World Wide Web (2015) 18:1541–1577
DOI 10.1007/s11280-014-0321-1

Maintaining schema versions compatibility in cloud applications collaborative framework

Abdullah Baqasah · Eric Pardede · Wenny Rahayu

Received: 1 September 2014 / Revised: 1 December 2014 /
Accepted: 22 December 2014 / Published online: 19 March 2015

Abstract The eXtensible Markup Language (XML) is a meta language that is widely used to provide a non-proprietary universal format for sharing hierarchical data among different software systems and application domains. Many organizations and content providers have been publishing and sharing their information through XML and its standard schemas. With the increased popularity of cloud application deployment, it is a common practice to share data and its schemas, which underpins integrated applications within the cloud environment. Cloud environment fosters collaboration more than in the traditional distributed system, through i) a direct access and update of shared files using a web-based collaboration packages and ii) a seamless access by new technologies such as smartphones and tablet devices. Since the heterogeneous schemas stored in the cloud tend to evolve across time, there is a need to handle their versions adequately. In this paper, we propose a central framework the can be deployed in a cloud environment to aid schema developers and standard groups to track XML Schema changes, maintain versions compatibility, and help in the enhancement of a particular schema version. The framework is prototyped as a tool (called XSM) to store and retrieve versioned XSDs and evaluate them based on the quality indicators defined for this purpose. The versioning correctness and functionality of the proposed indicators are examined through a set of XSDs.

Keywords XML schema compatibility · Versioning algorithms · Cloud collaboration · Version control · Version retrieval · Schema quality

1 Introduction

Introduced by W3C, XML has become a very popular language for storing and disseminating semi-structured information. It has been widely used to provide a non-proprietary universal format for sharing hierarchical data among different software systems and

A. Baqasah · E. Pardede (✉) · W. Rahayu
Department of Computer Science and Computer Engineering, La Trobe University, Melbourne, Australia
e-mail: e.pardede@latrobe.edu.au

A. Baqasah
e-mail: ambaqasah@students.latrobe.edu.au

W. Rahayu
e-mail: w.rahayu@latrobe.edu.au

application domains. Specifications and standards used in these domains are described by XML Schema Language [41]. It is natural that schema standards have undergone different type of changes due to new requirements and business developments; indeed each standard ends up with different versions of the same schema. For instance, OpenTravel [26] for travel industry, OASIS's ebXML [14] for Supply Chain, and ACORD [1] for insurance and related industries, are examples of industry-specific standards written in XML Schema language. Each of the previous bodies releases nearly two public versions per year and some of them with internal revisions within one version.

As collaboration through cloud system continues to increase in the last few years [35], a successful strategy for XML co-authoring and collaboration must be adopted to allow multiple users to remotely access and edit either documents or schema versions stored in a cloud environment. In this environment, such versions need to be uploaded to the cloud where they can then be accessed, queried, updated by different collaborators. Several approaches has been proposed for XML authoring and collaborative editing either by the research community such as [15, 34, 37], or as commercial tools such as SDL LiveContent Create [32], SERNA XML Editor [33], oXygen XML Editor [28], and ALTOVA [2]. These editors facilitate the deployment of seamless collaborative workflows across XML publishing and documentation. As some of the previous tools (such as oXygen XML Editor and ALTOVA) explicitly supports XML Schema creation and identifies the schema components through a graphical interface, there is a greater chance of developing a new approach that can manage XML Schemas in the cloud as well.

Systems that manage large corpora of XML documents such as [9–11] are commonly deployed on cloud infrastructure. Related to our work, the cloud environment is not used only to offer a storage and access to the shared schema versions but should also maintain the compatibility between different versions of one schema and provide useful information (e.g., evaluating the stored versions, allowing the retrieval of schema changes) to the users of the schema.

In this paper, we focus on the methodology for schema version control, which can support collaboration in the cloud environment. Schema versioning as a solution in this context can be used for the following motivations:

- First, to manage the process of accommodating XML Schema Definition (XSD) versions and maintain the compatibility of the stored versions.
- Second, to allow schema developers and working groups to track schema changes and record them for future assessment.
- Third, to guide schema developers on making a decision about new schema version i.e., accepting or considering a further revision of a particular schema version. The revised schema is altered until it reaches a reasonable level of acceptance determined by the quality indicators.

Our major contributions in this paper are as follows:

- We propose a versioning model and an algorithm for XML Schema versions. The algorithm takes the schema version V_i and the delta changes $\Delta_{i \to i+1}$ as inputs and produces the next version V_{i+1} as output. The internal representation of schema documents is then constructed based on XML Schema Object Model (XSOM) [16].

- We define a framework for monitoring XML Schema versions, and introduce its main tasks *Version Insertion*, *Versioning Comparison*, *Version Retrieval, and Delta Evaluation*.
- We extend our tracking tool (called XML Schema Monitor (XSM)) proposed in [4]. XSM can be used for two purposes. First, to efficiently store and retrieve different schema repositories each within its versions. Second, for a specific delta change, that transforms one version into another, it applies quality indicators to measure the *reusability* and *extensibility* of the delta.

The remainder of this paper is organized as follows. In Section 2, we consider related works on XML version management and schema quality. We describe our versioning representation, delta model, and delta storage techniques in Section 3. Section 4 introduces the framework of our monitoring system XSM and explains its main tasks. We evaluate the correctness and functionality of XSM tool in Section 5. Finally, concluding remarks and thoughts on future research work are discussed in Section 6.

2 Related work

2.1 Schema versioning in XML DBMS

XML schema versioning and its related issues on modification and evolution have been supported by many commercial DBMSs. For instance, in *SQL Server* [25], an XML Schema collection C is created to validate XML instances and to type XML data as it is stored in the database. Each time a new version of an XSD is added to the collection C, it is given a new target namespace. This allows a multiple storage of different schema versions. Therefore, XML column of type C can store instances conformed to different schema versions. Unlike SQL Server, an XML column in *DB2 9* can store any well-formed XML documents [31]. When validation is required, DB2 9 provides an XML Schema Repository (XSR) where the schemas of XML documents can be registered for validation purposes. DB2 9 also supports a simple form of schema evolution i.e., the new schema version replaces the old one if a backward compatibility is guaranteed. Schema versioning is also supported through the ability to retain the old schema or deleting it once the old schema version has been updated. All previous tools identify and store XML schema versions without storing the information about differences between them.

In *Oracle XML DB* [27], XMLType data is implemented as schema-typed or untyped data. For schema-typed data, XML Schema should be created and registered. Once registered, XML Schema can be used to validate the related documents. Although Oracle XML DB supports two kinds of schema evolution: copy-based and in-place without requiring backward-compatibility, it does not directly support schema versioning.

2.2 Schema versioning research approaches

In XML data processing, versioning techniques are proposed to manage multiple versions of the dynamic documents but research of XML schema versioning is scarce. For example [12], and [30] focus on document management. Authors of [12] proposed a temporal clustering

technique based on the notion of *page-usefulness* to optimize the versioning of XML documents. They introduce two schemes for storing and retrieving XML data: *edit-based* and *reference-based* scheme. The main concern of [30] was the efficiency for the delta storage. They propose a technique called *consolidated delta* ($C\Delta$) that combines an initial XML document version with a history of changes in one file. Based on the information available in the consolidated delta, the system reconstructs the required version.

Schema versioning has been previously studied in the context of multi-temporal databases [7]. Presents a schema versioning approach based on the XML Schema Definition language (XSD). A set of basic change operations, which deal with element and attribute components, is introduced as a guideline. Authors also discuss different issues related to the propagating of changes and maintaining temporal multi-schema queries in the versioning environment. In another work [6], proposes a set of schema change primitives in the context of *tXSchema* framework [13]. In tXSchema there are three levels to specify a schema for a time-varying data. The first level is for *conventional schema*, which is a standard XML Schema describes an XML document. The second and third levels are for *logical annotations* and *physical annotations* of the conventional schema. Those annotations are stored together in a single XML document called *annotation document* and used to identify which elements can vary over time and where timestamps should be placed for those elements. The proposed primitives in tXSchema context are applied to the annotation document. Authors stated that they do not deal with changes of the conventional schema, which is the main consideration in this paper.

2.3 XML schema quality analysis

Measuring XML schema qualities such as *complexity*, *extensibility*, *reusability*, and *understandability* is an important issue to maintain the easiness of XML schemas. Research developed in this area has covered designing metrics for Document Type Definitions (DTDs) [8] and XML Schema Documents (XSDs) [41]. For example [17], focuses on the complexity of the related XML documents by proposing five metrics for DTD documents. They are *lines of code*, *McCabe complexity*, *structure depth*, *fan-in*, and *fan-out* metrics. Since the DTD syntax is not in XML formal, its quality metrics can't be exploited to measure the quality of XML Schema versions.

For XML Schema, several approaches have been proposed [24]. Provides a common approach to measure schema complexity by counting number of components. The work extends the metrics proposed by [17] in two directions: i) it introduces new metrics to measure XML Schema annotation components and ii) it introduces metrics that exploit XML Schema user-defined types. An effort made in [5] is dedicated to measure the complexity due the recursion usage in the schema. Authors of this work argue that metrics proposed for XML Schema's complexity by counting number of components do not give sufficient information about the complexity value of a given schema. More recently [38], proposes three schema metrics: *Reusable Quality metric* (RQ), *Extensible Quality metric* (EQ) and *Understandable Quality metric* (UQ). The proposed metrics are based on 'Binary Entropy Function and Rank Order Centroid' method and help in facilitating the assessment of schema-based software products. The previous schema quality metrics intentionally designed to measure XML Schema qualities, which basically target the XSD documents. Our indicators proposed in Section 4.4 target XSD deltas. It investigates the delta used to generate a particular version and provides enough information about the extensibility and reusability of the expected version.

3 Versioning model

In this section, we first describe XSM versioning model, that is how versions are captured in a tree-like structure, and stored along with their deltas in the relational model (*XS-Rel*) described in [3]. Then, we define the delta model (*XS-Rel-Delta*) that is used to store differences between schema versions. After that, we discuss different storage policies for schema versions. That is, how successive versions are organized in the system. Finally, *SFXS* storage policy which meets XML Schema versioning requirements is defined with an illustrative example.

