XEM: Managing the Evolution of XML Documents *

Hong Su, Diane Kramer, Li Chen, Kajal Claypool and Elke A. Rundensteiner
Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609–2280
{*suhong*|*dkramer*|*lichen*|*kajal*|*rundenst*}*@cs.wpi.edu*

Abstract

As information on the world wide web continues to proliferate at an astounding rate, the extensible markup language (XML) has been emerging as a standard format for data representation on the web. In many applications, specific document type definitions (DTDs) are designed to enforce a semantically agreed-upon structure of the XML documents for management. However, both the data and the structure of XML documents tend to change over time for a multitude of reasons, including to correct design errors in the DTD, to allow expansion of the application scope over time, or to account for the merging of several businesses into one. However, most of the current software tools that enable the use of XML do not provide explicit support for such data or schema changes. In this vein, we put forth the first solution framework, called XML Evolution Manager (XEM) to manage the evolution of XML. XEM provides a minimal yet complete taxonomy of basic change primitives. These primitives, classified as either data changes or schema changes, are consistency-preserving, i.e., (1) for a data change, they ensure that the modified XML document conforms to its DTD both in structure and constraints; and (2) for a schema change, they ensure that the new DTD is a valid DTD and all existing XML documents are transformed also to conform to the modified DTD. We prove the completeness of the taxonomy in terms of DTD transformation. To verify the feasibility of our XEM approach we have implemented a working prototype system using PSE Pro as our backend storage system.

Keywords: Taxonomy of Changes, Schema Evolution, Consistency Preservation, XML Data Model.

*This work was supported in part by several grants from NSF, namely, the NSF NYI grant #IRI 94–57609, the NSF CISE Instrumentation grant #IRIS 97–29878, and the NSF grant #IIS 9988776. We also thank HP for partial support of Hong Su in terms of a grassroot grant and IBM for support of Li Chen in terms of an IBM Corporate Fellowship. Diane Kramer also thanks WPI for support via the Goddard Fellowship.

1 Introduction

Motivation. XML has become increasingly popular as the data exchange format over the Web [W3C00]. Although XML data is self-describing, most application domains tend to use Document Type Definitions (DTDs) to specify and enforce the structure of XML documents within their systems. DTDs thus assume a similar role as types in programming languages and schemata in database systems.

Many systems, such as Oracle 8i [Net00], IBM DB2 [IBM00] and Excelon [Obj99], have recently started to enhance their existing database technologies to accommodate and manage XML data. Many of them [Net00] assume that the DTD is provided in advance and will not change over the life of the XML documents. They hence utilize the given DTD to construct a fixed relational (or object-relational) schema which then can serve as structure based on which to populate the XML documents that conform to this DTD.

However, change is a fundamental aspect of persistent information and data-centric systems. Information over a period of time often needs to be modified to reflect perhaps a change in the real world, a change in the user's requirements, mistakes in the initial design or to allow for incremental maintenance. While these changes are inevitable during the life of an XML repository, most of the current XML management systems unfortunately do not provide enough (if any) support for these changes.

Motivating Example of XML Changes. We next give an example of changes in XML documents. Figure 1 depicts a DTD *article.dtd* and an XML document conforming to this DTD, both used as running example hence forth. Changes can be classified as either data updates or schema updates. An example of a data update is the deletion of the editor information, i.e, removal of *<editor name= "Won Kim">* from the actual XML document. In this case, an XML change support system would have to determine whether this is indeed a valid change that will result in an XML document still conforming to the given DTD. Since the *editor*

is a *REQUIRED* element in the specified DTD, such a data update should be rejected. Next, consider the DTD change where the definition of the element *monograph* which must have at least one sub-element *editor* is relaxed such that it is optional to have the sub-element *editor*. For such a DTD change, a change support system would need to verify that the suggested change leads to (1) a new valid DTD and (2) corresponding changes are propagated to all old XML documents to conform to the changed DTD. For our example, this leads to a DTD change, requiring no changes of the underlying XML data.

```
<!ELEMENT article (title,author+,related-work?)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (name)>
   <!ATTLIST author id ID #REQUIRED>
<!ELEMENT name (firstname,lastname)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT related-work (monograph)*>
<!ELEMENT monograph (title,editor>)
<!ELEMENT editor EMPTY>
   <!ATTLIST editor name CDATA #IMPLIED>

<article>
  <title>XML Evolution Manager</title>
  <author id = "dk">
    <name>
      <firstname>Diane</firstname>
      <lastname>Kramer</lastname>
    </name>
  </author>
  <author id = "er">
    <name>
      <firstname>Elke</firstname>
      <lastname>Rundensteiner</lastname>
    </name>
  </author>
  <related-work>
    <monograph>
      <title>Modern database systems</title>
      <editor name = "Won Kim">
    </monograph>
  </related-work>
</article>
```

