XEM: XML Evolution Management

Hong Su, Diane S. Kramer, Elke A. Rundensteiner
Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609-2280
{suhong|dkramer|rundenst}@cs.wpi.edu

Abstract

XML has been emerging as a standard format for data representation on the web. In many application domains, specific document type definitions (DTDs) are designed to enforce the structure (schema) of the XML documents. 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 environmental changes such as the merging of several businesses into one. Most of the current software tools that enable the use of XML do not provide explicit support for such data or schema changes. Using these tools in a changing environment entails first making manual edits to DTDs and XML documents and thereafter reloading them from scratch. To overcome this limitation, we put forth a framework, called the XML Evolution Manager (XEM), to manage the evolution of DTDs and XML documents. XEM provides a minimal yet complete taxonomy of basic change primitives. These primitives, classified as either schema or data changes, are consistency-preserving. For a schema change, they ensure that the new DTD is legal, and all existing XML documents are transformed to also conform to the modified DTD. For a data change, they ensure that the update is only performed if the modified XML document would conform to its DTD. We have implemented a working prototype system called XEM-Tool in Java with PSE Pro Object Oriented Database as our backend storage system. Our experimental study using this system compares the relative efficiencies of using these the primitive operations for in-place XML data and schema changes in terms of their execution times against the time to reload the modified XML data into the backend OO storage system.

Keywords: XML, Document Type Definition, Schema Evolution, Structural Consistency, XML Evolution Management

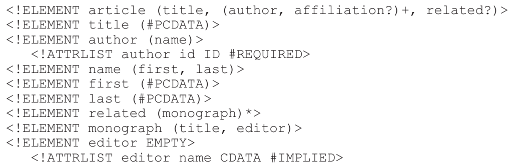
1 Introduction

1.1 Motivation

XML [38], the Extensible Markup Language, has become increasingly popular as the data exchange format over the Web. Although XML data is considered to be "self-describing", many application domains tend to use Document Type Definitions (DTDs) [38] or XML Schema [39] to specify and enforce the structure of XML documents within their systems. A DTD defines for example which tags are permissible in an XML document, the order in which such tags must appear and how the tags are nested to form a hierarchical structure. DTDs thus assume a similar role as types in programming languages and schema in database systems.

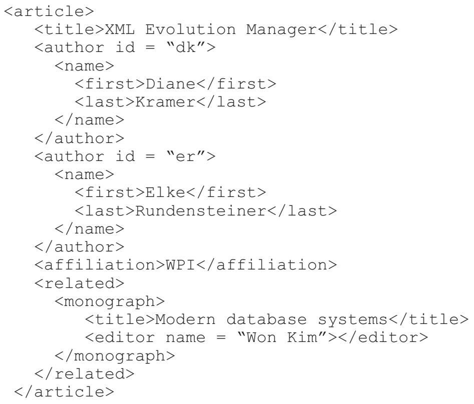
Many database vendors, such as Oracle 8i [27], IBM DB2 Extender [19] and Excelon [29], have recently started to enhance their existing database technologies to manage XML data as well. Many of them [27] assume that a DTD is provided in advance and will not change over the life of the XML documents. They hence utilize the given DTD to construct a relational [19] or object-relational [27] schema which serves as the structure into which to populate the XML documents. For example, Oracle8i [27]) provides one fixed mapping between a DTD and relational schemas. The mapping is done by matching the element tag names with the column names in the table. Elements with text only content map to scalar columns and elements containing subelements map to object types. In the DB2 XML Extender [19], a user can define a Document Access Definition (DAD) file to specify their own mapping. The DAD is an XML formatted document which allows the user to associate the XML document structure with particular relational tables and columns. However all of these systems do not provide sufficient change support for XML data.

We note that change is a fundamental aspect of persistent information and data-centric systems [32]. 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,



```
<!ELEMENT article (title, (author, affiliation?)+, related?)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (name)>
   <!ATTRLIST author id ID #REQUIRED>
<!ELEMENT name (first, last)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT related (monograph)*>
<!ELEMENT monograph (title, editor)>
<!ELEMENT editor EMPTY>
   <!ATTRLIST editor name CDATA #IMPLIED>
```

Figure 1: Sample DTD: *Article.dtd*



```
<article>
   <title>XML Evolution Manager</title>
   <author id = “dk”>
     <name>
       <first>Diane</first>
       <last>Kramer</last>
     </name>
   </author>
   <author id = “er”>
     <name>
       <first>Elke</first>
       <last>Rundensteiner</last>
     </name>
   </author>
   <affiliation>WPI</affiliation>
   <related>
     <monograph>
        <title>Modern database systems</title>
        <editor name = “Won Kim”></editor>
     </monograph>
   </related>
 </article>
```

Figure 2: Valid Sample XML Document Conforming to *Article.dtd*

mistakes in the initial design or to allow for incremental maintenance. While these changes are also 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.

1.2 Motivating Example of XML Changes

Here we present an example how changes in XML documents lead to various data management issues that must be addressed. Figure 1 depicts an example DTD *Article.dtd* on publications and Figure 2 shows a sample XML document conforming to this DTD. These sample documents are used for running examples hence forth in the remainder of this paper.

Changes can be classified as either data changes or schema changes. An example of a data change is the deletion of the editor information, i.e., removal of <*editor name* = “*Won Kim*”> from the XML document in Figure 2. In this case, an XML change support system would have to determine whether this is indeed a legal change that will result in an XML document still conforming to the given DTD. Since the element definition for *monograph*, <!*ELEMENT monograph* (*title*, *editor*)>, requires that the *editor* subelement must occur exactly once in the parent element *monograph*, this data change should be rejected.

Now, consider the DTD change where the definition of the element *monograph*, which must have an *editor* subelement, is relaxed such that it is optional to have the *editor* subelement, i.e., <!*ELEMENT monograph* (*title*, *editor*)> is changed to <!*ELEMENT monograph* (*title*, *editor*?)>. For such a DTD change, a change support system would need to verify that (1) the suggested change leads to a new legal DTD conforming to the DTD specification [38] and (2) the corresponding changes are propagated to the existing XML documents to conform to the changed DTD. A single occurrence of the *editor* subelement in the XML data would still conform to a new DTD definition in which the *editor* subelement is optional. Therefore this DTD change requires no changes to the underlying XML data. In fact, in this case, we can make this particular decision without even having to consult the particular XML data instances.

1.3 Limitations of XML Management Systems

In most current XML data management systems [27, 19], change support, if any, is inherently tied to the underlying storage system, its data model and its change specification mechanism. For example, in IBM DB2 XML Extender, once

XML Enabled System	Data Update	Schema Update	Generality	Consistency
DB2 XML Extender	Yes	No	No	No
Oracle 8i	Yes	No	No	No
Excelon	Yes	Yes	General for Data Update	No

Table 1: Support for XML Change

the structured XML documents are stored as relational instances, the user has to write SQL code to perform any type of update on the documents. This requires users to be aware of the underlying relational database, and the mapping between the DTD and the schema of the relational database as expressed by the DAD mapping file. In addition, the specification of updates tightly coupled to a specific XML data management system may induce extensive re-engineering work either for migration to another system or integration of several systems. This clearly points out the need for the development of a standard XML change specification and support system.

Moreover, a database system should maintain structural consistency [1], i.e., data should always be consistent with its schema. Hence, it is critical to detect in advance whether an update is a legal operation that preserves the structural consistency as illustrated in Section 1.2. However, this problem is ignored in most existing XML data management systems [27, 19, 29] and the tools [18, 21] specially designed for transforming XML documents from one format to another.

In Table 1, we compare the update support of the commercial XML data management systems in the following four aspects, namely (1) support for XML data updates, (2) support for XML schema updates, (3) update specification general for XML or tied to particular native format of the back-end storage engine, and (4) if the update is ensured to preserve the structural consistency of the XML data and the associated XML schema.

1.4 XML Evolution Manager (XEM) Approach

In this work we fill this void by proposing a general XML evolution management system that provides uniform XML-centric schema and data evolution facilities. To the best of our knowledge, XEM is one of the first efforts to provide such uniform evolution management for XML documents. The contributions of our work are:

1. We identify the lack of generic and safe support for evolution in current XML data management systems such as [27, 19, 29].

2. We propose a taxonomy of XML evolution primitives that provides a system independent way to specify changes to both the DTDs and XML documents.

3. We ensure three forms of system integrity during evolution in order for the change support system to be sound: *legal* DTDs, *well-formed* XML documents and *valid* XML documents.

4. We show that our proposed evolution taxonomy is complete and sound.

5. We develop a working XML Evolution Management prototype system called XEM-Tool using the Java object server PSE Pro as storage system to verify the feasibility of our approach.

6. We conduct experimental studies on XEM-Tool to assess the relative costs associated with different evolution primitives. We also analyze the dependency between specific implementation choices made and the resulting impact on change performance.

1.5 Outline.

The remainder of this paper proceeds as follows. Section 2 provides background information on XML documents and DTDs, and shows how we model these constructs in our system. In Section 3 we present our taxonomy of evolution primitives, and provide proofs showing that the taxonomy is both complete and sound. Section 4 reviews our prototype design and implementation. In Section 5 we present our experimental studies, including tests run on our prototype system and the results from those tests. Section 6 discusses other related research upon which we base our work. And finally, in Section 7 we present our conclusions, including future areas of study that could be taken up to continue this research, and a summary of the main contributions of this work.

2 XML and DTD Data Model

2.1 Background on DTD and XML

Both Document Type Definitions (DTDs) [38] and XML Schema [39] define the structure and content of an XML document. XML Schema is more powerful than a DTD. For example, it supports list types whereas a DTD cannot. However, XML Schemas are still in the preliminary stages of a proposed recommendation, while DTDs are currently the dominant de-facto industry standard. For this project, therefore, we choose to focus on DTDs rather than XML Schemas. However, our results should be transferable to XML Schemas with some extensions.

A DTD is *legal* if it conforms to the DTD specification [38]. For example, if a DTD uses illegal characters in element type names, or defines two element types with the same name, it is then not legal.

A XML document is *well-formed* if it meets all the well-formedness constraints enumerated in the specification [38]. For example, a well-formedness constraint of *element type match* requires that the name in an element's end-tag must match the element type in the start-tag (e.g., *</name>* matches *<name>*). The *unique attribute specification* constraint prohibits that one attribute name appears more than once in the same start-tag or empty-element tag.

A well-formed XML document can in addition be *valid* if it has an associated document type declaration (DTD) and if the document complies with the constraints expressed in it.

In the following, in order to distinguish an element in an XML document from an element declaration in a DTD, we use the term *"element instance"* to refer the former as opposed to the term *"element definition"* referring the latter. Similarly, the term *"attribute instance"* is used as opposed to *"attribute definition"*.

