N}_0 \wedge \mathit{getInVer}(\tilde{A}', v) \neq \mathit{null} \wedge \\ &\quad \mathit{class'}(\tilde{A}') = \widetilde{C'_n} \wedge (\exists C'_o \in \mathcal{S}'_c)(\mathit{class'}(\mathit{getInVer}(\tilde{A}', v)) = C'_o \wedge \mathit{getInVer}(\widetilde{C'_n}, v) \neq C'_o) \end{aligned}$$

Moving an attribute is an evolution operation that requires more in-depth enquiry. The aim of our approach is to keep the semantics of the revalidated document and not to lose the existing data during revalidation. The trivial solution – deleting the attribute from its former location in the document and creating a new attribute in the new location (as used in [6] and [5]) – is not suitable, because the value of the attribute is lost. The most general approach is to couple each instance $(\tilde{A}', \widetilde{C'_n}, \tilde{i}')$ of *attributeMoved* change with a revalidation function $attMove_{\tilde{A}'}(oldLocations, newLocation)$: $XSEMPath \times XSEMPath \rightarrow type'(\tilde{A}')$[4] where *oldLocations* is an expression returning all existing instances of A' and *newLocation* is an expression containing the path to one instance in the new document. The result of the function is the new value for the instance of $\tilde{A}'$ in the new document. In the general case the function $attMove_{\tilde{A}'}$ is defined by the user, but the system can give the user a suggestion in certain cases – several types of the most common scenarios can be distinguished.



The figure on the left shows an example of *tree* function, the result of $tree(f, g, i)$ is the set $\{a, b, d, f, g, i, a-b, b-f, f-i, a-d, d-g\}$, a is the root of the common subtree and $a - b$ the edge from a to b.

Fig. 4. Example for *tree* function

In the following text we expect that the attribute was moved between classes C'_o and $\widetilde{C'_n}$, $A' \in attributes'(C'_o)$, $\tilde{A}' \in attributes'(\widetilde{C'_n})$. Let $\widetilde{C'_o} = getInVer(C'_o, \tilde{v})$ and $C'_n = getInVer(\widetilde{C'_n}, v)$, be the new version of class C'_o and the old version of class $\widetilde{C'_n}$ respectively, both can be *null*. In the situation depicted in Figure 3 $A' = \texttt{price}_1, \tilde{A}' = \texttt{price}_2, C'_o = \texttt{Item}_1, C'_n = \texttt{Purchase}_1, \widetilde{C'_o} = \texttt{Item}_2, \widetilde{C'_n} = \texttt{Purchase}_2$ (we use subscripts to distinguish constructs in version v and $\tilde{v}$).

Let $tree(\mathcal{X}')$ return the subgraph of the smallest common subtree for a set $\mathcal{X}' \subseteq \mathcal{N}'$ containing the root B' of the common subtree, members of $\mathcal{X}'$ and for each $X' \in \mathcal{X}'$ path between X' and the root B' (see Figure 4 for an example). We define predicate $stable(\mathcal{X}') \leftrightarrow \mathcal{X}' \subseteq \{\mathcal{S}'_{all} \setminus \mathcal{S}'_a\} \wedge \{\forall X' \in \mathcal{X}' :$ X' is present in the new version, it was not moved, added or deleted and cardinality was not changed (if X' is an association)$\}$. The intuitive meaning of predicate $stable(\mathcal{X}')$ is that there were no radical changes in the structure for the members of $\mathcal{X}'$. If $C'_n \neq null, T' = tree(\{C'_o, C'_n\})$ and $stable(T')$ holds, than if:

[4] The formal definition of *XSEMPath* expressions is beyond the scope of this paper. Intuitively, the notation and usage is similar to *XPath* [15], only the *XSEMPath* expressions refer to the constructs from a PSM schema instead of the names of XML elements and attributes in the document. When executed upon a document $T \in \mathcal{T}(\mathcal{S}')$, *XSEMPath* query is translated to *XPath* query and the result of this query upon T is returned.