Figure 1. Article.dtd and One Valid Sample XML Document

Problems with XML Management Systems. XML management systems attempt to expose a virtual XML document-view independent of the underlying storage system, be it relational, object-based or some specialized XML data structure. However in most current XML management systems [Net00, IBM00], evolution support, if any, is still inherently tied to the underlying storage system, to its data model and its change specification mechanism. For example, in Oracle 8i, if the structured XML documents are stored as object-relational instances, the user has to write SQL code to perform any type of updates. This requires users to be aware of the underlying storage system and the mapping mechanism between XML, DTD and their underlying storage model. It prevents users from expressing desired transformations independent of the targeted underlying system. It is likely to result in errors in terms of mismatch of desired XML transformation and the actual system change. In addition, the system-specific expression of transformation may induce extensive re-engineering work either for migration to another system or integration of several systems. In short, the development of a standard XML change specification and support system is necessary.

Moreover, as illustrated above, structural inconsistency may arise in the XML management system. It hence is critical to detect in advance whether an update is a valid operation that preserves the structural consistency. However, this problem is ignored in most existing XML management systems and not directly treated by the tools [Gro, Inf00] specially designed for transforming XML documents from one format to another.

XML Evolution Manager (XEM) Approach In this work we propose an XML Evolution Manager (XEM) as a middleware solution that provides uniform XML-centric data and schema evolution facilities. To the best of our knowledge, XEM is the first effort to provide such uniform evolution management for XML documents. In brief the contributions of our work are:

1. We identify the lack of generic support for change in current XML management systems such as [Net00, IBM00, Obj99].
2. We propose a taxonomy of XML evolution primitives that provide a system independent way to specify changes both at the DTD and XML data level.
3. We analyze change semantics and introduce the notion of constraint checking to ensure structural consistency during the evolution.
4. We can show that our proposed change taxonomy is complete.
5. We describe a working XML Evolution Management prototype system we have implemented to verify the feasibility of our approach.

2 Background: XML Data Model and DTD Data Model

2.1 The XML Data Model

Here we briefly review the XML Data Model [W3C00]. XML is composed of nested tagged elements. Each tagged element has a sequence of zero or more attribute/value pairs, and an ordered sequence of zero or more sub-elements. These sub-elements may themselves be tagged elements, or they may be "tag-less" segments of text data. A well-formed document may have an associated schema, derived from one or more XML Schema documents; it may have

an associated DTD; or it may have no schema, then called "schema-less".

An instance of the XML Data Model represents one or more complete XML documents or document parts. XML is a node-labeled, ordered tree-structured representation that includes the concept of node identities. A document is represented by a unique *DocNode* that is the root node of the XML data tree. Each element and attribute data is represented by *ElemNode* and *AttrNode* respectively. Simple type values such as String, Boolean, etc. are represented by *ValueNode*.

2.2 The DTD Data Model

In order to enforce a structure as needed for effective management, we assume that all XML documents have an associated Document Type Definition (DTD). DTD allows for properties or constraints to be defined on elements and attributes. In a DTD, elements represent the tag names that can be used in an XML document. Elements can in turn have content particles or attributes or be empty. The structure of elements is defined via a *content-model* built out of operators applied to its content particles. Content particles can be grouped as sequences (*a,b*) or as choices (*a|b*) to be a content particle again. For every content particle, the content-model can specify its occurrence in its parent content particle using regular expression operators ($?, *, +$). There are also some special cases of the content-model: *EMPTY* for an element with no content particles; *ANY* for an element that can contain any content particles; *#PCDATA* for an element that can contain only text. When the element can contain content particles mixed with text, the content-model is called a *mixed-content*.

Attributes can be of various types such as *ID* for a unique identifier or *CDATA* for text. They can be optional (*#IMPLIED*) or mandatory (*#REQUIRED*). Optionally, attributes can have a default or a constant value (*#FIXED*).