2.2 The XML Data Model

A tree-structure can be used to represent an XML document. We use the following notation to describe our model of an XML data tree.

Definition 1 An **XML data tree** is a quadraple T = (N, $childrenON$, $childrenUN$, $labelN$) where N is the set of nodes in the tree, $childrenON$ is a function representing the relationship between a node and its ordered children while $childrenUN$ represents the relationship between a node and its unordered children with $childrenON$, $childrenUN$: $N \rightarrow N^k$, $k \geq 0$; and $labelN$ is a labeling function: $labelN$: $N \rightarrow I \times S$, where I is the set of node types, i.e., I ={`XMLDOC, ELEMNODE, ATTRNODE, VALNODE`}, and S is the set of strings which can serve as legal names of the node.

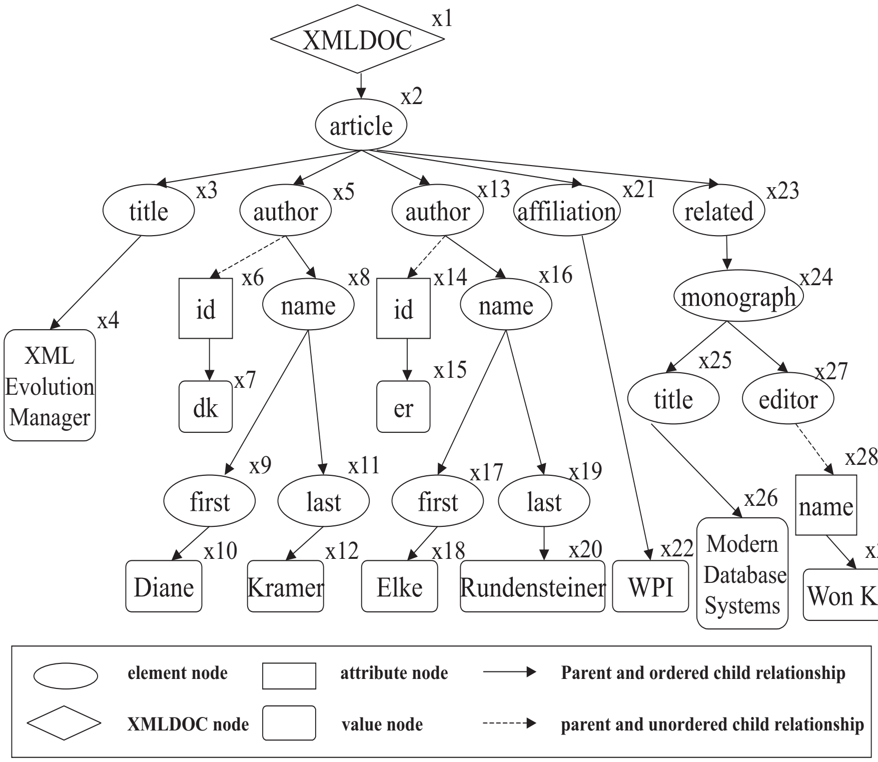


Figure 3: Tree Representation of XML Document in Figure 2

Figure 3 depicts an XML data tree which represents the XML document in Figure 2. For simplicity, we do not mark each node with its label [i, s]. Instead, we use different shapes of the node to distinguish its type "i" and only mark its name "s" inside the node.

Each XML document can be identified by a unique XMLDOC node n with *labelN*(n) =[XMLDOC, "XMLDOC"]. The XMLDOC node's child is an ELEMNODE node which represents the root element instance of the XML document. Each ELEMNODE node n with *labelN*(n) = [ELEMNODE, s] represents an element instance with the name s. For example, <*article*>...</*article*> is represented as the node labeled as [ELEMNODE, "*article*"] (the content between <*article*> and </*article*> is represented by the node's descendant nodes). Each ATTRNODE node n with *labelN*(n) = [ATTRNODE, s] represents an attribute instance with the name s. For a VALNODE n with $labelN(n)$ = [VALNODE, s], s is either a #PCDATA value of an ELEMNODE node or a CDATA value of an ATTRNODE node.

The unordered children relationship exists between an element node and its attribute nodes because attributes are not ordered in a XML document. Subelements of an element are ordered hence the ordered children relationship is used to model this. We also use the terms *children list* and *children set* to refer to the collection of ordered and unordered children respectively.

2.3 The DTD Data Model

A DTD is composed of a set of element definitions. An element definition can in turn contain subelement definitions or attribute definitions or be empty. The structure of an element definition is defined via a *content-model* built out of operators applied to its content particles. *Content particles* are either simple subelement definitions or groups of subelement definitions. Groups may be either sequences indicated by "," (e.g., *a*,*b*) or choices indicated by "|" (e.g., $a|b$) where both

a and b are content particles. For every content particle, the content-model can specify its occurrence in its parent content particle using regular expression operators such as "?" (zero or one occurrence), "*" (zero or more occurrences) or "+" (one or more occurrences). The content model of an element type can be: `EMPTY` (have no content particles); `#PCDATA` (contain only text); `ANY` (can contain any of defined subelements); `MIXED` (can contain both subelements and text).

Definition 2 A **DTD graph** is a quadraple G = (V, $childrenOV$, $childrenUV$, $labelV$). V is the set of vertices in the graph. $childrenOV$ is a function representing the relationship between a vertex and its ordered children, while $childUV$ represents the relationship between a vertex and its unordered children with $childrenOV$, $childrenUV : V \rightarrow V^k$, $k \geq 0$. $labelV$ is a labeling function: $labelV : V \rightarrow I \times (P \times V)^k$, $k \geq 0$ where I is the set of vertex types, i.e., I = {`DTDROOT`, `ELEMDEF`, `ATTRDEF`, `GROUPDEF`, `QUANTDEF`, `PCDATA`, `ANY`}, k is the number of the properties of the vertex, and each pair (p, v) where $p \in P$ and $v \in V$ indicates that the vertex takes value v for property p.

We use *labelV*(v).*Type* to represent the type of vertex v, e.g., `DTDROOT`, `ELEMDEF` and `PCDATA` etc. Vertices of different types can have different sets of properties. Among them, a vertex of type `ELEMDEF`, `GROUPDEF` or `QUANTDEF` is called a *content particle vertex* since it is associated with a content particle. Below we describe the properties of each vertex type. The six vertex types fall into three larger categories, namely, *tag vertex*, *constraint vertex* and *built-in vertex*.

1. **Tag Vertex:**

   (a) **ELEMDEF** (Element Definition Vertex): Each `ELEMDEF` vertex *v* represents an element definition. v has one property, denoted as *labelV*(v).*Name* which represent the name of the element type.

   (b) **ATTRDEF** (Attribute Definition Vertex): Each `ATTRDEF` vertex *v* represents an attribute definition. v has four properties, denoted as *labelV*(v).*Name*, *labelV*(v).*ValType*, *labelV*(v).*Default* and *labelV*(v).*DefaultVal*, which represent the name, value type (e.g., `CDATA`, `ID`, `IDREF`, `IDREFS` etc.), default property (i.e., `#REQUIRED`, `#IMPLIED`, `#FIXED` or with a default value), and default value (if any) of the attribute type respectively.

2. **Constraint Vertex:**

   (a) **GROUPDEF** (Group Definition Vertex): Each `GROUPDEF` vertex *v* has only one property, denoted as *labelV*(v).*GrpType* which specifies how the content particles are grouped in its parent content particle.

      i. if *GrpType* = "," (i.e., `LIST`): the children are grouped by sequence;

      ii. if *GrpType* = "|" (i.e., `CHOICE`): the children are grouped by choice.

   (b) **QUANTDEF** (Quantifier Definition Vertex): Each `QUANTDEF` vertex *v* has only one property, denoted as *labelV*(v).*QuantType* which specifies how many times the content particles occur in its parent content particle.

      i. if *QuantType* = "$*$" (i.e., `STAR`): children are *repeatable* but *not-required*.

      ii. if *QuantType* = "$+$" (i.e., `PLUS`): children are *repeatable* and *required*.

      iii. if *QuantType* = "?" (i.e., `QMARK`): children are neither *repeatable* nor *required*.

3. **Built-in Vertex**:

   (a) `DTDROOT` vertex: The `DTDROOT` vertex is the entry for the DTD graph, i.e., the only vertex in the graph whose indegree is zero. It has a set of children each of which is an element definition vertex. A `DTDROOT` vertex has no properties defined.

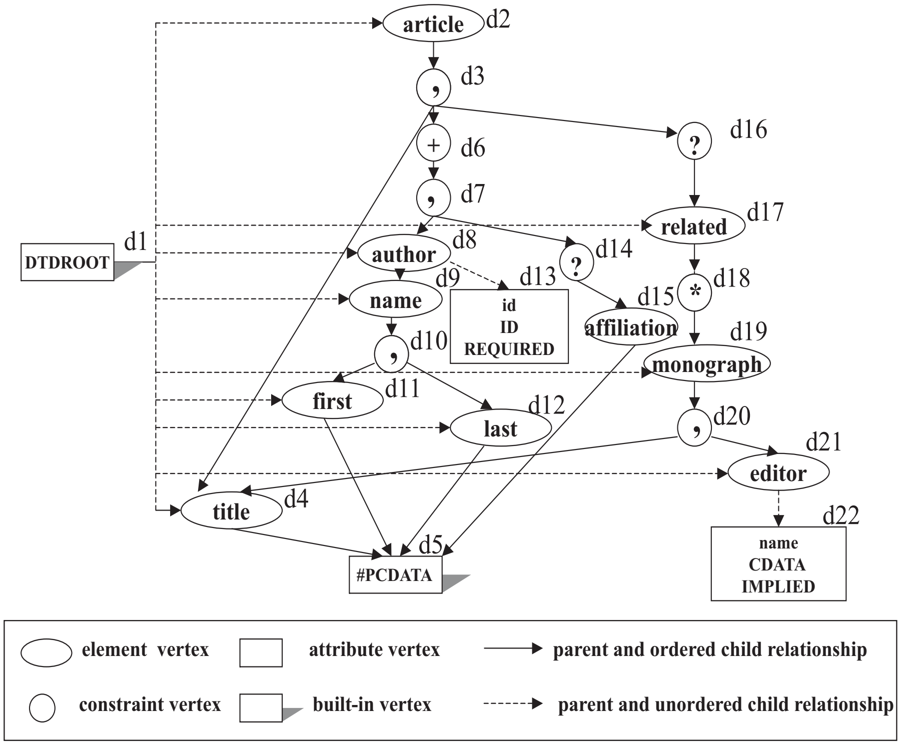


Figure 4: Graph Representation of *Article.dtd* in Figure 1

(b) PCDATA vertex: The PCDATA vertex indicates the content type of its parent, an element definition vertex, is #PCDATA.