3.1 Versioning tree model

To store XML Schema versions we use the method of decomposing them into relational tables. The stored versions are then compared and changes are stored in the delta tables. The process of differencing input versions is covered in our *XS-Diff* algorithm in [3]. In this work, we step further by maintaining the history of changes (set of deltas), so we are able to track changes for each schema component through different versions. The central representations in this versioning are *repositories*, *versions*, and *deltas*.

- ***Repositories*** are used to store information about versioned schemas. For example, shiporder is a repository name of the schema in our running example in Figures 1, 2, and 3. Repositories are stored in XS-Rel table repository, which has the following schema: repository (rid, rname, active, initVersion, currVersion, noVersions). As clearly seen, this table is designed to store basic information such as the initial version initVersion (e.g., V_b), current version currVersion (e.g., V_3),

```
1 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
2  <xs:attribute name="country" type="xs:NMTOKEN"
fixed="US"/>
3  <xs:element name="pOrder" type="POType"/>
4  <xs:complexType name="Items">
5   <xs:sequence>
6    <xs:element name="item" minOccurs="0"
maxOccurs="unbounded">
7     <xs:complexType>
8      <xs:sequence>
9       <xs:element name="productName" type="xs:string"/>
10      <xs:element name="quantity">
11       <xs:simpleType>
12        <xs:restriction base="xs:positiveInteger">
13         <xs:maxExclusive value="100"/>
14        </xs:restriction>
15       </xs:simpleType>
16      </xs:element>
17      <xs:element name="USPrice"  type="xs:decimal"/>
18     </xs:sequence>
19     <xs:attribute name="partNum" type="SKU"
use="required"/>
20    </xs:complexType>
21   </xs:element>
22  </xs:sequence>
23 </xs:complexType>
24 <xs:complexType name="POType">
25  <xs:sequence>
26   <xs:element name="shipTo" type="xs:string"/>
27   <xs:element name="billTo" type="xs:string"/>
28   <xs:element name="items"  type="Items"/>
29  </xs:sequence>
30  <xs:attribute name="orderDate" type="xs:date"/>
31  <xs:attribute name="orderID" type="xs:string"/>
32 </xs:complexType>
33 <xs:simpleType name="SKU">
34  <xs:restriction base="xs:string">
35    <xs:pattern value="\d{3}-[A-Z]{2}"/>
36  </xs:restriction>
37 </xs:simpleType>
38</xs:schema>
```

Vb

```
1 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
2  <xs:attribute name="country" type="xs:NMTOKEN"
fixed="US"/>
3  <xs:element name="pOrder" type="POType"/>
4  <xs:element name="comment" type="xs:string"/>
5  <xs:complexType name="Items">
6   <xs:sequence>
7    <xs:element name="item" minOccurs="0"
maxOccurs="unbounded">
8     <xs:complexType>
9      <xs:sequence>
10      <xs:element name="productName" type="xs:string"/>
11      <xs:element name="quantity">
12       <xs:simpleType>
13        <xs:restriction base="xs:positiveInteger">
14         <xs:maxExclusive value="50"/>
15        </xs:restriction>
16       </xs:simpleType>
17      </xs:element>
18      <xs:element name="USPrice"  type="xs:decimal"/>
19      <xs:element ref="comment"   minOccurs="0"/>
20     </xs:sequence>
21     <xs:attribute name="partNum" type="SKU"
use="required"/>
22    </xs:complexType>
23   </xs:element>
24  </xs:sequence>
25 </xs:complexType>
26 <xs:complexType name="POType">
27  <xs:sequence>
28   <xs:element name="shipTo" type="xs:string"/>
29   <xs:element name="billTo" type="xs:string"/>
30   <xs:element name="items"  type="Items"/>
31  </xs:sequence>
32  <xs:attribute name="orderDate" type="xs:date"/>
33 </xs:complexType>
34 <xs:simpleType name="SKU">
35  <xs:restriction base="xs:string">
36    <xs:pattern value="\d{3}-[A-Z]{2}"/>
37  </xs:restriction>
38 </xs:simpleType>
39</xs:schema>
```

V1

Figure 1 Changes between versions V_b and V_1 of shiporder schema

Figure 2 Changes between versions V_1 and V_2 of shiporder schema

and the number of versions noVersions, which counts versions for a specific repository (e.g., 4 versions). Here, active attribute in the table is used at run time to assign a flag to the repository that requires versioning.

- ***Versions*** are points in time that represent repository versions. They are stored in XS-Rel table version in a way that support the linear and the branched features of the versioned schema. The table has the following structure: version (vid, vnum, vparentnum, vdocument, date, rid), where vid is the unique id assigned by the system to each newly inserted version, vnum is the version number of a specific schema (e.g., 0, 1, 2, and 3 are the numbers of V_b, V_1, V_2, and V_3 versions in Figure 1, respectively), and vparentnum is the version number of the immediate parent of the version. Although vparentnum is insignificant in our running example, it is important in the branched versioning to identify the version in the tree. vdocument stores the actual name of the version file (e.g., shiporderVb.xsd is the name of the initial version in Figure 1). date is used to store the date and time when the version is inserted in the system. This field allows us to perform temporal queries such as retrieving a version at a specific time or querying schema changes in a time slice.
- ***Deltas*** are used to store components changes in the form of XS-Rel-Delta. Different XS-Rel delta tables are created according to XML Schema components discussed in [3]. For example, delta representing changes of schema components in Figure 1 are as follows:

XS-Rel-Delta$_{Vb \rightarrow V1}$*: {Delete (21, AD), Insert (24, ED), Insert (26, ED), Delete (12, F), Insert (25, F)}*, where *21*, *24*, *26*, *12*, and *25* are node ids, which replace the actual node paths of the changed components as follows:

```
1 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
2  <xs:attribute name="country" type="xs:NMTOKEN"
fixed="US"/>
3  <xs:element name="pOrder" type="POType"/>
4  <xs:element name="street" type="xs:string"/>
5  <xs:element name="comment" type="xs:string"/>
6  <xs:complexType name="Items">
7   <xs:sequence>
8    <xs:element name="item" minOccurs="0"
maxOccurs="100">
9     <xs:complexType>
10     <xs:sequence>
11      <xs:element name="productName" type="xs:string"/>
12      <xs:element name="quantity">
13       <xs:simpleType>
14        <xs:restriction base="xs:positiveInteger">
15         <xs:maxExclusive value="50"/>
16        </xs:restriction>
17       </xs:simpleType>
18      </xs:element>
19      <xs:element name="USPrice"  type="xs:decimal"/>
20      <xs:element ref="comment"   minOccurs="0"/>
21     </xs:sequence>
22     <xs:attribute name="partNum" type="partNumType"
use="required"/>
23    </xs:complexType>
24   </xs:element>
25  </xs:sequence>
26 </xs:complexType>
27 <xs:complexType name="POType">
28  <xs:sequence>
29   <xs:element name="shipTo" type="USAddress"/>
30   <xs:element name="billTo" type="USAddress"/>
31   <xs:element name="items"  type="Items"/>
32  </xs:sequence>
33  <xs:attribute name="orderDate" type="xs:date"/>
34 </xs:complexType>
35 <xs:complexType name="USAddress">
36  <xs:sequence>
37   <xs:element name="name"   type="xs:string"/>
38   <xs:element ref="street"/>
39   <xs:element name="city"  type="xs:string"/>
40   <xs:element name="state"   type="xs:string"/>
41   <xs:element name="zip"    type="xs:decimal"/>
42  </xs:sequence>
43  <xs:attribute ref="country"/>
44  <xs:attribute name="countryCode" type="xs:string"/>
45 </xs:complexType>
46 <xs:simpleType name="partNumType">
47  <xs:restriction base="xs:string">
48   <xs:pattern value="\d{3}-[A-Z]{2}"/>
49  </xs:restriction>
50 </xs:simpleType>
51</xs:schema>
```

V2

```
1 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
2  <xs:attribute name="orderDate" type="xs:date"/>
3  <xs:element name="pOrder" type="POType"/>
4  <xs:element name="productName" type="xs:string"/>
5  <xs:element name="comment" type="xs:string"/>
6  <xs:complexType name="Items">
7   <xs:sequence>
8    <xs:element name="item" minOccurs="0"
maxOccurs="100">
9     <xs:complexType>
10     <xs:sequence>
11      <xs:element ref="productName"/>
12      <xs:element name="quantity">
13       <xs:simpleType>
14        <xs:restriction base="xs:positiveInteger">
15         <xs:minInclusive value="1"/>
16        </xs:restriction>
17       </xs:simpleType>
18      </xs:element>
19      <xs:element name="USPrice"  type="xs:decimal"/>
20      <xs:element ref="comment"   minOccurs="0"/>
21     </xs:sequence>
22     <xs:attribute name="partNum" type="partNumType"
use="required"/>
23    </xs:complexType>
24   </xs:element>
25  </xs:sequence>
26 </xs:complexType>
27 <xs:complexType name="POType">
28  <xs:sequence>
29   <xsd:element name="deliveryInfo">
30    <xsd:complexType>
31     <xsd:sequence>
32      <xsd:element name="shipTo" type="USAddress"/>
33      <xsd:element name="billTo" type="USAddress"/>
34     </xsd:sequence>
35     <xsd:attribute name="orderID" type="xsd:string"/>
36     <xsd:attribute name="trackID" type="xsd:string"/>
37    </xsd:complexType>
38   </xsd:element>
39   <xs:element name="items"  type="Items"/>
40  </xs:sequence>
41  <xs:attribute ref="orderDate"/>
42 </xs:complexType>
43 <xs:complexType name="USAddress">
44  <xs:sequence>
45   <xs:element name="fullname"   type="xs:string"/>
46   <xs:element name="street type="xs:string"/>
47   <xs:element name="state"  type="xs:string"/>
48   <xs:element name="city"   type="xs:string"/>
49   <xs:element name="zip"    type="xs:integer"/>
50  </xs:sequence>
51  <xs:attribute name="country" type="xs:NMTOKEN"
fixed="US"/>
52 </xs:complexType>
53 <xs:simpleType name="partNumType">
54  <xs:restriction base="xs:string">
55   <xs:pattern value="\d{3}-[A-Z]{2}"/>
56  </xs:restriction>
57 </xs:simpleType>
58</xs:schema>
```

V3

Figure 3 Changes between versions V_2 and V_3 of shiporder schema

3.2 Delta model

The concept *delta* is used by most prior research to describe changes between two consecutive versions of XML document [23, 29, 30, 37, 43], but the term has not been used in XML Schema versioning. Changes between versions of XML Schema is usually denoted by *primitive changes* [6] and they are attached to the schema document itself. In addition, the change format differs from one approach to another, based on the purpose of the application that generates the changes. For example, algorithms maintain hierarchically-structured data store changes as edit script. Other algorithms for XML document differencing can store changes variously as edit script, XML document, relational records, or multiple formats.