A DTD can be modeled as a graph. We denote the graph *G = (N, p, e)* where N is the set of nodes, p is the parent function representing the edges in the graph, and l is the labeling function representing a tuple of node's properties including the node's name and other properties if any. Each node $n \in N$ falls into three categories according to its label:

1. Tag node:
   (a) Element node: Each element node e represents an element type. $l(e) =< Name >$ where $Name$ is element e's name.
   (b) Attribute node: Each attribute node a represents an attribute type. $l(a) =< Name, Type, DefType, Val(a) >$ where $Name$ is attribute a's name, $Type$ is a's type, e.g., CDATA, ID, IDREF, IDREFS, ENUMERATION etc., $DefType$ is a's default type, i.e., *#REQUIRED*, *#IMPLIED*, *#FIXED*, *#DEFAULT*, and Val is a's default value if any.

   We use $l(a).Name$ to denote the property $Name$ of node a and $|l(a)|$ to denote how many properties a has.
2. Constraint node:
   (a) Group node: Each group node g represents how its direct children are grouped together, that is, by sequence (i.e., $l(g) =<$ ",$''>$) or by choice (i.e., $l(g) =<$ "|$''>$).
   (b) Quantifier node: Each quantifier node q has only one child. It represents how many times its child can occur in its parent. The label of q can be:
      i. $<$ "$*''>$: the child is repeatable but not required
      ii. $<$ "$+''>$: the child is repeatable and required
      iii. $<$ "$?''>$: the child is neither repeatable nor required
3. Built-in node
   (a) Root node *dtdRtNode*: It is the entry for the DTD graph. All the nodes in DTD graph can be traversed starting from this node in a breadth-first manner.
   (b) Primitive data type node *PCDATA*: It represents PCDATA which is XML-specific rather than application specific. Its parent must be an element node, indicating the element is of type MIXED.

For example, the DTD in Figure 1 can be represented as the graph shown in Figure 2. The element *article*'s content model consists of a sequence of content particles as represented by the group node with label ",". The first content particle, element *title*, is represented by an element node with a label "title". And *author+*, is represented by a group node labeled "+" with a child element node labeled "author". Similarly, *related-work?*, is represented by a group node labeled "?" with a child element node labeled "related-work".

In addition to the nodes, DTD model also provides the concept of *Position in Content Particle*. We traverse the graph breadth- first and use a list of integers to refer to a content particle node. To illustrate, we use the following sample element type definition: *<!ELEMENT a* (*b*, (*c* |*d*))>. Here position [1] refers to group *(b, (c|d))*; [1,1] refers to sub-element *b*; [1,2] refers to group *(c|d)*; [1,2,1] refers to sub-element *c*; [1,2,2] refers to sub-element *d*.

An XML data tree is derived from a DTD graph by instantiating each node in the DTD graph. We call the nodes in XML data tree *instance nodes* of the DTD graph node which they are associated with.

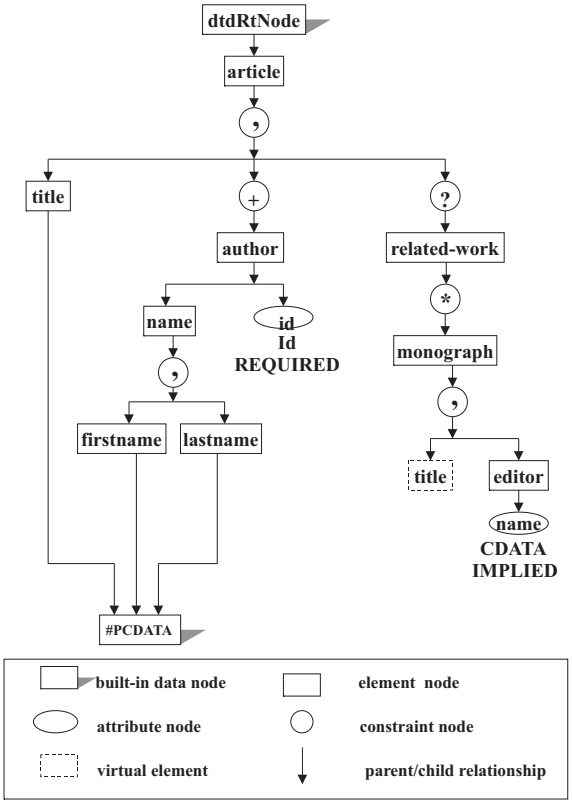

Figure 2. Graph Representation of Article.dtd

3 Taxonomy and Semantics of XML Change Primitives

3.1 Introduction

In this section we present the taxonomy of XML change primitives and define their semantics. The primitives fall into two categories: those pertaining to the DTD, and those pertaining to the XML data. We ensure that the execution of primitives violates neither the invariants nor the content model. Our goal is to provide a set of primitives with the following characteristics:

- **Complete:** All valid changes to manipulate DTD and XML are possible using our primitives.
- **Minimal:** Each primitive is atomic such that it cannot be achieved by combining two or more primitives.
- **Sound:** Every primitive is guaranteed to maintain system integrity in terms of well-formedness of both DTD and XML data, and consistency between DTD and XML data.

We list our primitives for DTD and XML data changes in Table 1. We then give the details of the DTD change primitives. Due to space limitations, we do not describe the XML data change primitives in detail here [Kra01].

DTD Operation	Description
createDTDEl(*u*)	Create element with name *u*
destroyDTDEl(*u*)	Destroy element with name *u*
renameDTDEl(*u, u'*)	Rename element from name *u* to *u'*
insertDTDEl(*E, pos, P, q, d*)	Add element *E* at position *pos* to parent *P* with quantifier *q* and default value *d*
removeDTDEl(*E, pos*)	Remove sub-element at position *pos* in parent *E*
changeQuant(*E, pos, q, d*)	Change quantifier of content particle at position *pos* in parent *E* to quantifier *q* with default value *d*
convertToGroup(*E, start, end*)	Group sub-elements from position *start* to position *end* in parent *E* into a list group
flattenGroup(*E, pos*)	Flatten group at position *pos* in element *E* to a list of sub-elements
changeGroupQuant(*E, pos, q*)	Change quantifier of group at position *pos* in element *E* to *q*
addDTDAtt(*u, E, t, d, v*)	Add attribute with name *u* to element *E* with type *t*, default type *d*, and default value *v*
destroyDTDAtt(*u, E*)	Destroy attribute with name *u* from element *E*
changeAttDefType(*u, E, t, v*)	Change element *E*'s attribute *u*'s type to *t*, with default value *v*
changeAttDefValue(*u, E, v*)	Change element *E*'s attribute *u*'s default value to *v*
changeAttFixedValue(*u, E, v*)	Change element *E*'s attribute *u*'s fixed value to *v*
XML Data Operation	**Description**
addDataAtt(*a, v, pos*)	Add attribute with name *a* with value *v* to position *pos*
destroyDataAtt(*a*)	Destroy attribute *a*
changeDataAtt(*a, v, e*)	Change attribute *a*'s value in element *e* to *v*
addDataEl(*e, pos*)	Add element *e* at position *pos*
destroyDataEl(*e*)	Destroy element *e p*
changeDataEl(*e, v*)	Change element *e*'s value to *v*

Table 1. DTD and XML Data Change Primitives

3.2 DTD Change Primitives

In this section, we define the semantics of each DTD change primitive. To ensure that the targeted DTD is valid, preconditions are enforced on each change primitive, i.e., a primitive will not be executed unless the corresponding preconditions are satisfied. We assume the following change primitives are applied to $G_1 = (N_1, p_1, l_1)$ and produce $G_2 = (N_2, p_2, l_2)$ as output. We use $c_1(m)$ to represent nodes m's children where $m \in N_1$. And we denote the child at position pos in node m in G_1 by $c_1(m, pos)$. A node $n = c_1(m, pos)$ may have more than one parent, and we denote the parent which is on the path from m to n from which pos is derived by $q_1(n, m, pos)$. $c_2(m)$, $c_2(m, pos)$, $q_2(n, m, pos)$ represent the same concepts in G_2.

3.2.1 Changes to the Document Definition

1. **createDTDEl(u)**

   Preconditions: No element type with name u has been defined, i.e., $\forall n \in N_1, l_1(n) \neq < u >$.

Results: A new element *E* with name u will be created with empty content. We get a graph $G_2 = (N_2, p_2, l_2)$ where $N_2 = N_1 \cup \{E\}$, $p_2(E) = \{dtdRtNode\}$, $l_2(E) =< u >$, and $p_2(n) = p_1(n)$, $l_2(n) = l_1(n)$, $\forall n \in N_1$. This primitive causes no changes to the XML data.