(c) ANY vertex: The ANY vertex indicates the content type of its parent, an element definition vertex, is ANY.

Note, if an element vertex does not have any subelement vertices, its content type is EMPTY. Otherwise, its content type is MIXED. Hence for those content types that can be derived from the parent and children relationship, we do not have explicit built-in vertices to express them.

Figure 4 depicts the DTD graph representing the *Article.dtd* in Figure 1. For simplicity, we use different shapes of the vertices to denote their types and the label of a vertex shows the values of all the vertex's properties.

The unordered children relationship exists between an element vertex and its attribute vertices, or between a DTDROOT vertex and its element definition vertices. A content particle vertex is modeled as an ordered child of its parent content particle vertex. We use *children list* and *children set* to refer to ordered and unordered children respectively.

To locate a content particle u within the content model of element p, we define the concept of a *DTD position* denoted by the format of a list of integers [i_1, i_2, ..., i_j, i_{j+1}, ..., i_k]. Each integer is associated with a vertex. A list of such integers is then associated with a path by which u can be reached from p. i_{j+1} ($j > 0$) is associated with a vertex which is the i_{j+1}th child in the children list of the vertex associated with i_j. For i_1, it is associated with the i_1th child of p. For example, in Figure 4, the content particle *related*? (*d16*) defined in *article* (*d2*) is reached through the path [*d3*, *d16*]. *d3* is the first child of *d2* and *d16* is the third child of *d3*, thus the DTD position of *d16* is then [1, 3].

2.4 Relationships between DTD Graph and XML Data Tree

It is required in [38] that a DTD must be deterministic, i.e., an element in the document can match only one occurrence of an element type in a content model. Hence in an XML data tree, each element or attribute instance node is "uniquely typed", i.e., an instance node is bound to a unique path in a DTD graph starting from the `DTDROOT` vertex and ending at either an element or an attribute definition vertex. We therefore define the bi-direction relationship between the XML data tree nodes and the DTD graph vertices.

Given an XML tree T = (N, *childrenON*, *childrenUN*, *labelN*) and a DTD graph G = (V, *childrenOV*, *childrenUV*, *labelV*), we define a function *typeOf*: $N \rightarrow V^i$ (i>1). $\forall\ n \in N$ and n an element or attribute instance node, *typeOf*(n) gives a list [v_1, v_2, ..., v_i, v_{i+1}, ..., v_k] (k≥1).

1. If n is not a root element node, then $k > 1$, and $v_1.v_2...v_i.v_{i+1}...v_k$ is a path in the DTD graph. In the path, v_1 is an element definition vertex that defines the type of n's parent element, v_{i+1} is v_i's child and v_k is the definition vertex that defines n's type.
2. If n is a root element node, it does not have any parent element, thus $k = 1$, and v_1 gives the definition vertex defining n's type.

We call the list [v_1, v_2,..., v_k] a *DTD path list*.

For example, in Figures 3 and 4, *typeOf*(*x3*) = [*d2*, *d3*, *d4*], and *typeOf*(*x23*) = [*d19*, *d20*, *d4*].

Conversely, we define a function as *extent*: $V^k \rightarrow (N^j)^i$ (i, j, k≥1). The input for *extent* is a DTD path list.

1. If v_1 is a `DTDROOT` vertex: v_2 can only be an element definition vertex, and *extent*(v_1, v_2) gives a singleton list which contains only one sublist. The sublist contains all the instance nodes defined by type v_2.
2. If v_1 is a `ELEMEDEF` vertex:
   (a) if v_2 is a `ELEMDEF` or `ATTRDEF` vertex: *extent*(v_1, v_2) gives a singleton list. The only sublist contains all the instance nodes m that satisfy *typeOf*(m) = [*DTDROOT*, v_1, v_2].
   (b) if v_2 is a `GROUPDEF` vertex: *extent*(v_1, v_2) gives a singleton list. The only sublist contains all the instance nodes m that v_2 groups together. m can be of different element or attribute types since v_2 can group elements or attributes of different types together.
   (c) if v_2 is a `QUANTDEF` vertex: *extent*(v_1, v_2) gives a list of sublists each of which contains one occurrence of a group of instances that v_2 quantifies.

For example, in Figures 3 and 4, *extent*(*d1*, *d2*) gives all instance nodes of type *article*, i.e, [[*x2*]]. *extent*(*d2*, *d4*) returns all instance nodes bound with the content particle *title* in *article*, i.e, [[*x3*]]. Moreover, *extent*(*d2*, *d6*) gives the binding of content particle (*author*, *affiliation*?)+ which is composed of two occurrences of groups bound with content particle (*author*, *affiliation*?), i.e., [*x5*], [*x13*, *x21*] respectively. Therefore *extent*(*d2*, *d6*) = [[*x5*], [*x13*, *x21*]].

3 Taxonomy and Semantics of Evolution Primitives

3.1 Overview of the Taxonomy

In this section we present our proposed taxonomy of evolution primitives and define their semantics. Our goal is to provide a set of primitives with the following characteristics:

DTD Operation	Description
createDTDElement(s, t)	Create target element type with name s and content type t
destroyDTDElement()	Destroy target element type
insertDTDElement(e, i, q, v)	Add element type e with quantifier q and default value v at DTD position i to target element type
removeContentParticle(i)	Remove content particle at DTD position i in target element type
changeQuant(i, q)	Change quantifier of content particle at DTD position i in target element type to q
convertToGroup(*start*, *end*, t)	Group content particles from DTD position *start* to *end* in target element type into a group of type t
flattenGroup(i)	Flatten group at DTD position i in target element type
addDTDAttr(s, t, d, v)	Add attribute type with name s with type t, default type d, and default value v to target element type
destroyDTDAttr(s)	Destroy attribute type with name s from target element type

XML Data Operation	Description
createDataElement(e, v)	Create target element node with type e and value v
addDataElement(e, i)	Add element node e at position i in target element node
destroyDataElement()	Destroy target element node
addDataAttr(s, v)	Add an attribute with name s and value v to target element node
destroyDataAttr(s)	Destroy attribute with name s in target element node

Table 2: Taxonomy of DTD and XML Data Change Primitives

- **Complete:** While we aim for a minimal set of primitives, all valid changes to manipulate DTDs and XML data should be specifiable by one or by a sequence of our primitives.
- **Sound:** Every primitive is guaranteed to maintain system integrity in terms of legality of DTD, well-formedness of XML data, and consistency between DTD and XML data. We ensure that the execution of primitives violates neither the invariants nor the constraints in the content model.

The primitives fall into two categories: those pertaining to the DTD, and those pertaining to the XML data. Table 2 gives the complete taxonomy of primitives for DTD and XML data changes. A more detailed explanation of the primitives and examples of their use are given in Section 3.2.

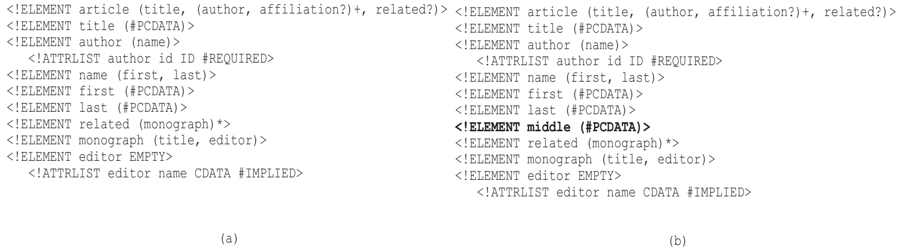
3.2 Details of Change Primitives

In this section, we define the precise syntax and semantics of each DTD and XML change primitive. We assume that the input DTD graph G_1 = (V_1, $childrenOV_1$, $childrenUV_1$, $labelV_1$) is legal and the input XML data tree T_1 = (N_1, $childrenON_1$, $childrenUN_1$, $labelN_1$) is well-formed and valid. To ensure that the targeted output DTD graph G_2 = (V_2, $childrenOV_2$, $childrenUV_2$, $labelV_2$) and XML data tree T_2 = (N_2, $childrenON_2$, $childrenUN_2$, $labelN_2$) remain legal, well-formed and valid after the changes, pre-conditions and post-conditions are enforced on each change primitive. The primitive will not be executed unless the corresponding pre-conditions are satisfied, and changes will not be committed unless the corresponding post-conditions are accomplished.

We now clarify some of the terms we are using in this paper. For a new vertex u, existing vertices v and w with w a descendant of v at DTD position i = [j_1, j_2, ..., j_{k-1}, j_k], if we say "u is *inserted at* DTD position i in v", it means u will be at DTD position i in v after being inserted and w will be the sibling right after u. If we say "u is *inserted above* DTD position i in v", it means u be will be at DTD position i after being inserted and w will now be u's child. If we say "w is *removed* from DTD position i in v", it means the incoming edge from the vertex at DTD position [j_1, ..., j_{k-1}] to w is deleted.

3.2.1 Changes to the DTD

Due to the space limitation, we describe the change operations formally only when necessary. The full formal definitions are in [23]. Each change operation is executed on the target object. And Primitive 1 *createDTDElement* and Primitive 10 *createDataElement* return the new object created through the operation.



```
<!ELEMENT article (title, (author, affiliation?)+, related?)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (name)>
   <!ATTRLIST author id ID #REQUIRED>
<!ELEMENT name (first, last)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT related (monograph)*>
<!ELEMENT monograph (title, editor)>
<!ELEMENT editor EMPTY>
   <!ATTRLIST editor name CDATA #IMPLIED>
```

(a)

```
<!ELEMENT article (title, (author, affiliation?)+, related?)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (name)>
   <!ATTRLIST author id ID #REQUIRED>
<!ELEMENT name (first, last)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT middle (#PCDATA)>
<!ELEMENT related (monograph)*>
<!ELEMENT monograph (title, editor)>
<!ELEMENT editor EMPTY>
   <!ATTRLIST editor name CDATA #IMPLIED>
```

(b)

Figure 5: Results of *createDTDElement* Primitive

For an element definition or built-in vertex, we use its name to represent it since its name is unique in the DTD. Similarly, an *XMLDOC* node in an XML data tree can be represented by its name. For an element instance node, it is represented by an XPath [37] uniquely identifying it (i.e., the XPath can only refer to this single node). Also, a variable can be used to represent an object (i.e., a vertex or a node) returned by some primitives. We use $ as a prefix to distinguish a variable. For example., *a* represents an element definition vertex with type name *a* while $*a* is a variable named a.

Primitive 1: *createDTDElement*

Syntax: *DTDROOT*.createDTDElement(String s, ConType t)

Semantics: Create and return a new non-nesting element definition named s with content type t.