In our context, delta format is based on the change detection model proposed in our previous work using delta operations in the format of relational tables [3]. This technique is already examined in the context of XML document change detection using different approaches with XML trees that are ordered [18, 21] and unordered [19, 20, 36]. In what follows, we discuss each type of delta and show the advantages of adopting the relational-based delta in the context of XML Schema. Simplified trees (showing only affected nodes and their neighbours) of the versions in Figure 1 are depicted in Figure 4.

3.2.1 XML delta

A simple XML delta can be defined as follows:

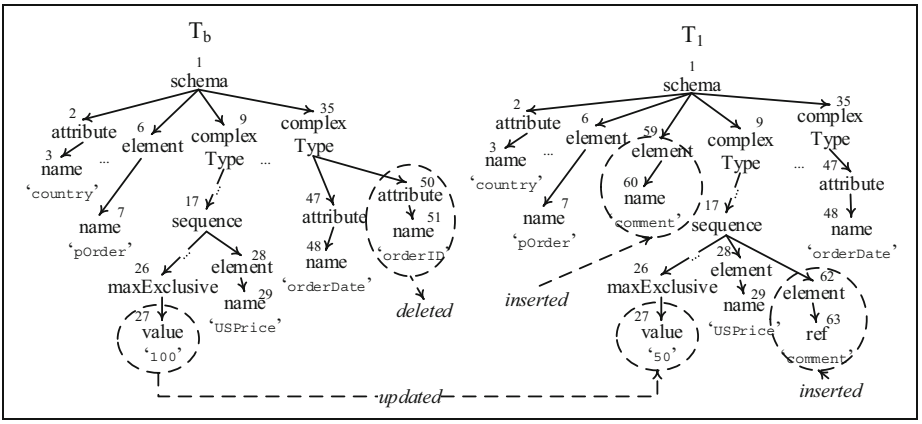


Figure 4 Two trees T_b and T_1 representing schema versions V_b and V_1 in Figure 1 (the tree model comply to [23] for defining XML Schema as XML document tree that has element, attribute, and text nodes)

Definition 1 -(XML-Delta) Let S be an XML Schema with two versions V and V', where $V \neq V'$. XML-Delta is an XML document with change operations that convert S from one state into another. XML-Delta consists of basic operations O= *{Insert, Delete, Update, Move}*, which if applied to V will produce V'.

The operations used by XML-Delta (e.g., those proposed by [23]) is also applicable to XML Schemas and defined as follows:

Definition 2 (XML-Delta operations) Given XML Schema with two trees T_1 and T_2 representing two successive versions V_1 and V_2, respectively, the following XML-Delta Operations can be defined on an XML node (i.e., element, attribute, or text nodes from XML point of view) from T_1 and/or T_2:

- Delete (UP_N): delete the node N which has unique path UP_N from T_1;
- Insert (UP_P, po, N): insert the node N at position po under the parent node with unique path UP_P into T_2;
- Update (UP_N, nv): update the node N which has unique path UP_N, from its old value in T_1 to the new value nv in T_2;
- Move (UP_P, po, UP_N): move the node N, which has unique path UP_N, from its old position in T_1 to the new position po in T_2 under the parent node with unique path UP_P.

Example Based on Definition 2, XML-Delta operations describing changes between V_b and V_1 in Figure 1 are: {*Delete (50)*, *Insert (1, 3, '<element name='comment'>')*, *Insert (17, 4, '<element ref='comment'>')*, and *Update (27, '50')*} as seen in the tree representation in Figure 4.

As observed in the previous example, the information available in the delta operations are limited (e.g., given only a new value '50' in the *Update* operation is not enough to invert the operation so that it transforms V' into V). This leads in creating different alternatives for XML-Delta to allow it to move forward, backward, or bi-directionally between schema versions.

3.2.2 XML forward, backward, and completed delta

The operations in the forward delta can only be used to transform an old version of XML Schema into a new one. We formally define XML forward delta as follows:

Definition 3 (XML-Forward-Delta) Let S be an XML Schema with two successive versions V_1 and V_2, where $V_1 \neq V_2$, XML-Forward-Delta is defined as XML-Delta with change operations that transform V_1 into V_2.

Example For the two trees T_b and T_1 (shown in Figure 4) representing versions V_b and V_1 of shiporder schema, the XML-Forward-Delta which transforms V_b into V_1 consists of the following operations: {*Delete (50)*, *Insert (1, 3, '<element name='comment'>')*, *Insert (17, 4, '<element ref='comment'>')*, *Update (27, '50')*}.

Opposite to the previous delta, operations in the backward delta can only be used to transform the new version of XML Schema into the old one. XML backward delta is defined as follows:

Definition 4 (XML-Backward-Delta) Let S be an XML Schema with two successive versions V_1 and V_2, where $V_1 \neq V_2$, XML-Backward-Delta is defined as XML-Delta with change operations that transform V_2 into V_1.

Example For the two trees T_b and T_1 (shown in Figure 4) representing versions V_b and V_1 of shiporder schema, the XML-Backward-Delta which transforms V_1 into V_b consists of the following operations: {*Delete (59)*, *Delete (62)*, *Insert (35, 3, '<attribute name='orderID'>')*, *Update (27, '100')*}.

For the previous two types of delta, XML-Forward-Delta and XML-backward-Delta, the major problem (as can be seen in the examples) is that using one type of delta is not enough to move back and forth through the versions of the schema. For example, the operation *Delete (50)* in XML-Forward-Delta in Figure 4 is only useful to move forward and cannot be reversed to move backward (e.g., from new version V_1 into old version V_b). In other words, there is missing information that disallows us to reverse the operation to move backward.

To resolve the issue of *one-direction* transformation, a new delta type called 'completed delta' has been introduced by [23]. We define XML completed delta in the context of XML Schema versioning as follows:

Definition 5 (XML-Completed-Delta) Let S be an XML Schema with two successive versions V_1 and V_2, where $V_1 \neq V_2$, XML-Completed-Delta is defined as XML-Delta with change operations that transform V_1 into V_2 and V_2 into V_1.

To meet the requirements of XML-Completed-Delta in Definition 5, we modify the operations in Definition 2 as follows:

Definition 2a (Enhanced Delta Operations). Given XML Schema with two trees T_1 and T_2 representing two successive versions V_1 and V_2, respectively, the following XML-Delta Operations can be defined on an XML node (i.e., element, attribute, or text nodes from XML point of view) from T_1 and/or T_2:

- Delete (UP_P, *po, N*): delete the node N existing at position *po* under the parent node with unique path UP_P from T_1;

- Insert (UP_P, *po, N*): insert the node *N* at position *po* under the parent node with unique path UP_P into T_2;
- Update (UP_N, *ov, nv*): update the node *N*, which has a unique path UP_N, from the old value *ov* in T_1 to the new value *nv* in T_2;
- Move (UP_{OP}, *opo*, UP_N, UP_{NP}, *npo*): move the node *N*, which has a unique path UP_N, from the old position *opo* in T_1 under the parent node with unique path UP_{OP} to the new position *npo* in T_2 under the parent node with unique path UP_{NP}.

Example For the two trees T_b and T_1 (shown in Figure 4) representing versions V_b and V_1 of shiporder schema, the XML-Completed-Delta contains the following operations: {*Delete (35, 3, '<attribute name='orderID'>')*, *Insert (1, 3, '<element name='comment'>')*, *Insert (17, 4, '<element ref='comment'>')*, *Update (27, '100', '50')*}.

3.2.3 XML Schema completed delta (XS-Rel-Delta)

We adapt XML-Completed-Delta as a change model but with some alterations to satisfy XML Schema specific changes (e.g., migration change of element/attribute declarations and order change of sequence model group children). The enhanced change model XS-Rel-Delta was introduced in [3] where the relational engine was used as a medium to find changes between successive schema versions and store delta changes. Operations that can be applied to XML Schema components is defined as follows.

Definition 6 (XML Schema operations). Given XML Schema with two trees T_1 and T_2 representing two successive versions V_1 and V_2, respectively, the following XML Schema operations can be defined on XML Schema nodes (i.e., *AD*, *ED*, *ST*, *CT*, *MG*, *F*, *AG*, and *GD*, where *AD* is an attribute declaration, *ED* is element declaration, *ST* is a simple type definition, *CT* is a complex type definition, *MG* is a model group, *F* is a restriction facet, *AG* is an attribute group definition, and *GD* is a group definition) from T_1 and/or T_2:

- Delete (UP_N, *N*): delete the node *N*, which has a unique path UP_N, from T_1;
- Insert (UP_N, *N*): insert the node *N*, which has a unique path UP_N, into T_2;
- Update (UP_N, *OV, NV*): update one or more of the properties (e.g., name, type, minOccurs, or value) of the node with unique path UP_N, from the old values *OV* in T_1 to the new values *NV* in T_2;
- Move (UP_{Old}, UP_{New}, *N*): move the node *N* from its old unique path UP_{Old} in T_1 to the new unique path UP_{New} in T_2.
- Migrate (UP_{Old}, UP_{New}, N_{Old}, N_{New}): move the node *N* from its old unique path UP_{Old} in T_1 to the new unique path UP_{New} in T_2 and update the node properties from N_{Old} to N_{New}.

Based on the Definition 6, XS-Rel-Delta is defined as follows:

Definition 7 (XS-Rel-Delta). Given any two successive versions V_1 and V_2 of an XML Schema *S*, where $V_1 \neq V_2$, XS-Rel-Delta is a set of relational tables that record the changes of the schema from one version to another. XS-Rel-Delta consists of a set of operations *O*=

{Delete, Insert, Update, Move, Migrate} (listed in Definition 6) that transform V_1 into V_2 and V_2 into V_1.