2. **destroyDTDEl(E)**

   Preconditions: Element *E* must be a non-nested element node whose content model is either *EMPTY* or composed of only *PCDATA*, i.e., $c_1(E) = \Phi$ or $c_1(E) = \{PCDATA\}$.

   Results: The element *E* will be removed from any content model in which it is a content particle. We get a graph $G_2 = (N_2, p_2, l_2)$ where $N_2 = N_1 - \{E\}$, and $p_2(n) = p_1(n)$, $l_2(n) = l_1(n)$, $\forall n \in N_1 - \{E\}$. All the instance nodes of element *E* will be deleted from the XML data trees.

3.2.2 Changes to an Element Type Definition

1. **insertDTDEl(*E, pos, P, q, d*)**

   Preconditions: If quantifier *q* signifies a *required* constraint and *E* is a *PCDATA* element, the default value *d* must not be null.

   Results: An existing element *E* will be added to the content model of parent element *P* at position *pos*. We get a graph $G_2 = (N_2, p_2, l_2)$ where $N_2 = N_1$, $p_2(E) = p_1(E) \cup \{P\}$, $p_2(n) = p_1(n)$, $\forall n \in N_1 - \{E\}$, and $l_2(m) = l_1(m)$, $\forall m \in N_1$. If *q* signifies a *required* constraint, then an instance node of element *E* with default value *d* will be added as a child to each instance node of *P* in the XML data trees.

2. **removeDTDEl(*E, pos*)**

   Preconditions: $c_1(E, pos)$ must be a non-nested element node.

   Results: The element node $m = c_1(E, pos)$ is removed from E's content model. We get a graph $G_2 = (N_2, p_2, l_2)$ where $N_2 = N_1$, $p_2(m) = p_1(m) - q_1(m, E, pos)$. All the instance nodes of content particle m are removed.

3. **changeQuant(*E, pos, q, d*)**

   Precondition: $c_1(E, pos)$ is a content particle node which is either an element node or a constraint node.

   Results: The quantifier for the content particle $t = c_1(E, pos)$ will be changed to q. We get a graph $G_2 = (N_2, p_2, l_2)$ where $N_2 = N_1$, $p_2(n) = p_1(n)$, $\forall n \in N_1$, $l_2(q_2(t, E, pos)) =< q >$, and $l_2(m) = l_1(m)$, $\forall m \in N_1 - \{q_2(t, E, pos)\}$. The XML data changes required for this primitive depend on the old and new quantifier values. Due to space limitations, we summarize using the following two rules:

   (a) If the old quantifier represented a *repeatable* constraint and the new quantifier does not, we find all the instance nodes of content particle *t* and remove all but the first occurrence of the instance node of *t*.

   (b) If the new quantifier represents a *required* constraint and the old quantifier did not, for each instance node *e* of element *E* which did not contain any instance node of *t*, we must create a new instance node of element *t* with the default value *d* and insert it to *e*'s children list.

4. **convertToGroup(*E, start, end*)**

   Preconditions: All the content particles falling within the range $(start, end)$ share a same parent.

   Results: All the content particles that range from position $start$ to end in content particle E are grouped into a list. We get a graph $G_2 = (N_2, p_2, l_2)$ where $N_2 = N_1 \cup \{g\}$, $l_2(g) =<$ ",$" >$, $p_2(m) = (p_1(m) - q_1(m, E, pos)) \cup \{g\}$, $\forall m \in c_1(E, pos)$ where pos falls within the range $(start, end)$, and $l_2(n) = l_1(n)$, $n \in N_1$. To illustrate, we use the following sample element type declaration: *<!ELEMENT author* (*address*, *lastname*, *firstname*)>. $convertToGroup(author, [1,2], [1,3])$ will change the element type declaration to *<!ELEMENT author* (*address*, (*lastname*, *firstname*))> This primitive causes no changes to the XML data.

5. **flattenGroup(*E, pos*)**

   Preconditions: $c_1(E, pos)$ must be a list group node.

   Results: The group $g = c_1(E, pos)$ will be flattened. We get a graph $G_2 = (N_2, p_2, l_2)$ where $N_2 = N_1 - \{g\}$, $p_2(m) = q_1(g, E, pos)$, $\forall m \in c_1(g)$, and $l_2(n) = l_1(n)$, $\forall n \in N_1 - \{g\}$. To illustrate, we use the following sample element type declaration: *<!ELEMENT author* (*address*, (*lastname, firstname*))>. $flattenGroup(author, [1,2])$ will change the element type declaration to *<!ELEMENT author* (*address*, *lastname*, *firstname*)>. This primitive causes no changes to the XML data.