1. $\forall$ *association* $R' \in T'$: *card'*$(R') = m_{R'}..1$ (only cardinalities 0..1 and 1..1 are allowed in the affected part of the schema). In this simplest case the attribute $\tilde{A}'$ will have 0 or 1 instance in the old schema. Then this instance can be copied to the only one new location ($attMove_{\tilde{A}'}$ will be *identity* function).
2. C'_o is a descendant of C'_n in the PSM tree (the attribute is moved upwards, but the associations between C'_o and C'_n can have arbitrary cardinalities). Then all instances of A' under each instance of C'_n should be "aggregated" to one instance of $\tilde{A}'$ (this is the case depicted in Figure 3). Several aggregation functions can be offered (e.g. *sum, count, avg, max, min* known from relational databases or *concat* inlining the respective values).
3. C'_o is a descendant of C'_n in the PSM tree, *card'*$(A') = m'..1$ and *card'*$(\tilde{A}') = \widetilde{m'}..*$. Then this case is similar as the case above, but the cardinality of attribute $\tilde{A}'$ is adjusted, so all the values from existing instances can be used as values of $\tilde{A}'$, no aggregation is needed ($attMove_{\tilde{A}'}$ can be called *toList*).
4. C'_o is an ancestor of C'_n in the PSM tree, *card'*$(A') = m'..*$ and *card'*$(\tilde{A}') = \widetilde{m'}..1$. Then this is an inverse case to the one above. The respective values of A' can be distributed to the new locations. Nonetheless, a user may have to specify the distribution precisely ($attMove_{\tilde{A}'}$ can be called *distribute*).

When none of the conditions above is satisfied, a possible general approach is to use the function $attMove_{\tilde{A}'} = identityN$ which returns the value of the n-th instance of A' when required on the n-th location in the revalidated document. Other $attMove_{\tilde{A}'}$ functions must be entered by the user.

6 Revalidation

In the first step, the revalidation algorithm detects all changes between two versions of the model. If changes that require revalidation occur, a revalidation script is generated. Applying the script on a document valid against the old version produces a document valid against the new version. Since changes are defined as changes in the XSEM model, similarly as XSEM model can be translated to an XML schema in any XML schema language, the set of changes can be used to generate a revalidation script in any kind of implementation language.

Example 3. The in-depth description of the process of generating the revalidation script is beyond the scope of this paper. As an example, we will show the revalidation script in XSL for the schema depicted in Figure 3, concretely if the first interpretation from Example 4. In that case, we have to revalidate the *attributeMoved* change. In this example, we will use the subscript 1 for constructs from the old version and 2 for constructs from the new version. We are revalidating *attributeMoved*($\texttt{price}_2, \texttt{Purchase}_2, 0$). Since there are no other changes, *stable*($\{\texttt{Purchase}_1, \texttt{Item}_1\}$) also holds and since the association `Purchase-Item` has cardinality 1..$*$ and `Purchase` is a parent of `Item`, this case belongs to the second group mentioned on page 9 during the description of the revalidation of *attributeMoved* change (aggregation upwards). Before generating the script,

The XSL revalidation script consists of four templates. The first template processes `purchase` element – it creates the wrapping tag and calls a template for the moved PSM attribute `pur-price` and for `item` elements.
The second template processes item elements (the template is necessary, because we need to remove the `price` subelement from `item` elements).
The fourth template copies the content of each `name` element to the output.
The third, named, template `pur-price` calls the $attMove_{\texttt{price}_1}$ function (the *XSEMPath* expressions are converted to regular *XPath* expressions – *oldLocations* to `‘item/price’` and *newLocation* to `‘price’`). The user selected the function *sum* as $attMove_{\texttt{price}_1}$, a generic function built in the system. If none of the built-in function is suitable for the situation, the user can extend the system with a custom set of functions (having the same header as *sum*).

```
<xsl:stylesheet>
  <xsl:template match='/purchase'>
    <purchase>
      <xsl:call-template
        name="pur-price' />
      <xsl:apply-templates
        select='item' />
    </purchase>
  </xsl:template>
  <xsl:template
    match='/purchase/item'>
    <item>
      <xsl:copy-of select='name' />
    </item>
  </xsl:template>
  <xsl:template name='pur-price'>
    <price>
      <xsl:value-of select=
        'xc:sum(item/price,
                 price)'/>
    </price>
  </xsl:template>
  <xsl:template
    match='/purchase/item/name'>
    <xsl:copy-of select='.' />
  </xsl:template>
  <xsl:function name='xc:sum' >
    <xsl:param
      name='oldLocations' />
    <xsl:param
      name='newLocation' />
    <xsl:value-of select=
      'sum($oldLocations)' />
  </xsl:function>
</xsl:stylesheet>
```

Fig. 5. Example of a Revalidation Script in XSL

the user selects the function *sum* to be used as $attMove_{\texttt{price}_1}$. The resulting revalidation script in XSL is depicted and explained in Figure 5.