To give an example of XS-Rel-Delta, the changes between V_b and V_1 in Figure 1 are transformed to their respective trees using our XSD tree model (proposed in [3]) in Figure 5.

Example For the two trees T_b and T_1 (shown in Figure 5) representing V_b and V_1 schema versions (shown in Figure 1), operations that are included in the XS-Rel-Delta and cover different schema components are: {*Delete (12, 'maxExclusive:100')*, *Delete (21, 'attribute:orderID')*, *Insert (24, 'element:comment')*, *Insert (25, 'maxExclusive:50')*, *Insert (26, 'elementRef:comment')*}.

It is important to note that types of changes in XS-Rel-Delta are, to some extent, different from those contained in XML-Completed-Delta. In XS-Rel-Delta, we put emphasis on changes that are more practical to XML Schemas. We show the benefit of introducing the new set of XML Schema changes and omitting the existing XML traditional changes (i.e., operations in Definition 2 and its enhanced version Definition 2a) by the following example.

Assume that we need a transformation between two versions V_2 and V_3 in Figure 3. The traditional XML operations counted by XML-Completed-Delta are shown in Table 1 (a). On the other side, the changes recorded by XS-Rel-Delta model are listed in Table 1 (b). The traditional XML-Completed-Delta poses a number of issues, especially relating to the irrelevance of delta to XML Schema changes. Some of the most important are detailed as follows:

- *Counting order changes in XML ordered model* – if the XML ordered model (i.e., when both parent–child relationship and right-to-left order among siblings in XML tree are significant) is applied to compute XML-Completed-Delta, it will calculate all order

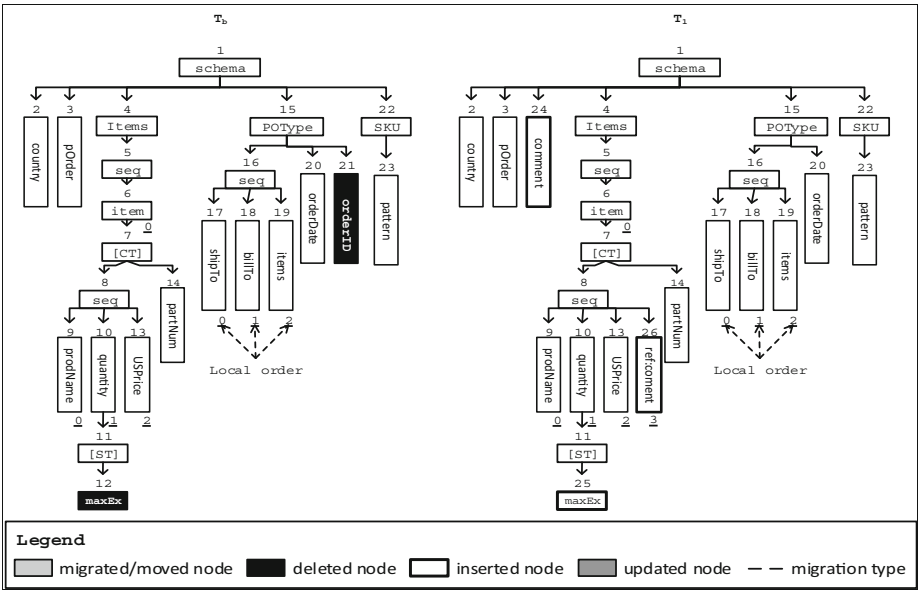


Figure 5 Two trees T_b and T_1 representing schema versions V_b and V_1 in Figure 1 (using XSD tree model defined in [3])

changes in the XML tree. In other words, the order between siblings of the same parent node in XML Schema is not always important. The order should be maintained only for children of a sequence model group. This issue does not appear in the operations listed in Table 1 (a), however, we can think of a switch (position change) between the two complex type definitions Items and POType (lines 6 and 27 in Figure 3, respectively) as an order change (from XML ordered document perspective). The XML-Completed-Delta would contain the additional two operations *Move (1, 5, 15, 1, 6)*, which moves Items complex type from the fifth position under schema node to the sixth position, and *Move (1, 6, 44, 1, 5)*, which moves POType complex type from the sixth position to the fifth.

- *Missing a detailed information about XML Schema changes* – as operations in XML-Completed-Delta can be applied to both single nodes and sub-trees, one operation, such as *Insert (46, 1, S_{10})*, can be used to insert a sub-tree with a large set of elements and attributes. Unfortunately, such an operation is not helpful in understanding the types of changes occurring inside the sub-tree. This issue is solved in XS-Rel-Delta by decomposing a sub-tree operation into different schema component operations (e.g., *Insert (91, S_7)* and its subsequent operations *Insert (93, S_8)*, *Insert (94, S_9)*, *Insert (95, S_{10})*, and *Insert (98, S_{11})* shown in Table 1 (b)).
- *Unaware of an integrity between related operations* – some operations in XML Schema are related to each other (e.g., *remove* an element or attribute from the global definition and *update* its local reference to include the full definition). These operations in XML-Completed-Delta are maintained apart from each other, meaning that if one operation is missing, the other one is still valid and can be used in the delta. The more robust and accurate way to maintain XML Schema changes is to consider such operations as associated to each other, as we see in a *Migrate* operation in Table 1 (b).

3.3 Storage techniques for XML Schema versions

In this section, we discuss various storage policies (or the physical organization, to be more precise) of historical XML data. The aim is to discuss possible solutions for storing XML Schema deltas using techniques to store XML documents. Each of the existing storage policies is briefly discussed in order to decide the one that best fits the requirements of XML Schema versioning process. Four techniques (extensively studied by [30] in the context of multi-versioned XML documents) are used as a guideline and redefined in the context of XML Schemas. These techniques are (1) *SFD (Store the First XML document and all forward Deltas)*, (2) *SLD (Storing Last version of the document and the Deltas)*, (3) *SFLD (Storing First version, Last version, and the Deltas)*, and (4) *CΔ (the Consolidated Delta)*. Table 2 provides a definition, a brief description, and the version retrieval mechanism for each pre-existing storage policy.

As observed in policies (SFD, SLD, SFLD), it is clearly seen that there are some drawbacks in those techniques related to redundancy of data (e.g., the duplication of old and new values in different deltas) or operations (e.g., repeating the same operations every time in creating intermediate versions until the required version is retrieved). These issues is solved by proposing the fourth technique, that is a consolidated delta CΔ, however CΔ may not be the best storage technique to be applied for versioning XML Schemas. Applying CΔ to XML Schema versions poses the following issues:

Table 1 Operations computed in XML-Completed-Delta (based on Definition 2a) and XS-Rel-Delta (based on Definition 6) models for the XML Schema versions V_2 and V_3 in Figure 3 (abbreviations ED, AD, CT, ST, MG, F, AG, GD in (b) denote XML Schema components)

(a) *XML-Completed-Delta*		(b) *XS-Rel-Delta*	
Operation	Affected XML node (S_x)	Operation	Affected XML Schema component node (S_x)
Delete (30, 1, S_1)	element labelled 'maxExclusive'	*Delete (32, S_1)*	F named 'maxExclusive'
Delete (46, 1, S_2)	element named 'shipTo'	*Delete (47, S_2)*	ED named 'shipTo'
Delete (46, 2, S_3)	element named 'billTo'	*Delete (50, S_3)*	ED named 'billTo'
Delete (61, 1, S_4)	element named 'name'	*Delete (62, S_4)*	ED named 'name'
Delete (61, 2, S_5)	element with ref = 'street'	*Delete (78, S_5)*	AD named 'countryCode'
Delete (59, 2, S_6)	element labelled 'attribute' and named 'country'	*Insert (89, S_6)*	F named 'minInclusive'
Delete (59, 3, S_7)	element labelled 'attribute' and named 'countryCode'	*Insert (91, S_7)*	ED named 'deliveryInfo'
Insert (23, 1, S_8)	element with ref = 'prodName'	*Insert (93, S_8)*	CT (anonymous) under 'deliveryInfo' ED
Insert (30, 1, S_9)	element labelled 'minInclusive'	*Insert (94, S_9)*	MG (sequence) under 'deliveryInfo' ED
Insert (46, 1, S_{10})	subtree rooted at element with name = 'deliveryInfo'	*Insert (95, S_{10})*	ED named 'shipTo'
Insert (44, 2, S_{11})	element named 'fullname'	*Insert (98, S_{11})*	ED named 'billTo'
Insert (61, 1, S_{12})	element labelled 'attribute' and with ref= 'orderDate'	*Insert (101, S_{12})*	AD named 'orderID'
Update(75,'decimal','integer')	attribute with name = 'type' and value = 'decimal'	*Insert (104, S_{13})*	AD named 'trackID'
Move (1, 1, 2, 59, 2)	element labelled 'attribute' and named 'country'	*Insert (62, S_{14})*	ED named 'fullname'
Move (1, 3, 9, 61, 2)	element named 'street'	*Update (67, {order=3}, {order=4})*	ED named 'city'
		Update (70, {order=4}, {order=3})	ED named 'state'
		Update (73, {type=decimal}, {type=integer})	ED named 'zip'
		Migrate (2, 2, {type=Ø, fixed=Ø, ref=1}, {type=NMTOKEN,fixed=US, ref=0})	AD named 'country'
Move (23, 1, 24, 1, 3)	element named 'prodName'	*Migrate (9, 9, {type=Ø, ref=1}, {type=string,ref=0})*	ED named 'street'
Move (44, 2, 56, 1, 1)	element labelled 'attribute' and named 'orderDate'	*Migrate (56, 56, {type=date, ref=0}, {type=Ø, ref=1})*	AD named 'orderDate'
Move (61, 3, 67, 61, 4)	element named 'city'	*Migrate (24, 24, {type=string, ref=0}, {type=Ø, ref=1})*	ED named 'prodName'
Move (61, 4, 70, 61, 3)	element named 'state'		

Table 2 Current storage policies for XML documant versioning according to [30]