6. **changeGroupQuant(*E, pos, q*)**

   Preconditions: $c_1(E, pos)$ must be a group node. We do not allow the new quantifier to represent a *required* constraint if the old quantifier did not. This is because it would be difficult to specify appropriate default values for a group of element types to assign to their data instances which do not already contain this group.

   Results: The quantifier for the group $g = c_1(E, pos)$ will be changed to q. We get a graph $G_2 = (N_2, p_2, l_2)$ where $N_2 = N_1$, $q_2(n) = p_1(n)$, $\forall n \in N_1$, $l_2(q_2(g, E, pos)) =< q >$, and $l_2(n) = l_1(n)$, $\forall n \in N_1 - q_2(g, E, pos)$. The semantics for this primitive

are similar to those described above for changing the quantifier of a sub-element.

7. **addDTDAtt(***u, E, t, d, v***)**

   Preconditions: No attribute with name u has been defined in element E, i.e., $\forall n \in c_1(E)$ where $|l_1(n)| > 1$ (n is an attribute node), $l_1(n).Name \neq u$. If the default type is *#FIXED* or *#REQUIRED*, the default value *v* must not be null.

   Results: A new attribute will be added to the element type definition of E. We get a graph $G_2 = (N_2, p_2, l_2)$ where $N_2 = N_1 \cup \{a\}$, $l_2(a) =< u, t, d, v >$. If default value *v* is not null, for each instance node *e* of element type *E*, a new instance node of the new attribute type with value *v* will be added to *e*'s attribute set.

8. **destroyDTDAtt(***u, E***)**

   Preconditions: There exists an attribute with name u defined in Element E.

   Results: The attribute with name *u* will be removed from the element type declaration of element *E*. We get a graph $G_2 = (N_2, p_2, l_2)$ where $N_2 = N_1 - \{n\}$, $n \in c_1(E)$, $|l_1(n)| > 1$ and $l_1(n).Name = u$. The instance node of this attribute type will also be removed from the XML data trees.

3.2.3 Changes to an Attribute Type Definition

1. **changeAttDefType(***u, E, t, v***)**

   Preconditions: If the default type *t* of attribute with name u in element E is *#REQUIRED* or *#FIXED*, the value *v* must not be null.

   Results: The default type of the attribute with name u in E will be changed to *t*. The XML data changes required for this primitive depend on the old and new attribute types. Rather than listing each possibility separately here, we summarize the rules for those DTD changes that will induce data changes:

   (a) If the attribute's new default type is *#REQUIRED* and the old default type is *#IMPLIED*, value *v* will be assigned to the attribute's each instance node that does not have a value before.
   (b) If the attribute's new default type is *#FIXED*, value *v* will be assigned to all the instance nodes of the attribute.

2. **changeAttDefValue(***u, E, v***)**

   Preconditions: The attribute with name u in element E must be of default type *#DEFAULT*, as default values for the default types *#REQUIRED* and *#IMPLIED* are not allowed in an attribute default declaration, and changing the default value for the default type *#FIXED* is taken care of in a separate primitive.

   Results: The default value for the attribute will be changed to *v*. All instance nodes' values of the attribute will be made to conform to the new default value *v* by first checking whether there was a default or an actual value before. If the value was a default before, then the value will be changed to the new default *v*. Otherwise, the old value has higher precedence over a new given default value and hence the actual old value will remain unchanged.

3. **changeAttFixedValue(***u, E, v***)**

   Preconditions: The attribute with name u in element E must be of default type *#FIXED*.

   Results: The fixed value for the attribute will be changed to *v*. All instance nodes' values of the attribute will be changed to the new fixed value *v*.

We illustrate in Figure 3 how to use these primitives to achieve the changes we have described in Section 1, i.e., removal of <*editor name = "Won Kim"*> from the XML document in Figure 1. First we use DTD change primitive *changeQuant* to change the quantifier of content particle $< 1, 2 >$ in element *monograph* to "?", i.e., sub-element *editor* is optional in its parent element *monograph*. And then we can use XML data primitive *destroyDataEl* to safely destroy the *ElemNode* specified by an XPath expression [W3C99] which results in a new XML document conforming to the new DTD.

```
changeQuant(monograph, [1,2], ``?'');

destroyDataEl(``article/related-work/monograph[1]/editor'');
```

Figure 3. Primitives to describe the change

4 Completeness of DTD Change Operations

The taxonomy in Section 3 intuitively captures all changes needed to manipulate a DTD. In this section we show that this set of changes indeed subsumes every possible type of DTD change (completeness criteria). The proof given here has its basis on the completeness proof for the evolution taxonomy of Orion [BKKK87].