Preconditions: No existing element definition vertex with name s has been defined. That is, $\forall v \in V_1$ and $labelV_1(v).Type$ = ELEMDEF, $label_1(v).Name \neq s$. Also, t must be either *EMPTY* or *#PCDATA*.

Resulting DTD Changes: A new element definition vertex e with name s will be created with content type t, and will be added to the children set of the *DTDROOT* vertex. That is, V_2 = $V_1 \cup e$, $labelV_2(e).Type$ = ELEMDEF, $labelV_2(e).Name$ = s, $childrenUV_2$(*DTDROOT*) = $childrenUV_1$(*DTDROOT*) $\cup$ e, $labelV_2(e).ConType$ = t. $\forall v \in V_1$, we have $childrenOV_2(v)$ = $childrenOV_1(v)$, $childrenUV_2(v)$ = $childrenUV_1(v)$, and $labelV_2(v)$ = $labelV_1(v)$.

Resulting Data Changes: The newly created element type is not a subelement of any other element type yet, i.e., it cannot be reached from any other defined element. We say such an element definition vertex is “dangling”. No instances of u will be created. Therefore, this primitive causes no changes to the XML data.

Example 1 *For the DTD in Figure 5 (a), we create a new element type* middle *to represent the concept of an author's middle name. The command is:*

DTDROOT.*createDTDElement(“middle”,* #PCDATA*).*

This primitive changes the DTD in Figure 5 (a) to the form in Figure 5 (b).

Primitive 2: *destroyDTDElement*

Syntax: *e*.destroyDTDElement()

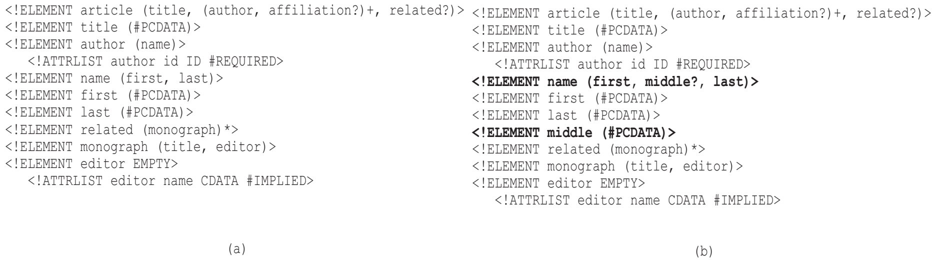
Semantics: Destroy the element definition e.

Preconditions: An element definition vertex e must exist, be non-nesting, i.e., its content model is either *EMPTY* or *#PCDATA*.

Resulting DTD Changes: e will be removed from the children list of the *DTDROOT* vertex and then be destroyed.

Resulting Data Changes: All instance nodes of type e are removed.

Example 2 *For the DTD in Figure 5 (b), we destroy the dangling element definition* middle*. The command is:*



```
<!ELEMENT article (title, (author, affiliation?)+, related?)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (name)>
   <!ATTRLIST author id ID #REQUIRED>
<!ELEMENT name (first, last)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT related (monograph)*>
<!ELEMENT monograph (title, editor)>
<!ELEMENT editor EMPTY>
   <!ATTRLIST editor name CDATA #IMPLIED>
```

(a)

```
<!ELEMENT article (title, (author, affiliation?)+, related?)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (name)>
   <!ATTRLIST author id ID #REQUIRED>
<!ELEMENT name (first, middle?, last)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT middle (#PCDATA)>
<!ELEMENT related (monograph)*>
<!ELEMENT monograph (title, editor)>
<!ELEMENT editor EMPTY>
   <!ATTRLIST editor name CDATA #IMPLIED>
```

(b)

Figure 6: Results of *insertDTDElement* Primitive Operation

middle.*destroyDTDElement().*

This primitive restores the DTD in Figure 5 (b) to the form in Figure 5 (a). Since no instance of middle *exists yet, no data change will be made to the XML document.*

3.2.2 Changes to an Element Type Definition

Primitive 3: *insertDTDElement*

Syntax: p.insertDTDElement(ElemDef e, DTDPosition i, QuantType q, Value v)

Semantics: Insert the element definition e with quantifier q into target element definition p at DTD position i. The default value of the instances correspondingly generated (if any) is v.

Preconditions: There must exist an element definition vertex e. $q \in \{$`STAR`, `PLUS`, `QMARK`, `NONE`$\}$. If q signifies a *required* constraint, $\{$i.e., q = `STAR` or q = `NONE`$\}$ and e is a *PCDATA* element, v must not be null.

Resulting DTD Changes: If q is `NONE`, the element definition vertex e will be added to p at DTD position i. Otherwise, a new quantifier vertex u of type q will be created with e as its only child and u will then be added to p at DTD position i.

Resulting Data Changes: If q signifies a *required* constraint, then for each instance node $m \in$ *extent*(*DTDROOT*, p), a subtree rooted at an instance node n with *typeOf*(n) = [*DTDROOT*, e] will be created based on v (if any) and then inserted below m.

Example 3 *For the DTD in Figure 6 (a), we create the element* middle*, as was done above in Example 1.*

DTDROOT.*createDTDElement("middle",* `PCDATA`*);*

We then insert element middle *with quantifier* `QMARK` *into the target element* name *at DTD position [1, 2]) (i.e., between* first *and* last*). The command is:*

name.*insertDTDElement("middle", [1, 2],* `QMARK`*);*

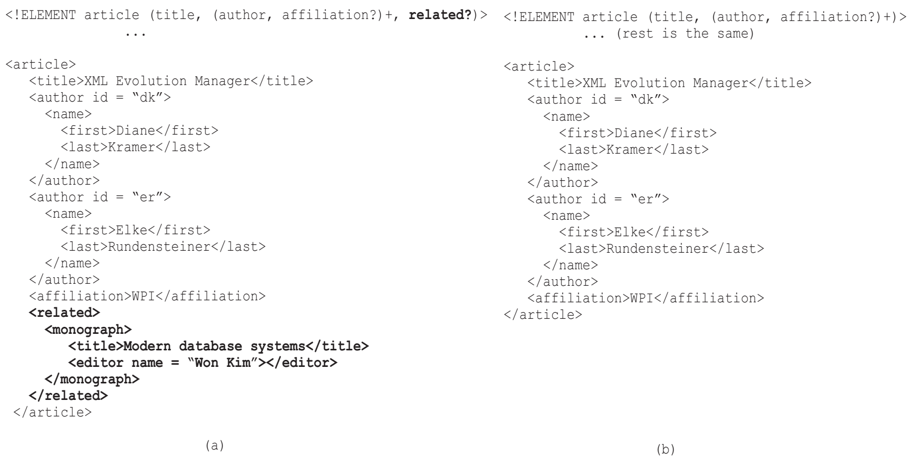
Since the quantifier does not impose a required *constraint, no data changes are required for this operation. These primitives change the DTD in Figure 6 (a) to the form in Figure 6 (b).*

Primitive 4: *removeContentParticle*

Syntax: p.removeContentParticle(DTDPosition i)

Semantics: Remove the content particle at DTD position i in the target element definition p.

Preconditions: There must exist a content particle vertex u at DTD position i in p.



```
<!ELEMENT article (title, (author, affiliation?)+, related?)>
               ...

<article>
   <title>XML Evolution Manager</title>
   <author id = "dk">
     <name>
       <first>Diane</first>
       <last>Kramer</last>
     </name>
   </author>
   <author id = "er">
     <name>
       <first>Elke</first>
       <last>Rundensteiner</last>
     </name>
   </author>
   <affiliation>WPI</affiliation>
   <related>
     <monograph>
        <title>Modern database systems</title>
        <editor name = "Won Kim"></editor>
     </monograph>
   </related>
 </article>
```

(a)

```
<!ELEMENT article (title, (author, affiliation?)+)>
          ... (rest is the same)

<article>
   <title>XML Evolution Manager</title>
   <author id = "dk">
     <name>
       <first>Diane</first>
       <last>Kramer</last>
     </name>
   </author>
   <author id = "er">
     <name>
       <first>Elke</first>
       <last>Rundensteiner</last>
     </name>
   </author>
   <affiliation>WPI</affiliation>
</article>
```

(b)

Figure 7: Results of removeContentParticle Primitive Operation

Resulting DTD Changes: u is removed from the children list of p.

Resulting Data Changes: $\forall n \in extent(p, u)$, all subtrees rooted at n are destroyed.

Example 4 *For the DTD in Figure 7 (a), we remove the content particle* related? *from the element type* article*. The DTD position of* ralated? *is [1, 3]. Thus the command is:*

article.*removeContentParticle(*article*, [1, 3]);*

The primitive changes the DTD in Figure 7 (a) to the form in Figure 7 (b). As for the XML data change, since the only instance of the content particle related? *in* article *is* x23*, the subtree rooted at* x23 *which corresponds to the bold part in the XML document in 7 (a) is removed. The XML document is then changed to the form in 7 (b).*

Primitive 5: *changeQuant*

Syntax: p.changeQuant(DTDPosition i, QuantType q)

Semantics: Change the quantifier for the content particle at DTD position i in the target element definition p to type q.

Preconditions: There must exist a content particle vertex u at position i in p. u must be one of the two following cases. First, u is not a quantifier vertex and u does not have a parent quantifier vertex, i.e., u's quantifier state is NONE. Second, u is a quantifier vertex.

Resulting DTD Changes: If u is not a quantifier vertex without a parent quantifier vertex, then a new quantifier vertex of type q will be inserted above u. If u itself is a quantifier vertex, then u is updated to the new quantifier type q. Especially, if q is NONE, u will be removed.

Resulting Data Changes: The XML data changes required for this primitive depend on the old and new quantifier types. These changes can be summarized using the following three rules:

1. If the old quantifier u' represented a *repeatable* constraint and the new quantifier does not, we must remove the multiple occurrence of the instances that u' quantifies. We adopt a "first kept" policy where only the first occurrence is kept. That is, for list $L = extent(p, u')$, remove all nodes in the sublists in L besides the ones in the first sublist.