Technique	Description	Definition
SFD	Store the initial version of the document plus all forward deltas	$SFD=V_b\cup\{\Delta_i\|1\le i\le now\}$ where V_b is the initial version and Δ is a set of forward deltas associated to V_b
SLD	Stores the last version of the document plus all backward deltas	$SLD=V_{now}\cup\{\Delta'_i\|1\le i\le now\}$ where V_{now} is the last version and Δ' is a set of backward deltas associated to V_{now}
SFLD	Store both the first and last version of the document along with all completed deltas	$SFLD=V_b\cup V_{now}\cup\{\Delta^c_i\|1\le i\le now\}$ where V_b and V_{now} are the first and last versions, respectively, and Δ^c is a set of completed deltas
C Δ	Store the initial version of the document along with its deltas as stamped versions	$C\Delta=V_{cons}$ where V_{cons} is an initial version V_b that is incrementally updated to include information about changes in each version $V_i\|1\le i\le now$

- *Undesirable delta presentation by stamping a large (and unnecessary) amount of nodes* – the rules of building consolidated delta from a series of XML versions state that 'if any of the children is either *modified*, *deleted* or *inserted*, the parent node is stamped as *modified*'. The children in this rule refer to any XML node other than the root node schema. Therefore, if we stamp all parents in the XML Schema starting from the changed node, we will end up with a large set of nodes that are stamped 'modified' in the schema file, which may not be the desirable method for displaying XML Schema deltas.
- *Absence of supporting attribute changes* – CΔ method assigns an *ID* to each new element inserted with a new version. However, it is not clear how it handles attribute changes (e.g., the change of maxExclusive facet value from '100' to '50' in shiporder schema in Figure 1). Unlike XML document, attributes in XML Schema language play a key role in defining schema components (i.e., name, and type are always used to assign a name and a data type to XML elements and attributes). If we assume that attribute changes in CΔ method are computed as element modification, then we insert the whole element node with the new attribute value after the original element, which causes unnecessary redundant data.

3.4 Storage technique - SFXS (Storing First version and the XS-Rel-Deltas)

To address issues presented by the previous storage techniques we adapt the SFD technique and modify it so that it meets XML Schema versioning requirements. In our technique (*SFXS*), we use XS-Rel-Delta model (proposed in Section 3.2.3) to store changes between versions of XML Schema. We first give an overview of how SFXS is built, followed by an explanation of how it is used to query a specific version from a series of XML Schema versions.

As stated above, we adapt SFD technique to satisfy the build of our storage technique. In SFD, the set of forward deltas Δ consists of basic operations defined on general XML documents. For XML Schemas, we define different sets of operations and hence, the delta model XS-Rel-Delta used in SFXS technique is altered to meet schema-specific changes. SFXS is defined as $SFXS=V_b\cup\{\Delta_i^{xsc}\|1\le i\le now\}$ where V_b is the first version and Δ^{xsc} is a set of completed deltas in the form of XS-Rel-Delta. An XML Schema version V_j can be retrieved by applying a set of deltas Δ^{xsc} starting from the first version V_b (e.g., $V_j=V_b+\Delta^{xsc}_{b+1}+\ldots+\Delta^{xsc}_{j-1}+\Delta^{xsc}_j$). This method is

appropriate when the number of versions is relatively small and the cost of reconstructing a version is low. Our investigation of 40 real-world XML Schemas with a total number of 245 XSD versions[1] shows that the number of versions for each schema varies from 2 to 27 and the average number of versions per schema is 6. The investigation also reveals that the overall average size of a schema version is 181.27 kb.

The benefits of using SFXS technique can be shown by applying it to the example of XML Schemas in Figures 1, 2, and 3. In this example, a shiporder schema is developed in four consecutive versions V_b, V_1, V_2, and V_3 (the subscript $_b$ denotes the base or initial version). The changes between each version and the following one is highlighted and the respective tree representations using our XSD tree model are shown in Figures 5, 6, and 7. In SFXS technique, we only store changes to XML Schema components based on the matching between components with the same type. For example, a change of maxExclusive value from '100' in V_b to '50' in V_1 in Figure 1 is considered as facet *delete* and *insert* changes. Note that we do not compare the label of the facet node maxExclusive (as it would be done in XML Documents matching) because in this case we would match maxExclusive in V_b with all facets with similar label in the second version V_1, which is probably inefficient for versioning XML Schemas. Table 3 shows a history of changes to shiporder schema components through its different versions.

As seen in Table 3, we only store the changed components between any two consecutive versions. For example, an element street with node id 25 is recorded in delta V_{1-2} as *element insertion*. The same node is also migrated from the global definition at node 4 to the local definition at node 27 in delta V_{2-3}. Similarly, the facet named maxExclusive with value '50' is inserted at node 25 as seen in delta $V_{b\text{-}1}$ and deleted after that as seen in delta V_{2-3}.

As illustrated by the previous examples, SFXS technique serves XML Schema versioning tasks as follows:

- *It provides a human-readable way to track XML Schema changes that is also clearly legible* - by recording deltas as relational records in one place, we can query a history of a specific schema component (e.g., element or attribute) by issuing an SQL query. This type of information is important as it helps a schema designer avoid repeating the operation in the same component (e.g., inserting attribute country multiple times during the set of versioned schema).
- *It does not require extra processing to derive information from XML delta files* - as SFXS stores deltas as relational records, we can apply a SQL query to retrieve those delta changes. This means that we do not need to parse the delta file (e.g., consolidated delta file in case of CΔ storage technique) to identify the changed nodes.
- *It reduces the number of operations required to transfer one version into another* - since traditional XML document change operations are based on elements, attributes, and text nodes (as we have seen in Section 3.2.1), the number of operations required can easily increase, especially with the deep nesting of XML Schema tags. In SFXS technique, we incrementally only store the important changes and omit the child nodes that are dependents of the main components. For example, although a restriction node (line 35 in V_1 in Figure 2) is deleted along with the deletion of its parent simple type SKU, it is

1 Available at: https://docs.google.com/document/d/1mkJmt28f100DwPqCa8QKtrKWWhDilC1oBUI1pH9xh-M/edit?usp=sharing

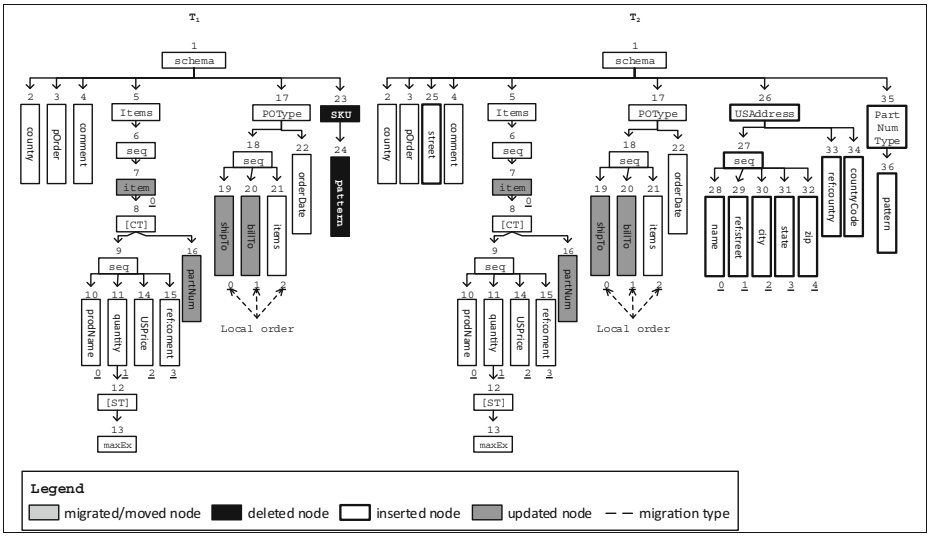


Figure 6 Two trees T_1 and T_2 representing schema versions V_1 and V_2 in Figure 2

not recorded as a deleted node in the resulting delta $V_{1–2}$ in Table 3. Instead, the deletion of restriction is embedded in the simple type operation *Delete (23, ST)*. The reduction feature becomes significant with a large XML Schema file containing a lot of simple and complex type definitions. For instance, complex types can have different descendent nodes whose changes are not counted (e.g., simpleContent, complexContant, restriction, and extension nodes).

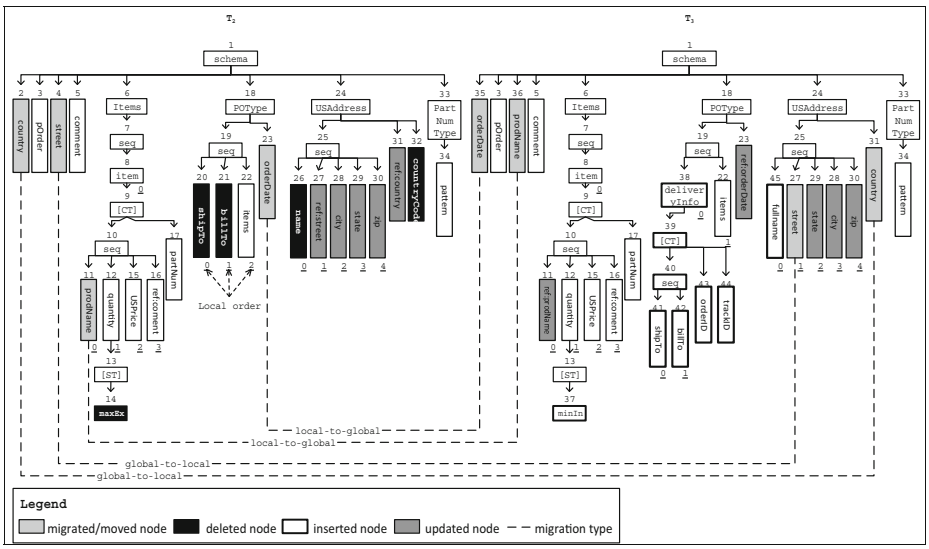


Figure 7 Two trees T_2 and T_3 representing schema versions V_2 and V_3 in Figure 3

Table 3 Set of deltas Δ^{xsc} storing a history of changes to shiporder schema through versions shown in Figures 5, 6, and 7