With the DTD graph we focus primarily on manipulations of nodes and of the directed edges between parent and children nodes. Towards that end we define six operations that correspond to the operations that we have defined for a DTD in Section 3, Table 1. We prove that every legal DTD graph operation is achievable using this set of six operations. This set of operations and its basic semantics for the DTD graph are given in Table 2.

Lemma 1 *For any given DTD graph* G*, there is a finite sequence of* {op6} *that can reduce the DTD graph* G *to another DTD graph* E' *with only one root node.*

EGM Notation	Operation	Description	Taxonomy Equivalent
op1	add-attribute	Adds new attribute to node	3.2.2.7
op2	delete-attribute	Deletes new attribute from the node (element)	3.2.2.8
op3	remove-node-edge	Removes the edge from the parent to the node	3.2.2.2, 3.3.2.5
op4	add-node-edge	Adds an edge between the parent and node	3.2.2.1, 3.2.2.4
op5	create-nonattr-node	Creates a new node; default is no parent	3.2.1.1
op6	delete-nonattr-node	Deletes a leaf node	3.2.1.2

Table 2. The DTD Graph Operations.

Proof: It is apparent that if we repeatedly apply the operation op6 which removes a non-nested element node n, we can after a finite number of applications reduce any given DTD graph G to a new DTD graph *G'* which only has the root node.

Lemma 2 *There is a finite sequence of operations* {op1, op4, op5} *that generates any desired DTD graph* G *from a DTD graph with only a root node* G'.

The proof based on a construction algorithm that adds all nodes in a breadth-first manner can be found in [Kra01].

Theorem 1 *Given two arbitrary DTD graphs* G *and* G', *there is a finite sequence* F *of* {op1, op4, op5, op6}, *such that* F(G) = G'.

Proof: We can prove this by first reducing the DTD graph *G* to an intermediate DTD graph G1 using Lemma 1. The DTD graph G1 can then be converted to the EGM G' using Lemma 2. □

Theorem 2 *Given two arbitrary DTD graphs* G *and* G', *there is a finite sequence of DTD graph operations* F *such that* F(G) = G'.

Proof: The set of operations {op1, op4, op5, op6} is a subset of the operations {op1, op2, op3, op4, op5, op6}. Hence the completeness of this set of operations is given from Theorem 1.

Soundness and Minimality of Primitives. A taxonomy of primitives is sound if every operation on a DTD graph produces as output a valid DTD graph. While we do not formally prove soundness of our primitives, it is intuitive that the semantics of the primitives such as adding a node (op5) produce as output a valid graph model conforming to the DTD properties.

Moreover, we have taken care to define minimal semantics as well as a minimal set of primitives, i.e., no primitive defined in the taxonomy subsumes the functionality of another primitive defined in the taxonomy. We do not formally prove this as the proof for this is rather laborious requiring proof steps for each defined primitive.

5 System Implementation: MARROW

To verify the feasibility of our approach, we have implemented the ideas presented in this paper in a prototype system. We have implemented Marrow [Kra01], a working framework for XML management[1]. In Marrow we use Excelon Inc's Pse Pro [Obj93], a lightweight object database system repository, as the underlying persistent storage system for XML documents. We require that the DTDs with which the incoming XML documents will comply are entered first into the system. PSE Pro's schema repository has been enhanced to not only manage traditional OO schema but also DTD as meta-data. The DTD-OO schema mapper generates an OO schema according to the DTD meta-data. Then we load the XML documents into the just prepared schema. The mapping and loading details are given in [Kra01]. We implemented all the proposed change primitives. Comparison of the performance of using the primitives to achieve incremental change versus reloading from scratch can be found in [Kra01].

6 Related Work

Most object database systems (ODB) [Tec94, Obj93] today have build-in schema evolution facilities for supporting the re-structuring of the application schema. Besides those simple pre-defined schema evolution operations, research has gone further to deal with complex changes [Bré96]. They string together several primitives to form higher level yet still specific change transformations. Finally, SERF [CJR98] is an extensible schema evolution framework that allows complex user-defined schema transformations in a flexible yet secure fashion.