2. If the new quantifier u represents a *required* constraint and the old quantifier u' did not, we must check whether there exists any instance of the content particle that u' quantifies. If not, an instance must be created. That is, for each

```
<!ELEMENT article (title, (author, affiliation?)+, related?)>
            ...

<article>
   <title>XML Evolution Manager</title>
   <author id = "dk">
     <name>
       <first>Diane</first>
       <last>Kramer</last>
     </name>
   </author>
   <author id = "er">
     <name>
       <first>Elke</first>
       <last>Rundensteiner</last>
     </name>
   </author>
   <affiliation>WPI</affiliation>
   <related>
     <monograph>
        <title>Modern database systems</title>
        <editor name = "Won Kim"></editor>
     </monograph>
   </related>
 </article>
```

(a)

```
<!ELEMENT article (title, (author, affiliation?), related?)>
          ... (rest is the same)

<article>
   <title>XML Evolution Manager</title>
   <author id = "dk">
     <name>
       <first>Diane</first>
       <last>Kramer</last>
     </name>
   </author>
   <related>
     <monograph>
        <title>Modern database systems</title>
        <editor name = "Won Kim"></editor>
     </monograph>
   </related>
</article>
```

(b)

Figure 8: Results of changeQuant Primitive Operation

instance node $n \in extent$(*DTDROOT*, p), we check whether there exists a sublist $L' \in extent(p, u')$ such that $\forall m \in L'$, $m {\in} childrenON(n)$. If not, add a new default subtree which represents an instance of the content particle that u' quantifies.

3. The remaining combinations of old and new quantifiers such as *not-repeatable* becomes *repeatable*, or *required* becomes *not-required* cause no changes to the XML data.

Example 5 *For the DTD in Figure 8 (a), we change the quantifier of subelement type* author *in the parent element type* article *from* `PLUS` *to* `NONE`*. The DTD position of the content particle* author *in* article *is [1, 2]. Thus the command is:*

article.*changeQuant([1, 2],* `NONE`*);*

*This primitive, in addition to changing content particle (*author, affiliation?*)+) to (*author, affiliation?*), also deletes some instances of (*author, affiliation?*) according to rule 1. As illustrated in Section 2.4,* extent*(*d2, d6*) = [[*x5*], [*x13, x21*]], therefore only the nodes in the first sublist [*x5*] are kept while the nodes in the other sublists [*x13, x21*] are all deleted. This primitive changes the DTD and XML document in Figure 8 (a) to the forms in Figure 8 (b).*

Primitive 6: *convertToGroup*

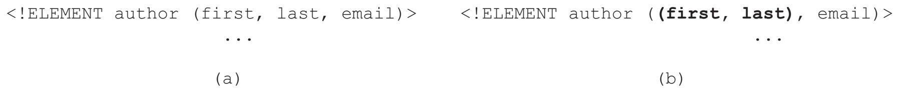
Syntax: p.convertToGroup(DTDPosition $start$, DTDPosition end, GroupType t)

Semantics: Group together a sequence of content particles, whose DTD positions range from $start$ to end in the target element definition p, with group type t.

Preconditions: $start$ and end must be at the same level in p, i.e., the content particles to be grouped must be siblings. Also, $t \in$\{`LIST`, `CHOICE`\}.

Resulting DTD Changes: We create a new group definition vertex u, move a set of children G whose DTD positions are falling into the range ($start$, end) in p to be u's children, and then insert u into p at DTD position $start$.

Resulting Data Changes: Since this primitive only changes the hierarchical organization of the content particle vertices, it does not cause any change to the XML data.



```
<!ELEMENT author (first, last, email)>    <!ELEMENT author ((first, last), email)>
                  ...                                          ...

                  (a)                                          (b)
```

Figure 9: Results of convertToGroup Primitive Operation

Example 6 *For the DTD in Figure 9 (a), subelements of* author, first, last *and* email*, are all at the same hierarchical level. We can group content particles* first *and* last *into a sequence list group. The group implies that* first *and* last *are more semantically coupled (they together convey the information of* name*). The command is:*

author.*convertToGroup([1, 1], [1, 2],* `LIST`*);*

This primitive changes the DTD in Figure 9 (a) to the form in Figure 9 (b). There is no data change caused by this primitive.

Primitive 7: *flattenGroup*
Syntax: p.flattenGroup(DTDPosition i)
Semantics: Flatten a group of content particles at DTD position i in the target element definition p.
Preconditions: There must exist a group definition vertex u at DTD position i in p.
Resulting DTD Changes: Move all u's children G to be p's children and then remove u in p.
Resulting Data Changes: Similar to Primitive 6, this primitive only changes the hierarchical organization of the content particle vertices, it does not cause any change to the XML data.

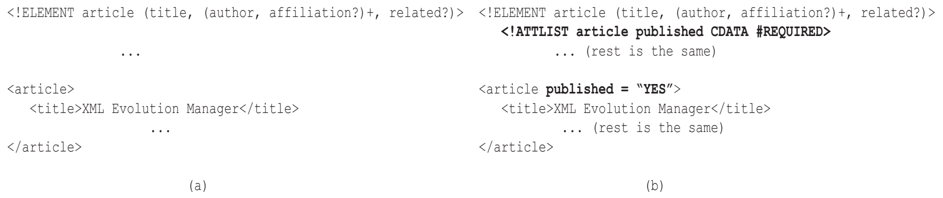
Example 7 *For the DTD in Figure 9 (b), we flatten the list group composed of* first *and* last *to restore the* article *definition to the DTD in Figure 9 (a). The DTD position of the content particle (*first, last*) in* author *is [1, 1]. Thus the command is:*

author.*flattenGroup([1, 1]);*

Primitive 8: *addDTDAttr*
Syntax: p.addDTDAttr(String s, AttrType t, DefType d, String v)
Semantics: A new attribute definition with name s, attribute type t, default type d, and default value v will be defined with the target element definition p.
Preconditions: No attribute with name s has been defined in p. $t \in \{$`CDATA`, `CHOICE`, `HREF`, `ID`, `IDREF`, `IDREFS`, `NMTOKEN`$\}$. $d \in \{$`#REQUIRED`, `#IMPLIED`, `#FIXED`, `#DEFAULT`$\}$. If d is not `#IMPLIED`, the default value v must not be null.
Resulting DTD Changes: A new attribute definition vertex u will be created with the specified properties and added to the attribute children set of p.
Resulting Data Changes: If the default type t is `#REQUIRED`, $\forall n \in$*extent*(*DTDROOT*, p), a new attribute instance node m will be created with default value v and added to n's attribute children set.

Example 8 *For the DTD in Figure 10 (a), we add a new attribute type* published *to the element type* article *to indicate whether this article has published or not. The command is:*

article.*addDTDAttr("published",* `CDATA`*,* `#REQUIRED`*, "YES");*



```
<!ELEMENT article (title, (author, affiliation?)+, related?)>

              ...

<article>
   <title>XML Evolution Manager</title>
                  ...
</article>
```

(a)

```
<!ELEMENT article (title, (author, affiliation?)+, related?)>
   <!ATTLIST article published CDATA #REQUIRED>
          ... (rest is the same)

<article published = "YES">
   <title>XML Evolution Manager</title>
           ... (rest is the same)
</article>
```

(b)

Figure 10: Results of addDTDAttr Primitive Operation

This primitive changes the DTD in Figure 10 (a) to the form in Figure 10 (b). Moreover, since the attribute is required to be present, an instance of the attribute would be created with the provided default value "YES". Hence the XML document shown in Figure 10 (a) is changed to the form in Figure 10 (b).

Primitive 9: *destroyDTDAttr*
Syntax: *p*.destroyDTDAttr(String s)
Semantics: An attribute definition named s defined in the target element definition p will be deleted.
Preconditions: An attribute definition vertex u named s must exist in the attribute children set in p.
Resulting DTD Changes: u will be destroyed.
Resulting Data Changes: For all $n \in$ *extent*(*DTDROOT*, u), n is destroyed.

Example 9 *We delete the attribute type* published *from the element type* article *shown in the DTD in Figure 10 (a). The command is:*

article.*destroyDTDAttr("published");*

This primitive restores both the DTD and the XML data in Figure 10 (b) to the forms in Figure 10 (a).

3.2.3 Changes to the XML Data

In our work, schema is the first-class citizen. This means that a DTD cannot be changed by any data change operation while a DTD change can imply some data changes, i.e., data changes may be caused due to update propagation during the DTD change without being explicitly specified. If users mean to perform some data changes that would result in an XML document becoming inconsistent with the current DTD, they have to explicitly perform the appropriate DTD change primitives first. For example, if users want to delete an subelement instance node which is however required to exist in its parent's contents, a *changeQuant* primitive may be performed to change the quantifier property of this subelement type from *required to* not-required. The data change is then allowed to happen.
Primitive 10: *createDataElement*
Syntax: *XMLDOC*.createDataElement(ElemDef e, DataEleVal v)
Semantics: Create and return a new element of type e with value v.
Preconditions: v must be a valid value for element type e. For example, if the content model of e is of type `#PCDATA`, v must be a legal string that can serve as a `PCDATA` value. If the content model of e is of type `EMPTY`, v must be null. Especially when the content model of e is of type `MIXED`, v must be a list of nodes each of which represents a subelement of the to-be-created element.
Resulting Data Changes: A new data element instance of type e will be created. However it is "dangling" in the XML data tree in the sense that it is only reachable from the *XMLDOC* node rather than from any other element node in the

XML tree. In other words, if the XML data tree is dumped into an XML text file, this newly created element is not visible in the XML text file. Only when this element instance is added to the XML data tree using the *addDataElement* primitive (Primitive 11), is it part of the XML document.

Example 10 *We first create new element instances of type* title *and* editor. *Variables n_1 and n_2 are used to represent them respectively. We then create an element instance n_3 of type* monograph *and specify its value as [n_1, n_2] which means that the element instance n_1 is its first subelement and n_2 is its second subelement. The commands are:*

$$n_1$ = XMLDOC.*createDataElement(*title*, "XML");*
$$n_2$ = XMLDOC.*createDataElement(*editor*, "W3C");*
$$n_3$ = XMLDOC.*createDataElement(*monograph*, [n_1, n_2]);*

These primitive operations cause no change to the DTD and XML document visible outside.

Primitive 11: *addDataElement*
Syntax: n.addDataElement(ElemDef e, DataPosition i)
Semantics: Add a new element e to be the i_{th} subelement of element n.
Preconditions: A new element instance is allowed to be added only in two cases. In case 1, the type of the new element instance is a repeatable content particle. In case 2, the type of the new element instance is an optional content particle and no instance of this content particle exists before.
Resulting Data Changes: The element instance e will be added to the children list of n as the i^{th} child.