7 Conclusion and Future Work

In this paper we proposed a formal model for schema evolution. We formally defined changes that can be detected between two versions of a schema. We sketched an implementation of an algorithm that produces revalidation script on the basis of the detected set of changes.

The set of the detected changes is useful not only for the revalidation algorithm, but it is also possible to immediately decide, whether revalidation is needed or not and help the user to locate the changes in the schema.

The revalidation script can deal with structural modifications and with a significant part of the content modifications automatically, user input is required only where necessary (e.g. when a new content must be added during revalidation). We plan to extend its capabilities in adding content in our future work.

The algorithm in its current version deals mainly with revalidation of 1) structure and 2) data already present in the document. Because new data are often required for new versions, we will focus our future work on obtaining this data for the revalidated documents. For this purpose, we will utilize the existing connection between PIM and PSM and and a new similar connection between PIM and the model of a data storage (e.g. an ER schema [13]).

The change detection algorithm requires semantic links between the two compared versions of the schema, which are not available for schemas were imported and not created with an XSEM-enabled tool. Existing approaches to schema comparison and matching can be utilized as heuristics for creating these links.

Type inheritance is a fundamental part of large specifications, but support for inheritance is limited in the current version of XSEM. We intend to extend both PIM and PSM levels with inheritance support.

References

1. XCase – tool for XML data modeling. `http://xcase.codeplex.com/`.
2. *MOF 2.0 / XMI Mapping Specification, v2.1.1.* OMG, 2009.
3. R. Bourret. XML and Databases. September 2005. `http://www.rpbourret.com/xml/XMLAndDatabases.htm`.
4. D. Booth, C. K. Liu. *Web Services Description Language (WSDL) Version 2.0 Part 0: Primer.* W3C, June 2007. `http://www.w3.org/TR/wsdl20-primer/`.
5. G. Guerrini, M. Mesiti, and M. A. Sorrenti. Xml schema evolution: Incremental validation and efficient document adaptation. In *XSym*, volume 4704 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2007.
6. Hong Su, D. K. Kramer, E. A. Rundensteiner. XEM: XML Evolution Management, Technical Report WPI-CS-TR-02-09. 2002.
7. M. Klettke. Conceptual xml schema evolution — the codex approach for design and redesign. In *Workshop Proceedings Datenbanksysteme in Business, Technologie und Web (BTW 2007)*, pages 53–63, Aachen, Germany, March 2007.
8. J. Miller and J. Mukerji. *MDA Guide Version 1.0.1.* Object Management Group, 2003. http://www.omg.org/docs/omg/03-06-01.pdf.
9. M. Necasky. *Conceptual Modeling for XML*, volume 99 of *Dissertations in Database and Information Systems Series.* IOS Press/AKA Verlag, January 2009.
10. M. Necaský and I. Mlýnková. Five-level multi-application schema evolution. In *DATESO*, pages 90–104, 2009.
11. Object Management Group. *UML 2.1.2 Specification*, nov 2007. `http://www.omg.org/spec/UML/2.1.2/`.
12. T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau. *Extensible Markup Language (XML) 1.0 (Fifth Edition).* W3C, November 2008. `http://www.w3.org/TR/2008/REC-xml-20081126/`.
13. B. Thalheim. *Entity-Relationship Modeling: Foundations of Database Technology.* Springer Verlag, Berlin, Germany, 2000.
14. H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema Part 1: Structures (Second Edition).* W3C, October 2004. `http://www.w3.org/TR/xmlschema-1/`.
15. W3C. *XML Path Language (XPath) 2.0.* `http://www.w3.org/TR/xpath20/`.