Since XML data has an inherent nature of being "loosely" structured, some projects either totally ignore the schema of XML data or just consider it implicated by the actual storage structure and hence to be a "second-class" citizen. They therefore do not deal with schema evolution issues. DOEM [CAW98] is further proposed as a model to

[1] The preliminary Marrow system - ReWeb [RCC+00] has been demonstrated at ACM SIGMOD 2000.

represent changes in semi-structured data via temporal annotations. However, it only deals with the changes at the data level and is schema-blind. All versions of a data item will be stored together over time. Hence it results in an ever-growing complex annotated data structure.

More recently, tools are emerging to map XML data to traditional databases as data storage devices. Oracle's XML SQL Utility (XSU) [Net00] and IBM's DB2 XML Extender [IBM00] are well-known commercial relational products extended with XML support. They mainly provide two methods to manage XML data. The first option is to store XML data as a blob while the second option is to decompose XML data to relational instances. However, if there is any update to the external XML data, for the first storage option, they need to reload the data, and for the second option, they have to first figure out and then make the change on the relational schema level. In other words, the evolution of the data inside or outside of the database are independent from each other. Hence the change propagation from an external XML document to its internal relational storage or schematic structure is not supported. In a related effort at WPI, we have developed the Clock system [ZMLR01] that synchronizes internal relational storage with external XML documents.

XSLT [Gro] is a language designed for transforming individual XML documents. It does not require any DTD and users can specify arbitrary XML data transformation rules. Hence no schema constraints are enforced on the data or on the transformation. Lexus (XML Update Language) [Inf00] is a declarative language proposed by an open source group, Infozone, to update stored documents. However, its primitives also only work on the document level without taking DTD into account. So both XSLT and Lexus cannot serve in the scenario where structure is required.

7 Conclusion

In this paper, we present the first of its kind - a taxonomy of XML evolution operations. These primitives assure the consistency of XML documents, both when DTD changes are made and XML documents have to conform to the changes; and also when individual XML documents are changed to ensure that the changed documents still correspond to the specified DTD. We have implemented an XEM prototype system. The performance analysis can be found in [Kra01].

References

[BKKK87] J. Banerjee, W. Kim, H.J. Kim, and H.F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *ACM SIGMOND Record*, pages 311–322, 1987.

[Bré96] Philippe Bréche. Advanced Primitives for Changing Schemas of Object Databases. In *CAISE*, 1996.

[CAW98] S. Chawathe, S. Abiteboul, and J. Widom. Representing and Querying Changes in Semistructured Data. In *ICDE*, pages 4–13, February 1998.

[CJR98] K.T. Claypool, J. Jin, and E.A. Rundensteiner. SERF: Schema Evolution through an Extensible, Re-usable and Flexible Framework. In *CIKM*, pages 314–321, November 1998.

[Gro] W3C XSL Working Group. XSL Transformations (XSLT). http://www.w3.org/TR/xslt/.

[IBM00] IBM Software. DB2 XML Extender. http://www-4.ibm.com, 2000.

[Inf00] Infozone Group. Lexus. http://www.infozone-group.org/lexusDocs/html/wd-lexus.html, 2000.

[Kra01] D. Kramer. XML Evolution Management, Master Thesis, Worcester Polytechnic Institute, 2001.

[Net00] Oracle Technologies Network. Oracle8i. http://www.oracle.com/database/oracle8i, 2000.

[Obj93] ObjectStore, Inc. *ObjectStore Manual*, 1993.

[Obj99] Object Design. Excelon Data Integration Server. http://www.odi.com/excelon, 1999.

[RCC+00] E. A. Rundensteiner, K. T. Claypool, L. Chen, H. Su, and K. Onoeki. SERFing the Web: A Comprehensive Approach for Web Site Management. In *SIGMOD*, page 585, May 2000.

[Tec94] O_2 Technology. *O_2 Reference Manual, Version 4.5* . O_2 Technology, Versailles, France, 1994.

[W3C99] W3C. XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath, 1999.

[W3C00] W3C. XML Query Data Model. http://www.w3.org/TR/query-datamodel, 2000.

[ZMLR01] X. Zhang, G. Mitchell, Wang. Lee, and E. Rundensteiner. Clock: Synchronizing Internal Relational Storage with External XML Documents. In *RIDE*, April 2001.