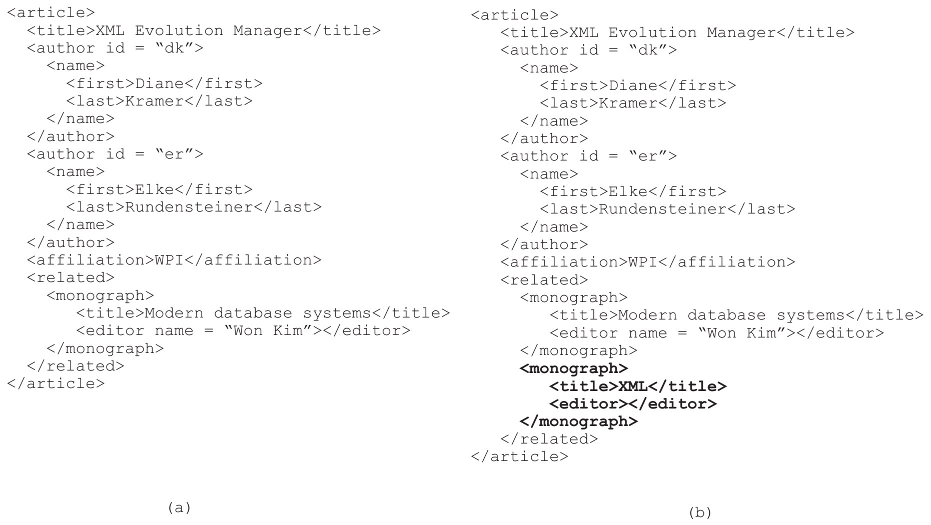
Example 11 *For the XML document in Figure 11 (a), we create a new element instance of type* monograph *and then add it as the second child of element* article/related. *The commands are:*

$$n_1$ = XMLDOC.*createDataElement(*title*, "XML");*
$$n_2$ = XMLDOC.*createDataElement(*editor*, null);*
$$n_3$ = XMLDOC.*createDataElement(*monograph*, [n_1, n_2]);*
article/related.*addElement($$n_3$, 2);*

These primitives cause no change to the DTD. They change the XML document in Figure 11 (a) to the document in Figure 11 (b).

Primitive 12: *destroyDataElement*
Syntax: *n*.destroyDataElement()
Semantics: Destroy the target element node n.
Preconditions: The type of the target element node n must be a *not-required* content particle in its parent element, i.e., the quantifier of the type is either `QMARK` or `STAR`.
Resulting Data Changes: The element instance node e will be removed from the children list of the parent element node n.

Example 12 *For the XML in Figure 11 (b), we remove the second* related *element. The command is:*



```
<article>
  <title>XML Evolution Manager</title>
  <author id = “dk”>
    <name>
      <first>Diane</first>
      <last>Kramer</last>
    </name>
  </author>
  <author id = “er”>
    <name>
      <first>Elke</first>
      <last>Rundensteiner</last>
    </name>
  </author>
  <affiliation>WPI</affiliation>
  <related>
    <monograph>
       <title>Modern database systems</title>
       <editor name = “Won Kim”></editor>
    </monograph>
  </related>
</article>
```

(a)

```
<article>
   <title>XML Evolution Manager</title>
   <author id = “dk”>
     <name>
       <first>Diane</first>
       <last>Kramer</last>
     </name>
   </author>
   <author id = “er”>
     <name>
       <first>Elke</first>
       <last>Rundensteiner</last>
     </name>
   </author>
   <affiliation>WPI</affiliation>
   <related>
     <monograph>
        <title>Modern database systems</title>
        <editor name = “Won Kim”></editor>
     </monograph>
     <monograph>
        <title>XML</title>
        <editor></editor>
     </monograph>
   </related>
</article>
```

(b)

Figure 11: Results of addDataElement Primitive Operation

article/related/monograph[2].*destroyDataElement();*

This primitive causes no changes to the DTD, but restores the XML document in Figure 11 (b) to the document in Figure 11 (a).

Primitive 13: *addDataAttr*

Syntax: n.addDataAttr(String s, String v)

Semantics: An attribute instance with name s and value v will be created within the element n.

Preconditions: An attribute definition vertex u named s must have been defined in the element type *typeOf(n)*. The default type of u must be *#IMPLIED* and no instance of u exists in the attribute children set of n yet.

Resulting Data Changes: A new attribute instance of type u, with value v, will be created and added to the attribute children set of n.

Example 13 *For the XML in Figure 12 (a), we first add an new attribute definition* primary *to the element type* author *to indicate whether an author is the primary author or not. We then use the* addDataAttr *primitive to add attribute values for an* author *element. The commands are:*

author.*addDTDAttr(“primary”,* `CDATA`*,* `#IMPLIED`*, null);*
article/author*[1].addDataAttr(“primary”, “YES”);*
article/author*[2].addDataAttr(“primary”, “NO”);*

Thess primitives change the DTD and XML document in Figure 12 (a) to the DTD and XML document in Figure 12 (b).

Primitive 14: *destroyDataAttr*

Syntax: n.destroyDataAttr(String s)

Semantics: The attribute instance with name s within the element instance n will be deleted.

```
<!ELEMENT author (name)>
  <!ATTLIST author id ID #REQUIRED>

             ...

<article>
   <title>XML Evolution Manager</title>
   <author id = “dk”>
     <name>
       <first>Diane</first>
       <last>Kramer</last>
     </name>
   </author>
   <author id = “er”>
     <name>
       <first>Elke</first>
       <last>Rundensteiner</last>
     </name>
   </author>
          ...
</article>
```

(a)

```
<!ELEMENT author (name)>
  <!ATTLIST author id ID #REQUIRED>
  <!ATTLIST author primary CDATA #IMPLIED>
          ... (rest is the same)

<article>
   <title>XML Evolution Manager</title>
   <author id = “dk”, primary = “YES”>
     <name>
       <first>Diane</first>
       <last>Kramer</last>
     </name>
   </author>
   <author id = “er”, primary = “NO”>
     <name>
       <first>Elke</first>
       <last>Rundensteiner</last>
     </name>
   </author>
          ... (rest is the same)
</article>
```

(b)

Figure 12: Results of addDataAttr Primitive Operation

Operation	Description	Taxonomy Equivalent
create-ver	Creates new dangling vertex	1, 6, 8
add-edge	Adds an edge between two vertices	3, 6, 8
delete-ver	Deletes vertex with zero out-degree and removes all incoming edges	2, 7, 9
remove-edge	Removes the edge between two vertices	4, 7, 9

Table 3: The DTD Graph Operations.

Preconditions: An attribute definition vertex u with name s must exist in the element type of instance node n. The default type of u must not be *#REQUIRED*, since a required attribute cannot be deleted.
Resulting Data Changes: The attribute instance with name s in n will be destroyed.

Example 14 *For the XML in Figure 12 (b), we destroy all the attribute instances of* primary. *The command is:*

article*[2]*/author*[1].destroyDataAttr(“primary”);*
article*[1]*/author*[2].destroyDataAttr(“primary”);*

Since the default type of the attribute type primary *in* author *is #IMPLIED, i.e., em primary is not required to be present, this primitive is allowed to be executed. It causes no changes to the DTD, but restores the XML document in Figure 12 (b) to the form in Figure 12 (a).*

4 Discussion of the Change Taxonomy

4.1 Completeness of DTD Change Operations

In this section we discuss the set of change primitives in Section 3.1 supports all possible types of DTD changes, i.e., the primitives are complete. The proof given here has its basis in the completeness proof for the evolution taxonomy of Orion [2].

With the DTD graph we focus primarily on manipulations of vertices and directed edges between parent and children vertices. We prove that every legal DTD graph operation is achievable using a set of graph operations. The semantics of the graph operations are shown in columns 1 and 2 in Table 3. Our taxonomy equivalents of the general graph operations are given in column 3. If a DTD graph operation is a combination of multiple general graph operations, we list the DTD graph operation in multiple cells in the third column.

Lemma 1 *For any given DTD graph G, there is a finite sequence of* $\{$delete-ver$\}$ *that can reduce G to another DTD graph G' with only a* DTDROOT *vertex.*

Lemma 2 *Given a DTD graph G' with only a* DTDROOT *vertex, there is a finite sequence of operations* $\{$create-ver, add-edge$\}$ *that generates any desired DTD graph G'' from G'.*

Theorem 1 *Given two arbitrary DTD graphs G and G'', there is a finite sequence of* $\{$delete-ver, create-ver, add-edge$\}$ *that can transform G to G''.*

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

Theorem 2 *Given two arbitrary DTD graphs G and G'', there is a finite sequence of DTD change operations shown in Table 3 that can transform G to G''.*

Proof: The set of operations $\{$*create-ver, add-edge, delete-ver*$\}$ all have equivalent operations in the DTD change taxonomy. Hence the completeness of this set of operations is given from Theorem 1.

4.2 Soundness of Change Primitives

A taxonomy of XML and DTD change primitives is sound if the following properties hold true:

- Every operation on a legal input DTD graph produces a legal output DTD graph, and every operation on a well-formed input XML tree produces a well-formed output XML tree (legality and well-formedness criteria).
- Every operation on a valid input XML tree produces a new valid output XML tree (validity criteria).
- Every operation on an input DTD graph which has an associated valid input XML tree produces a valid output XML tree (consistency criteria).

A formal proof of soundness would be rather laborious, requiring detailed proof steps to demonstrate that each of the above properties holds for each defined primitive. Instead, we illustrate below proofs for these properties for a few of these operations. Other proofs could be done similarly.

Legality and Well-formedness. For example, let us consider the *createDTDElement* primitive, an operation which makes only changes to the input DTD graph. Since the original DTD is legal and the newly added element type is dangling (i.e., it is independent from any other element types), the only violation this primitive can bring is duplicate element names. Our pre-condition checking mechanism requires that whether an element type with the same name already exists must be checked. Only when it finds no duplicate element name, the primitive is allowed to be executed. This thus prevents an illegal DTD.

Validity. Let us consider the *addDataElement* primitive, an operation which makes a change to an XML data tree. Prior to executing this primitive, the pre-condition checking mechanism will check whether the element instance to be added is allowed at the requested position as a subelement of the specified element instance. Thus, a primitive passing this checking ensures that the changed XML tree will still conform to its DTD.

Consistency. Let us consider the *removeContentParticle* primitive, an operation which explicitly changes the DTD and implicitly changes the XML data. When pre-conditions are satisfied, we remove the content particle definition from the

XML Concept	Representation in XML Tree	OO Concept
element	element instance node	class instance
attribute	attribute instance node	member variable in class instance
nested structures	edge	member variable in class instance

Table 4: Mapping from XML to OO

content model of the specified parent element definition, i.e., remove the directed edge between the parent and the content particle vertices. If we stopped at this point, we would have a legal DTD graph and a well-formed XML tree, but the XML tree would no longer be consistent with the output DTD graph. We therefore continue to make appropriate changes to the XML trees. We must now remove all corresponding instance nodes to achieve the consistency.

The primitive definitions in Section 3.1 specify precisely when a change to a DTD also requires a change to the XML data in order to maintain consistency via post-conditions. Since any given change will either be rejected due to the pre-conditions not being satisfied, or will occur in both the DTD and the XML data when required, and since we could demonstrate one by one that all of our operations fulfill these requirements, we conclude that our taxonomy of combined DTD and XML change primitives is sound.