Component		V_{b-1}	V_{1-2}	V_{2-3}
Attribute (AD)	orderID	Delete (21, AD)		Insert (43, AD)
	ref:country		Insert (33, AD)	Update (31, {type=Ø, fixed=Ø, ref=1}, {type=NMTOKEN, fixed=US, ref=0})
	countryCode		Insert (34, AD)	Delete (32, AD)
	partNum		Update (16, {type=SKU}, {type=partNumType})	
	country			Migrate (2, 31, {type=Ø, fixed=Ø, ref=1}, {type=NMTOKEN, fixed=US, ref=0})
	orderDate			Migrate (23, 35, {type=date, ref=0}, {type=Ø, ref=1}), Update (23, {type=date, ref=0}, {type=Ø, ref=1})
	trackID			Insert (44, AD)
Element (ED)	comment	Insert (24, ED)		
	ref:comment	Insert (26, ED)		
	Street		Insert (25, ED)	Migrate (4, 27, {type=Ø, ref=1}, {type=string, ref=0})
	Name		Insert (28, ED)	Delete (26, ED)
	ref:street		Insert (29, ED)	Update (27, {type=Ø, ref=1}, {type=string, ref=0})
	City		Insert (30, ED)	Update (28, {order=2}, {order=3})
	state		Insert (31, ED)	Update (29, {order=3}, {order=2})
	zip		Insert (32, ED)	Update (30, {type=decimal}, {type=integer})
	item		Update (7, {maxOccurs=unbounded}, {maxOccurs=100})	
	shipTo		Update (19, {type=string}, {type=USAddress})	Delete (20, ED), Insert (41, ED)
	billTo		Update (20, {type=string}, {type=USAddress})	Delete (21, ED), Insert (42, ED)
	productName			Migrate (11, 36, {type=string, ref=0}, {type=Ø, ref=1}), Update (11, {type=string, ref=0}, {type=Ø, ref=1})
	deliveryInfo			Insert (38, ED)
	fullname			Insert (45, ED)
Facet (F)	maxExclusive, '100'	Delete (12, F)		
	maxExclusive, '50'	Insert (25, F)		Delete (14, F)
	pattern, '\d{3}-[A-Z]{2}'		Delete (24, F)	
	pattern, '\d{3}-[A-Z]{2}'		Insert (36, F)	
	minInclusive, '1'			Insert (37, F)
Simple Type (ST)	SKU		Delete (23, ST)	
	partNumType		Insert (35, ST)	

Table 3 (continued)

Component		V_{b-1}	V_{1-2}	V_{2-3}
Complex Type (CT)	USAddress		Insert (26, CT)	
	CT (Anonymous)			Insert (39, CT)
Model Group (MG)	sequence		Insert (27, MG)	
	sequence			Insert (40, MG)

4 XSM – high-level architecture

The main objective of XML Schema Monitor (XSM) system is to allow schema designers to monitor XSD versions. In XSM, different tasks can be performed not only to version XML Schemas but also to monitor the development process of those schemas. The main tasks are (1) *Version Insertion*, (2) *Versioning Comparison*, (3) *Version Retrieval*, and (4) *Delta Evaluation* shown in shaded boxes 1, 3, 4, and 5 in Figure 8, respectively.

4.1 Version insertion

In this task (box 1 in Figure 8), we explain how a new version should be added to the system. To simplify the process, Let R be an XML Schema repository where a collection of versions belong to one schema (i.e., shiporder in our running example), V_b is the initial version, and V_c is the current (last) version. The user adds a version V_{c+1} to R by performing the following steps:

- ***Step 1*** - Load version V_{c+1} of the schema and store it in R as a current version.
- ***Step 2*** - If V_{c+1} is the initial version (i.e., $V_b = \emptyset$), set V_{c+1} as both initial and current version of R (the inserted version are tagged in the repository relational table) and assign version id to V_{c+1}.
- ***Step 3*** - If V_{c+1} is not the initial version, set V_{c+1} as current version and create a tree representation of both V_{c+1} and its previous version V_c. This step includes parsing each version (using XSOM parser[2]), generating the corresponding *XML Schema Internal Representation* (box 2 in Figure 8), and storing nodes information to the corresponding relational tables. The internal representation of XSDs is constructed on the basis of *XML Schema Components Model*.[3]

At this point, a new version has been inserted and it is ready to be compared with its preceding version. The *Version Comparison* task is explained next.

4.2 Version comparison

By adding any two successive XSD versions to the system, the *Version Comparison* task (box 3 in Figure 8) then uses the parsed versions to match nodes of the same

[2] Available at: https://xsom.java.net/
[3] Available at: http://www.w3.org/TR/xmlschema-1/

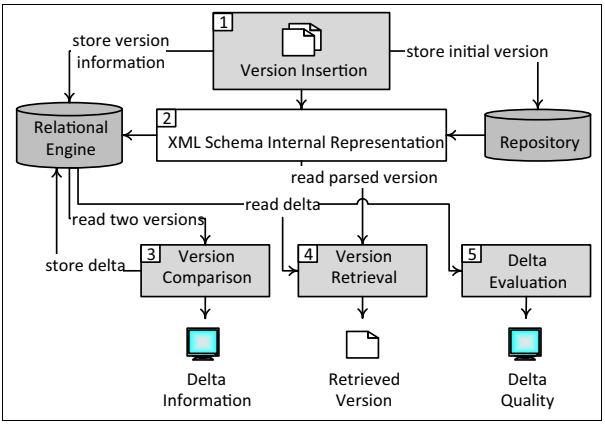


Figure 8 XSM high-level archeticture

type (i.e., element, attribute, or simpleType) and generates delta changes in the form of XS-Rel-Delta explained earlier. This task is further explained in our previous work [3] where the change detection algorithm XS-Diff is developed. At the versioning stage, we use our proposed storage technique (SFXS) to accommodate delta changes between the compared versions. This process is repeated with each new version added to the system. At the end, we only keep the first version V_b, and a set of deltas storing the history of changes to V_b. We will see how a version can be retrieved in the following section.

4.3 Version retrieval

In the *Version Retrieval* task (box 4 in Figure 8), we show how a specific version V_j can be generated from a set of n versions by using SFXS storage method. The process of querying an earlier version of an XML Schema is shown in Figure 9. It is a recursive procedure of retrieving the required version V_j by deriving a number of j intermediate versions starting from the available original one (V_b). In each turn, we read the input version (e.g., V_1) and apply XS-Rel-Delta (e.g., $V_{1\rightarrow 2}$) to create the next one (e.g., V_2). The process continues until we reconstruct the required version V_j. We propose an algorithm that is responsible for creating a specific version from the previous one. The pseudo-code of generate_schema_version algorithm is depicted in Table 4.

The steps in *Version Retrieval* task are as follows:

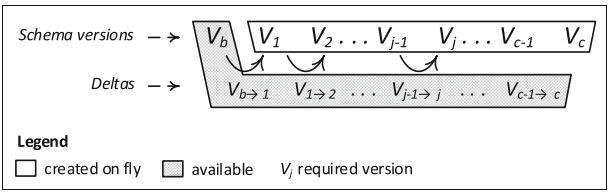


Figure 9 Versioning using SFXS storage technique, an intermediate version (V_j) can be reconstructed by starting from V_b

Table 4 generate_schema_version algorithm (Δ denotes XS-Rel-Delta)

```
01  Input: version V_i, 1 ≤ i ≤ n-1 & Δ_j, i+1 ≤ j ≤ n
02  Output: version V_j, i+1 ≤ j ≤ n

03  // Parse and create an XSOM tree model T_i for V_i
04  T_i ← V_i
05  // Phase 1. visit attributeGroup nodes (ag)
06  for each node ag ∈ T_i, do
07    if ag ∉ Δ_j(deleted ag)
08      insert node ag <attributeGroup...>
09      iterate_attribute_group_child_nodes(ag)
10  next ag
11  append_inserted_attributeGroup()
12  // Phase 2. visit attribute nodes (ad)
13  for each node ad ∈  T_i, do
14    if ad ∉ Δ_j(migrated, moved, or deleted ad)
15      if ad ∈ Δ_j(updated ad)
16        insert node ad, ad ∈ Δ_j <attribute...>
17      else
18        insert node ad <attribute...>
19      if ad_type ∉ Δ_j(l-t-g migration)
20        if ad_type ∈ Δ_j(g-t-l migration)
21          append_glob_to_loc_stype()
22        else if ad_type is local
23          visit_stype(ad_type)
24  next ad
25  append_ltg_attibute()
26  append_inserted_attribute()
27  // Phase 3. visit group nodes (gd)
28  for each node gd ∈ T_i, do
29    if gd ∉ Δ_j(deleted gd)
30      insert node gd <group...>
31      modelGroup(mg ← getModelGroup(gd))
32  next gd
33  append_inserted_group()
34  // Phase 4. visit element nodes (ed)
35  for each node ed ∈ T_i, do
36    if ed ∉ Δ_j(migrated, moved, or deleted ed)
37      if ed ∈ Δ_j(updated ed)
38        insert node ed, ed ∈ Δ_j <element...>
39      else
40        insert node ed <element...>
41      if ed_type ∉ Δ_j(l-t-g migration)
42        if ed_type ∈ Δ_j(g-t-l migration)
43          append_glob_to_loc_type()
44        else if ed_type is local
45          visit_type(ed_type)
46  next ed
47  append_ltg_element()
48  append_inserted_element()
49  // Phase 5. visit complexType nodes (ct)
50  for each node ct ∈ T_i, do
51    if ct ∉ Δ_j(migrated or deleted ct)
52      visit_ctype(ct)
53  next ct
54  append_ltg_ct()
55  append_inserted_ct()
56  // Phase 6. visit simpleType nodes (st)
57  for each node st ∈ T_i, do
58    if st ∉ Δ_j(migrated or deleted st)
59      visit_stype(st)
60  next st
61  append_ltg_st()
62  append_inserted_st()
63  // write the constructed tree T_i into V_j
64  V_j ← T_i
65  return V_j
```

- ***Step 1*** - Load the initial version V_b of the schema.
- ***Step 2*** - From the database, query versions that are needed to get the required version V_j. This step returns a list of versions information (e.g., data from version table).
- ***Step 3*** - Loop the versions falling between V_b and V_j and inspect V_i in each turn.
- ***Step 4*** - If version V_i is the initial version (V_b), call generate_schema_version algorithm and pass the initial version (V_b) and the delta to the next version ($V_{b\rightarrow 1}$). The result of this step is an on-fly generated version (V_{temp}), which is used in the following steps.
- ***Step 5*** - If version V_i is not the initial version (V_b), call generate_schema_version algorithm and pass version V_{temp} and the delta to its next version ($V_{temp \rightarrow temp+1}$). The result of this step is also an on-fly generated version (V_{temp}).
- ***Step 6*** - Continue the reconstruction until the required version (V_j) is reached.