5 XEM Prototype System

To verify the feasibility of our approach, we have implemented a working prototype system for XML evolution management, *XEM-Tool*[1]. In this section we first present our system design and overall architecture. Next we discuss our mapping model between XML and the underlying storage system.

5.1 Mapping XML Data Model to OO Data Model

We use an Object Oriented (OO) approach for *XEM-Tool* because the OO model is a data model closer to the XML data model due to its hierarchical structures. Table 4 describes the mapping strategy that we have used to map data in the XML format to the OO format. Basically, an element type is mapped to a class. The attribute type defined in an element type e is mapped to a member variable defined in the class that models e. The relationship between one element and its subelements is modeled by a member variable named *children* which is implemented as a Java vector. Each object in this vector refers to an instance of a class which is mapped from the subelement type. Based on this mechanism, a set of class definitions can be defined given a DTD. We call such classes *application classes*.

Table 5 shows part of the schema of the application classes when processing the *Article.dtd* shown in Figure 1. The first column shows the element type in the DTD. The second column shows the name of the object class generated for the given element type in the first column. The third column shows the names and types of the class' member variables mapped from the attributes defined within the element type . For each generated class definition, its name has a prefix "D" followed by a number indicating the identifier of the DTD for the purpose of managing multiple DTDs. In this example, the identifier of *Article.dtd* is 1.

5.2 *XEM-Tool* Architecture

We use Excelon Inc.'s PSE Pro [30], a lightweight Java object database system, as the underlying persistent storage system. PSE Pro provides a object repository and a schema repository which manages the Java objects and schema information (i.e., the class definitions) respectively. The PSE Pro system has been extended by schema and data evolution functionalities added by our previous project, SERF [12]. SERF supports updating the object repository and schema repository, for

[1] A preliminary version, *ReWeb*, has been demonstrated at ACM SIGMOD 2000.

DTD Element Type	Class Name	Member Variables Mapped from DTD Attributes
article	D1_article	none
title	D1_title	none
author	D1_author	name=“id”, type=String
name	D1_name	none
editor	D1_editor	name=“name”, type=String

Table 5: Application Class Definitions Mapped from DTD

example creating or deleting a class into or from the schema repository and adding or deleting an attribute to or from to a class definition at run-time. Figure 13 depicts the architecture of the XML Evolution Management Prototype system (*XEM-Tool*). The main modules of the *XEM-Tool* system architecture include the following:

- The *DTD Manager* takes DTDs as the input and supports:

  1. converting the DTDs to DTD graphs and managing the information of DTD graphs via the *DTD graph manager*;
  2. generating necessary class definitions of application classes via the *application class definition generator*.
  3. managing the DTD graphs, such as querying and modifying the DTD graphs.

- The *XML Document Manager* takes an XML document as its input and supports:

  1. converting the XML data to object instances of application classes and managing them via the *application class instance manager*;
  2. managing a bi-directional relationship between DTDs and XML data via the *extent manager* as described in Section 2.4. The *extent manager* is able to look up all the application class objects representing the XML data instances of a given DTD path list. Conversely, it can look up the DTD path list (refer to Section 2.4) given an application class object which corresponds to an XML instance node.
  3. regenerating XML documents from the stored instances of application classes via the *XML regenerator*.

- The XML Evolution Manger supports executing the XEM operations defined in Section 3.1 via the *Primitive Executor*.

DTDs and XML documents are loaded in the object repository in PSE Pro by the DTD manager and XML document manager respectively. Once a change primitive is submitted, the *Primitive Executor* interacts with the DTD manager and in some cases also the XML Document manager to check the pre-conditions. If the primitive passes the pre-condition checking, the OO evolution functionalities will be invoked to carry on the desired changes. Some changes are performed only on the underneath object repository while some others are performed on the schema repository as well. Further details will be discussed in Section 6.2.

6 Experimental Study

6.1 Experimental Set Up and Data Sets

We have conducted a series of experiments comparing the time needed to perform incremental updates versus reloading the updated XML documents from scratch. The execution platform is Microsoft Windows NT 4.0 with service pack 6, Intel Pentium II 433MHZ and 128M memory. We selected the set of Shakespeare’s plays [3] as the data set for our experiments. Some statistics about the Shakespeare files are as follows:

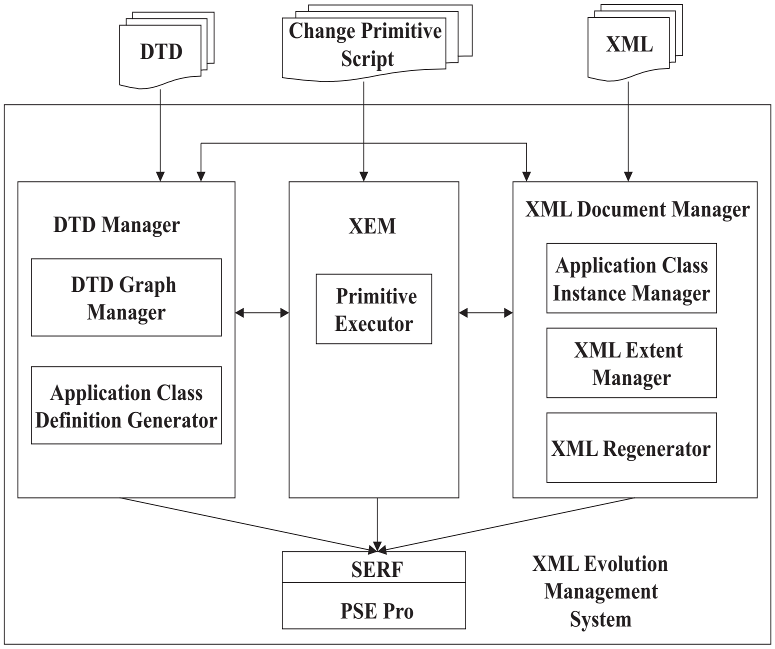


Figure 13: Architecture of *XEM-Tool* System

- 1 DTD with 21 element definitions
- 37 XML data files, one play per file, all conforming to the same DTD
- Smallest data file is 141,345 bytes long, and contains 3133 Elements
- Largest data file is 288,735 bytes long, and contains 6600 Elements
- Average data file is 213,449 bytes long, and contains 4840 Elements.

Since the original Shakespeare DTD did not contain any attribute definitions, we have added some attributes in order to be able to test our primitives that deal with attributes.

6.2 Comparing Time Efficiency of Each Primitive

In our implementation, the XML data change primitives only lead to object changes in the backend OO storage system. However the DTD change primitive operations can be grouped into the following categories based on the types of changes they lead to in the backend OO storage system.

1. For DTD changes not implying any XML data change:
   (a) DTD changes leading to only OO object changes
   (b) DTD changes leading to both OO object changes and application class definition changes (i.e., OO schema changes)
2. For DTD changes implying XML data changes:
   (a) DTD changes leading to only OO object changes

ID	Primitive Name	Time (s)	ID	Primitive Name	Time
1	createDTDElement	2.181	2	destroyDTDElement	0.063
3	insertDTDElement	0.141	4	removeContentParticle	0.122
5	changeQuant	0.004	6	convertToGroup	0.006
7	flattenGroup	0.005	8	addDTDAttr	8.421
9	destroyDTDAttr	7.014			

Table 6: Execution Times for DTD Primitives

(b) DTD changes leading to both OO object changes and application class definition changes

The fact that some DTD changes lead to OO schema changes while some do not is due to our mapping mechanisms. For example, in our mapping mechanisms, children content particles are stored in a Java Vector, while attributes are stored as member variables in the associated class. In the former case, a change to an element's children content particles, e.g., *removeContentParticle*, is not an OO schema change, since we are not changing the definition of the application class mapped from the element. In the latter case, on the other hand, a change which adds or removes an attribute, i.e., *addDTDAttr* and *destroyDTDAttr*, does correspond to a schema change in which a member variable is added into or removed from the definition of the associated application class.

The experiment examines each of the evolution primitives individually. The purpose of this experiment was to give an intuition how much actual change and the performance overhead is caused, by each single change primitive. This experiment was run on 15 XML data files, and each operation was run ten times for accuracy. The results in Table 6 show the averages of the ten runs for each operation.

Some OO schema change is time-consuming in that it requires recompilation of the changed class definitions. In Table 6, we can see some DTD change primitives leading to OO schema changes such as *createDTDElement*, *addDTDAttr* and *destroyDTDAttr* take significantly more time than other DTD change primitives. Note though primitive *destroyDTDElement* leads to OO schema changes as well, it does not take as long as the other three DTD change primitives leading OO schema change. This is because while executing *destroyDTDElement*, the system simply deletes the definition file of the associated application class without requiring a time-consuming recompilation.

6.3 Incremental Update versus Reloading from Scratch

We have tested two DTD change primitives *insertDTDElement* and *addDTDAttr*. Both of them involve implied data changes besides the explicitly specified DTD change. However, due to the mapping mechanism, the first change primitive does not lead to any OO schema evolution while the second one does. And we randomly choose a target DTD element. In our data set, approximately 17.5% of the total amount of data loaded is affected on average by the execution of each primitive operation.

Figures 14 and 15 compare the efficiency of incremental change versus reloading for the two primitives respectively. It is obvious doing an incremental change gains over reloading from scratch. The reason that *addDTDAttr* gains not as much as *insertDTDSubElement* lies in the mapping mechanism we are using. For *insertDTDSubElement*, the definition of the class mapped from the target parent element type remains the same. This is because the new element and subelement relationship is captured in the data content of member variable "children" rather in the class definition itself. However, for *addDTDAttr*, the definition of the class mapped from the target element type is changed. A new member variable is added to represent the newly added attribute. Thus the class definition needs to be recompiled, which is an expensive time consuming process.

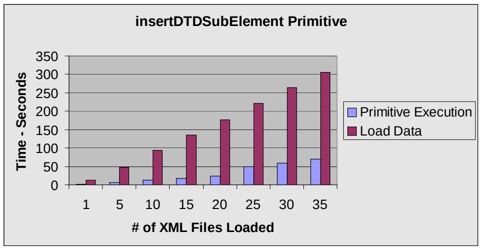


Figure 14: Incremental Updates *insertDTDElement* Vs. Complete Reload of Data

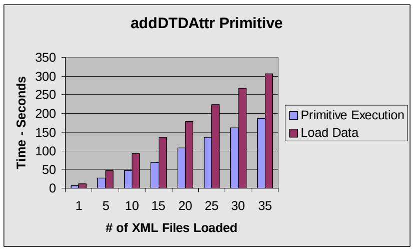


Figure 15: Incremental Updates *addDTDAttr* Vs. Complete Reload of Data

7 Related Work

XML Management Tools. Since XML is primarily used as a data exchange format on the World Wide Web, many research projects dealing with XML have focused on web site management [26, 13, 16, 9]. These projects attempt to alleviate difficulties associated with managing large amounts of data contained in web sites by representing web pages as XML documents. Although our XEM work does not focus on web site management, research into these projects proved useful in understanding storage and manipulation of XML documents.

Other research on XML focuses on its semi-structured nature [8, 7, 15]. In dealing with semi-structured data, some projects either totally ignore the schema, 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. For example, Object Exchange Model [31] represents semi-structured data, similar in nature to XML, without any associated DTD definition. DOEM [8] is further proposed as a model to represent changes in semi-structured data via temporal annotations. [34] proposes extensions to XQuery [40] to support XML updating. However, all these approaches only deals with the changes at the data level and they all are schema-blind.

Some XML tools have focused on various language formats as a mechanism for manipulating XML data. For example, Extensible Stylesheet Language Transformations (XSLT) [18] 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) [21] 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 the DTD into account. So neither XSLT nor Lexus can serve in scenarios where a schema or structure is required.

DTD has limited power to express integrity constraints. For example, it is not sufficient to express keys and foreign keys. [15, 14, 6] proposes a model of constraints for XML. The model can capture relational constraints, object-oriented models (with object identity and scoped reference), and the ID/IDREF mechanism of DTDs. In our system, we only focus on the inherent constraints in the DTD model, while extensions to also take care of these constructs remain to be investigated.

Schema Evolution. Many traditional database projects have focused on the issue of schema evolution [2, 5, 33, 8, 42], where the main goal is to develop mechanisms to change not only the schema but also the underlying objects such that they conform to the modified schema. This issue was for the first time tackled for the XML model in our current XEM project. Most commercial database systems for RDB or OODB today [20, 35, 30, 36] provide support for the re-structuring of the application schema by means of a fixed set of simple evolution primitives, as does our XEM system. Recent work has been done to focus on the issues of supporting more complex schema evolution operations for OODBs [4, 24]. These allow the user to string together several primitives to form higher level yet still specific change transformations. Finally, SERF

[12, 11] is a template-based, extensible schema evolution framework developed at WPI that allows complex user-defined schema transformations in a flexible yet secure fashion.

XML and Database Systems. A number of projects and tools have emerged to map XML and similar semi-structured data formats to traditional database systems. [17] studies storing and querying XML data using a relational database management system (RDBMS). Both [22] and [25] investigate semi-structured data in relational databases, while [10, 28] studies SGML (the predecessor of XML) storage in an object-oriented database management system (OODBMS). Oracle's XML SQL Utility (XSU) [27] and IBM's DB2 XML Extender [19] are well-known commercial relational database 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 manually make the change on the relational schema or data. 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, the database research group has developed the *Clock* system [41] that synchronizes internal relational storage with external XML documents. This *Clock* system deals with basic XML data updates only and does not handle XML schema changes.

8 Conclusions

Summary. In our work on the XEM project, we make a number of important contributions in the area of XML data management, including the first approach for addressing evolution in an XML context. We show the motivation behind the need for such support, while identifying the lack of existing support in current XML data management systems. We propose a taxonomy of XML evolution primitives which includes both schema and data updates to fill this gap. We identify various forms of system integrity which a sound XML management system must maintain during evolution. These include the *well-formedness* of DTDs and XML documents, which must conform to the standard language format; the *consistency* of XML documents in terms of their invariants; and the *validity* of XML documents with respect to the constraints specified in the corresponding DTD. We show that our proposed change taxonomy is *complete* in that all valid desired transformations are possible using our primitives, and *sound* in its maintenance of system integrity.

We verify the feasibility of our approach by developing a working *XEM-Tool* prototype implementation using the Java programming language and an underlying object-oriented database. Our prototype provides automated XML evolution management facilities which are superior to making manual edits. We conduct experimental studies to verify the correct execution of the primitive operations within our prototype system. We also present a performance analysis which shows that incremental updating using the primitives is more efficient than reloading data from scratch, which would be necessary using other current XML management tools.

Future Work. During the course of our research on the XEM project, a number of issues arose which were beyond the scope of our project, which however present interesting issues of future study. Here we present new research directions which could be undertaken to continue this work.

- **Model Mapping:** Our XEM-Tool implementation is currently tied to the PSE Object Store database. A more generic storage independent XEM middleware would be a more flexible solution to develop.
- **Versioning:** If the XEM-Tool system were modified such that changes were made to a new copy of DTDs and XML documents, rather than "in place", or if deltas were stored which could be applied to old documents to produce new

ones, our system could be used to provide revision control and version management services.

- **Embedding into XML Query Languages:** Currently, primitives are implemented as APIs. However the primitives can be embedded into XML query language, say XQuery [40]. This would enable users to declaratively specify desired changes rather than using a programming language.
- **XML Schemas:** XML Schema includes more powerful constructs for defining the structure and content of an XML document than a DTD. Our XEM DTD change primitives would be adapted to handle XML Schemas with some extensions.
- **Customization of Evolution Rules:** XEM has defined default rules for update propagation to ensure consistency. However XEM also provides the flexibility for users to define their own escape rules for update propagation. For example, when changing the quantifier of a sub-element from REPEATABLE to ONCE, users may prefer to keep the last occurrence of the sub-element rather than the first occurrence. Further study into this direction towards a fully customizable XEM document management system is desirable.

References

[1] J. Andany, M. Leonard, and C. Palisser. Management of schema evolution in databases. In *VLDB*, pages 161–170, September 1991.

[2] J. Banerjee, W. Kim, H. J. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *SIGMOD*, pages 311–322, 1987.

[3] J. Bosak. Shakespeare's Plays in XML Format, v2.00. http://metalab.unc.edu/bosak/xml/eg/shaks200.zip.

[4] P. Bréche. Advanced Primitives for Changing Schemas of Object Databases. In *CAISE*, pages 476–495, 1996.

[5] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams. The GemStone Data Management System. In *Object-Oriented Concepts, Databases and Applications*, pages 283–308. ACM Press, 1989.

[6] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. In *In Proceedings of WWW10*, pages 201–210, 2001.

[7] S. Chawathe. Describing and Manipulating XML Data. In *IEEE Data Engineering Bulletin 22(3)*, pages 3–9, 1999.

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

[9] L. Chen, K. T. Claypool, and E. A. Rundensteiner. SERFing the Web: The Re-Web Approach for Web Re-Structuring. *WWW Journal - Special Issue on Internet Data Management, Baltzer/ACM Publication*, 2(1):33, 2000.

[10] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From Structured Documents to Novel Query Facilities. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Minneapolis*, pages 313–324, June 1994.

[11] K.T. Claypool, J. Jin, and E.A. Rundensteiner. OQL_SERF: An ODMG Implementation of the Template-Based Schema Evolution Framework. In *Centre for Advanced Studies Conference*, pages 108–122, November 1998.

[12] K.T. Claypool, J. Jin, and E.A. Rundensteiner. SERF: Schema Evolution through an Extensible, Re-usable and Flexible Framework. In *Int. Conf. on Information and Knowledge Management*, pages 314–321, November 1998.

[13] A. Deutsch, M.F. Fernandez, and D. Suciu. Storing Semistructured Data with STORED. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 431–442, Philadephia, USA, June 1999.

[14] W. Fan, G. Kuper, and J. Simon. A Unified Constraint Model for XML. In *In Proceedings of WWW10*, pages 179–190, 2001.

[15] W. Fan and J. Simon. Integrity constraints for XML. In *In Proceedings of the Nineteenth ACM Symposium on Principles of Database Systems*, pages 23–34. ACM Press, 2000.

[16] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. System Demonstration - Strudel: A Web-site Management System. In *ACM SIGMOD Conference on Management of Data*, pages 549–552, 1997.

[17] D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDBMS. In *IEEE Data Engineering Bulletin*, pages 27–34, 1999.

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

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

[20] Itasca Systems Inc. Itasca Systems Technical Report. Technical Report TM-92-001, OODBMS Feature Checklist. Rev 1.1, Itasca Systems, Inc., December 1993.

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

[22] A. Koeller. Semi-Structured Data in Relational Databases. Technical report, Worcester Polytechnic Institute, 1999.

[23] D. Kramer. XML Evolution Management, Master Thesis, Worcester Polytechnic Institute. Master's thesis, Worcester Polytechnic Institute, 2001.

[24] B.S. Lerner. A Model for Compound Type Changes Encountered in Schema Evolution. Technical Report UM-CS-96-044, University of Massachusetts, Amherst, Computer Science Department, 1996.

[25] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. In *SIGMOD Record 26(3)*, pages 54–66, September 1997.

[26] G. Mecca, P. Merialdo, and P. Atzeni. Araneus in the era of xml. In *Bulletin of the Technical Committee on Data Engineering*, pages 19–26, September 1999.

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

[28] A. Nica and E. A. Rundensteiner. Uniform Structured Document Handling using a Constraint-based Object Approach. In *ADL*, pages 83–101, 1995.

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

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

[31] Y. Papakonstantinou, H. Garcia Molina, and J. Widom. Object Exchange across Heterogeneous Information Sources. In *Proceedings of the 11th International Conference on Data Engineering, Taipei, Taiwan*, pages 251–260, March 1995.

[32] D. Sjoberg. Quantifying Schema Evolution. *Information and Software Technology*, 35(1):35–54, January 1993.

[33] A. H. Skarra and S. B. Zdonik. The Management of Changing Types in an Object-Oriented Databases. In *Proc. 1st OOPSLA*, pages 483–494, 1986.

[34] Igor Tatarinov, Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Updating XML. In *SIGMOD*, 2001.

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

[36] Versant Object Technology. *Versant User Manual*. Versant Object Technology, 1992.

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

[38] W3C. *Extensible Markup Language (XML) 1.0, 2nd Edition – W3C Recommendation 6-October-2000*. http://www.w3.org/TR/REC-xml, 2000.

[39] W3C. *XML Schema – W3C Proposed Recommendation 2001-03-16*. http://www.w3.org/XML/Schema, 2001.

[40] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, 2001.

[41] X. Zhang, G. Mitchell, W. Lee, and E. A. Rundensteiner. Clock: Synchronizing Internal Relational Storage with External XML Documents. In *Eleventh International Workshop on Research Issues in Data Engineering (RIDE), Heidelberg, Germany*, pages 111–118. IEEE Computer Society, April 2001.

[42] R. Zicari. A Framework for O_2 Schema Updates. In *7th IEEE Int. Conf. on Data Engineering*, pages 146–182, April 1991.