Before starting, the original version is parsed and the corresponding XML Schema Internal Representation is generated, similar to ***Step 3*** in *Version Insertion* task. The idea behind the algorithm is that we traverse each node of the generated object model and examine whether it is *migrated*, *moved*, or *deleted* from their positions in the original version. At certain positions during the schema traversing, we check the possible *insertions* of the schema tree nodes. The delta information for performing the previous checks can be gathered by issuing a set of SQL queries on the delta tables. We rely on XSDL syntax [40] to examine possible occurrences of XML Schema components. The algorithm consists of six phases partitioned according to the type of the schema component. These phases are: (1) *iterate attribute groups*, (2) *iterate attribute declarations*, (3) *iterate groups*, (4) *iterate element declarations*, (5) *iterate complex type definitions*, and (6) *iterate simple type definitions*. In what follows, we look at each phase in more detail.

4.3.1 Phase 1 - Iterate attribute groups

This phase (lines 6–11 in Table 4) deals with attribute group definition (ag) components, tagged <attributeGroup> in XML Schema document. Since attribute groups can only be defined globally, there is one possible deletion of its node. Thus, deletion check is made based on this observation. The iterate_attribute_group_child_nodes procedure is then called to check possible occurrences of the attribute group child nodes, namely *attributes* (ad) and *attribute groups* (ag). Manipulated attributes in this procedure are called *attribute uses*. The attribute use can be an attribute with a local definition if it migrates from global to local or it can be as same as in its original situation, i.e., a reference to global attribute. At the end of this Phase, we *append* possible inserted components by calling append_inserted_attributeGroup procedure.

4.3.2 Phase 2 - Iterate attributes

In this phase (lines 13–26 in Table 4), we iterate global attribute declarations (ad) tagged <attribute>. In this case, the attribute may be migrated from global to local definition. Therefore, we perform a *migration* test along with *deletion* and *update* tests as seen in lines 14 and 15. We also check if there is a local simple type defined for the attribute. The visit_stype procedure is called at line 23 to perform the check. This procedure is explained in *Phase 4* (iterate simple types) where we inspect each individual simple type. Finally, two procedures append_ltg_attibute and append_inserted_attribute are called, as seen in lines 25 and 26 respectively. The first procedure queries the XS-Rel-Delta tables and checks if there are any attributes that have migrated from local to global. Following this, migrated attributes are added to the global definition. In the second procedure (line 26), we check possible inserted attributes as global declarations.

4.3.3 Phase 3 - Iterate groups

Group definitions (gd) in this phase (lines 28–33 in Table 4) is treated similar to attribute groups in Phase 1. The only difference is that instead of attributes and attribute groups as child nodes, there is only one possible model group (represented by <sequence>, <choice>, or <all> tags) exists as a child node of the group <group>. The procedure modelGroup is used for this purpose and it is shown in Table 5. Since the model group can possibly consist of one or more particles (i.e., *elements*, *model groups*, and/or *group definitions*) with changing orders, we will explain modelGroup procedure in the next paragraph. Similar to the previous phases, we check the inserted groups at the end of this phase at line 33 in Table 4.

Maintaining model group sequence and its children (particles) order changes One of the most important issues in the version retrieval task is how to handle model groups and their children order changes correctly. In XML Schema the node only considers the order of its children in the model group <sequence>, so we handle the ordering issue in modelGroup procedure and its related adjustOrder (Table 6 (a)) and indexOfSmallest (Table 6 (b)) procedures.

Table 5 modelGroup procedure

```
01  Input: modelGroup mg, mg ∈ Vi & delta Δj, i+1  j  n
02  Output: mg written to version Vj, i+1  j  n

03  // Step 1. model group order preparation
04  mgOreder ← 0
05  if getParent(mg) = 'sequence'
06    // get mg order from the parsed version Vi
07    mgOrder ← getChildCount(getParent(mg))
08  if mg == 'sequence'
09    // call SQL query to insert new record to num_ins_mov_del table
10    insert_new_sequence_record()
11  mgOrder ← adjustOrder(mgOrder)
12  // Step 2. model group check and node insertion
13  if mg ∉ Δj(moved or deleted mg)
14    if mg ∈ Δj(updated mg)
15      insert node mg, mg ∈ Δj <sequence..>, <choice..>, or <all..>
16    else
17      insert node mg <sequence..>, <choice..>, or <all..>
18    // Insert possible child nodes and parse particels
19    append_inserted_group()
20    append_inserted_modelGroup()
21    append_inserted_element()
22    for each i, where 0    index < getSize(mg), do
23      visit_particle(getChild(mg, i))
24    next i
25    // Step 3. reorder the child nodes using selection sort
26    index ← 0
27    indexOfNextSmallest ← 0
28    for each index, where 0    index < getChildCount(mg)-1, do
29      indexOfNextSmallest ← indexOfSmallest(index, mg, getChildCount(mg))
30      iNode ← getChildAt(mg, index)
31      jNode ← getChildAt(mg, indexOfNextSmallest)
32      insert(mg, iNode, indexOfNextSmallest)
33      insert(mg, jNode, index)
34    next index
35  // Step 4. call SQL query to update record in num_ins_mov_del table
36  else  // moved or deletes mg found
37    upd_num_mov_del_childs()
38  write inserted mg node and its children to Vj
```

Initially, we prepare the *order number* of the model group itself (since it can be a child of another model group) in ***Step 1*** of the procedure (lines 4–11 in Table 5). The model group's parent is tested so that we can get the last order of its children using a built-in function getChildCount. Then, we check the model group itself. If it is a sequence, then we insert a new record to the temporary table num_ins_mov_del used in the order change manipulation. The num_ins_mov_del table has the following relational schema: num_ins_mov_del (id, path, num_ins_mov, num_mov_del), where path is a unique path of each inspected sequence model group, num_ins_mov records the number of *inserted* or *moved-to* operations of the sequence node siblings, and num_mov_del records the number of *moved-from* or *deleted* operations of the sequence node siblings. This table is updated each time we *insert*, *delete*, or *move* one of the child nodes of any sequence node during the version generation. At the end of this step, we call adjustOrder procedure passing the old order so that we can return the model group to its original position (i.e., before performing any insertion, deletion, or move operations on its siblings). The idea behind this procedure is that we query num_ins_mov_del table using the path of the parsed model group. Then numInsMov is subtracted from the old model group order oldO and numMovDel is added to oldO as seen in line 5 in Table 6 (a).

In ***Step 2*** (lines 13–24 in Table 5) we check if the model group is *moved*, *deleted*, or *updated* and, based on that, we decide whether a tree node for the model group should be created or not. Next, as we did with the previous components, we append the inserted child nodes. In this case, they might be *groups*, *model groups*, or *elements* as seen in lines 19–21. We then run a loop to visit each child of the model group node. Again, the model group sequence children called *particles*, consisting of the three types mentioned above, are parsed by calling a built-in function visit_particle.

After parsing (and possibly adding) all model group child nodes, we rearrange them based on the new orders achieved by calling adjustOrder procedure in line 11. ***Step 3*** of the procedure (lines 26–34) is dedicated to sort the model group child nodes. We use a *selection sort*, which is an easy and straightforward algorithm to sort a list of values (usually an array) and efficient for small lists. The idea is to divide the list (an array of model group children in our case) into two parts: 1) a sublist of nodes that are already sorted, which starts from left to right, and 2) a sublist of nodes to be sorted, which occupies the rest of the array. The idea of the selection sort is to find the node with the smallest order indexOfNextSmallest (line 29 in Table 5) during the loop by invoking indexOfSmallest procedure (shown in Table 6 (b)). Then, we move the node with the indexOfNextSmallest to the sublist of sorted nodes by swapping the inserted nodes index and indexOfNextSmallest as seen in lines 32 and 33 in Table 5.

Table 6 Procedures for maintaning model group order changes

(a) `adjustOrder` procedure

```
01  Input: old order oldO & delta Δ_j,
           i+1  j  n
02  Output: new order newO

03  numInsMov ← query_num_ins_mov_childs(Δ_j)
04  numMovDel ← query_num_mov_del_childs(Δ_j)
05  newO ← oldO - numInsMov + numMovDel
06  return newO
```

(b) `indexOfSmallest` procedure

```
01  Input: start index si, node n, and number of
           child nodes numChilds
02  Output: index of a child with the smallest
           order indexOfMin

03  minNode ← getChildAt(n, si)
04  indexOfMin ← si
05  index ← 0
06  for each index, where si+1  index <numChilds, do
07    indexNode ← getChildAt(n, index)
08    if getLocOrder(indexNode)<getLocOrder(minNode)
09      indexOfMin ← index
10  next index
11  return indexOfMin
```

Finally in ***Step 4***, since the checked model group (as child of another model group) is found *deleted* or *moved* from its position (line 37 in Table 5), its record in num_ins_mov_del table (introduced in *Step 1*) is updated so that it can be used to fix the order of the next siblings. The update is done using the following SQL query: UPDATE num_ins_mov_del SET numMovDel = numMovDel + 1 WHERE path = <currPath>, where currPath is a current path of the parsed particle (e.g., model group in this case).

4.3.4 Phase 4 - Iterate elements

The process of treating element declarations (ed) (lines 35–48 in generate_schema_version algorithm in Table 4) is similar to that for treating attribute declarations. If an element is child of a sequence model group, then it is given an order and the order is adjusted as we did with the modelGroup procedure Step 1 in Table 5. All global elements (tagged as <element>) are visited, and *migrated*, *deleted*, or *moved* elements are ignored, so that they do not appear in the resulting version. If the element is *moved* or *deleted*, then the value of num_mov_del in num_ins_mov_del table is increased by 1. The parsed element is then checked for an update operation as seen in line 37. If the element is updated (i.e., one or more of its attributes has changed), then the algorithm adds the element node to the generated tree based on the information available from the delta. Since elements can have simple or complex type, we apply a special function visit_type, which reads the type definition from the schema object model and visits it properly. Complex and simple type visiting are explained in Phase 5 and 6, respectively.

After visiting the element node, the algorithm performs an append task by querying the delta and checks for elements inserted at a global position either by a local-to-global migration or by a normal *insert*. This check is done by the two procedures append_ltg_element and append_inserted_element at lines 47 and 48, respectively.

4.3.5 Phase 5 - Iterate complex types

In this phase, we iterate global complex types (lines 50–55 in Table 4). In each turn, if the complex type is not found in the group of *migrated* or *deleted* complex types, then visit_ctype procedure is called. The procedure (shown in Table 7) simply investigates the four possible content types of the complex type component: *simple content*, *empty content*, *particle*, and *complex content*. Based on the investigation, it performs a proper visit to complex type descendent nodes.

In the first content type case, i.e., simple content (lines 5–22 in Table 7), we examine a two derivation types of the simple content *restriction* and *extension*. If the derivation is by restriction, then we iterate a set of facets defined under the complex type (see lines 13–15). The procedure visit_facet is called in this situation, which will visit and inspect facet nodes (e.g., pattern or enumeration). Complex type nodes are created and added to the generated schema tree as seen in lines 11 and 22. In the second case (lines 24 and 25), we investigate the empty content type. In this case, we just add a complex type node and leave the child attributes to the end of the procedure. In the third case (lines 28 and 29), we consider a complex type with a particle, i.e., exactly one model group of type sequence, choice, or all. Thus, a complex type is created and inserted as in the previous case. Finally (Case 4 in lines 31–43), a complex content is treated similar to the simple content case. At the end, we invoke a visit_particle procedure (line 44) as we did in the modelGroup procedure above. Also, we append possible inserted groups and model groups.

Table 7 visit_ctype procedure

```
01  Input: complexType ct, ct      Vi & delta Δj, i+1  j  n
02  Output: ct & its descendant nodes written to Vj, i+1  j  n
03
04  // Case 1. Simple Content
05  if contType(ct) = simpleContent
06    if derivation(ct) = restriction
07      if ct ∈ Δj(updated ct)
08        insert ct & its descendant nodes, ct ∈ Δj
09      else
10        insert ct & its descendant nodes,
11      <complexType...><simpleContent><restriction...>
12      F ← iterateFacets(ct)
13      for each child node f ∈ F, do
14        visit_facet(f)
15      next f
16      append_inserted_facet()
17    else  // extension
18      if ct ∈ Δj(updated ct)
19        insert ct & its descendant nodes, ct ∈ Δj
20      else
21        insert ct & its descendant nodes,
22      <complexType...><simpleContent><extension...>
23  // Case 2. Empty content
24  else if contType(ct) = empty
25    insert node ct, <complexType...>
26  else
27    // Case 3. Particle
28    if baseType(ct) = 'anyType'
29      insert node ct, <complexType...>
30    // Case 4. Complex Content
31    else
32      if derivation(ct) = restriction
33        if ct ∈ Δj(updated ct)
34          insert ct & its descendant nodes, ct ∈ Δj
35        else
36          insert ct & its descendant nodes,
37        <complexType...><complexContent><restriction...>
38      else  // extension
39        if ct ∈ Δj(updated ct)
40          insert ct & its descendant nodes, ct ∈ Δj
41        else
42          insert ct & its descendant nodes,
43        <complexType...><complexContent><extension...>
44    visit_particle(contType(ct))
45    append_inserted_group()
46    append_inserted_modelgroup()

47  visit_ctype_attributes()
48  append_inserted_attribute()
49  append_inserted_attGroup()
50  write inserted ct and its descendant nodes to Vj
```

In all four complex type cases, we visit complex type attributes and attribute groups since they occur in all types of the complex type.

4.3.6 Phase 6 - Iterate simple types

The same process is repeated for simple type global definitions (lines 57–62 in Table 4), but in this case, the derivation method of the simple type, i.e., *restriction*, *list*, or *union,* is inspected. The pseudocode for the inspected simple type is listed in Table 8.

Table 8 visit_stype procedure

```
01  Input: simple type st, st ∈ V_i & delta Δ_j, i+1  j  n
02  Output: st & its descendant nodes written to V_j, i+1  j  n
03
04  insert st node <simpleType...>
05  // Case 1. Restriction
06  if derivation(st) = retsriction
07    if st ∈ Δ_j(updated st)
08      insert st restriction node, st ∈ Δ_j <restriction...>
09    else
10      insert st restriction node <restriction...>
11    if baseType(st) is local type
12      visit_stype(baseType(st))
13    F ← iterateFacets(st)
14    for each child node f ∈ F, do
15      visit_facet(f)
16    next f
17    append_inserted_facet()
18  // Case 2. List
19  else if derivation(st) = list
20    if st ∈ Δ_j(updated st)
21      insert st list node, st ∈ Δ_j <list...>
22    else
23      insert st list node <list...>
24    if itemType(st) is local type
25      visit_stype(itemType(st))
26  // Case 3. Union
27  else if derivation(st) = union
28    if st ∈ Δ_j(updated st)
29      insert st union node, st ∈ Δ_j <union...>
30    else
31      insert st union node <union...>
32      // parse local memberTypes
33      MT ← getMembers(st)
34      for each member type node mt ∈ MT, do
35        visit_stype(mt)
36      next mt
37  write inserted st and its descendant nodes to V_j
```

4.4 Delta evaluation

In the schema evolution process, the main concern is with build new schema versions from previously developed schemas and ensuring that schemas are easy to extend for adapting them to new circumstances. Schema developer may not only be interested in knowing the history of changes for a particular element or type in the schema repository, but also in ascertaining the efficacy of a particular delta in generating the new version. In this task, we define two indicators to measure the proportion of *reusability* and *extensibility* of the delta. We will explain each indicator in the following subsections.

4.4.1 Reusability indicator (RI)

One of the most important goals in the design of XML Schema is reusability. Reusing schema components is not only easier for developers to maintain but also saves time. In the Reusability indicator, denoted by (RI), if delta contains elements or attributes that migrate (by changing its scope) from local to global (LTG) declarations, the level of reusability in the generated version becomes higher. Moreover, the same type of

migration is applicable to the definition of simple or complex types. The other way round is also possible for all the previous components, but the reusability level will decrease when shifting them from global to local (GTL). The scope of newly inserted components also affects the reusability, meaning that a locally inserted (anonymous) component cannot be leveraged by other schema components which reduce the reusability.

We calculate RI based on the parameters listed in Table 9 (a). Note that the information on parameters is collected from delta Δ_j, its preceding schema version V_{j-1}, and its resulting version V_j. To define the reusability for complex and simple types (R_t), the average of reusability for both migrated and inserted types is calculated as follows.

$$R_t = \frac{1}{2}\left(\left(\frac{M_{gt}}{A_t} - \frac{M_{lt}}{G_t}\right) + \left(\frac{I_{gt}}{T_t} - \frac{I_{lt}}{T_t}\right)\right) \tag{1}$$

Similarly, to calculate the reusability for elements and attributes (R_c), the following function is defined.

$$R_c = \frac{1}{2}\left(\left(\frac{M_{gc}}{L_c} - \frac{M_{lc}}{G_c}\right) + \left(\frac{I_{gc}}{T_c} - \frac{I_{lc}}{T_c}\right)\right) \tag{2}$$

Named groups and attribute groups also affect the reusability of the generated schema version. A function for group and attribute group reusability (R_g) is defined as follows.

$$R_g = \left(\frac{I_g}{T_g}\right) \tag{3}$$

Generally, to measure RI, the equation is formulated by taking the average of the reusability results from Eqs. (1), (2), and (3):

$$RI = \frac{1}{3}\left(R_t + R_c + R_g\right) \times 100 \tag{4}$$

4.4.2 Extensibility indicator (EI)

Extensibility is another important schema design consideration. For example, a schema developer may design a message for a product description. If the description expands by adding more features, the schema needs to be redesigned to satisfy the new settings. For this reason, the new versions of the schema should allow a certain level of extensibility to handle feature type variations.

The Extensibility Indicator, denoted by (EI), of the delta can be calculated by measuring inheritance types (i.e., complex and simple type derivations). It is also possible to combine one or more types by using union simple type, or to restrict another type by using restriction simple type. To measure EI, we use the parameters listed in Table 9 (b).

We formulate the EI function by counting the average number of simple type and complex type components that derive or combine other types.

$$EI = \frac{1}{2}\left(\left(\frac{S_r + S_u}{T_{st}}\right) + \left(\frac{C_r + C_e}{T_{ct}}\right)\right) \times 100 \tag{5}$$

Values for both RI and EI are ranging from −100 to 100. The proposed indicators give a designer a measure of how well the developed version is. If the indicator value is closer to 100 then the delta quality is high, but if it is near −100 then the delta

Table 9 Reusability and extensibility indicators parameters

(a) Reusability (RI)			(b) Extensibility (EI)		
Parameter	Code	av*	Parameter	Code	av
Number of LTG migrated elements and attributes	M_{gc}	Δ_j	Number of inserted simple types with restriction	S_r	Δ_j
Number of GTL migrated elements and attributes	M_{lc}		Number of inserted simple types with union	S_u	
Number of LTG migrated complex and simple types	M_{gt}		Number of inserted complex types with restriction	C_r	
Number of GTL migrated complex and simple types	M_{lt}		Number of inserted complex types with extension	C_e	
Number of globally inserted elements and attributes	I_{gc}				
Number of locally inserted elements and attriutes	I_{lc}				
Number of globally inserted complex and simple types	I_{gt}				
Number of locally inserted complex and simple types	I_{lt}				
Number of globally inserted reusable groups	I_g				
Number of local elements and attributes	L_c	V_{j-1}			
Number of global elements and attributes	G_c				
Number of anonymous complex and simple types	A_t				
Number of global complex and simple types	G_t				
Total number of elements and attributes	T_c	V_j	Total number of simple types	T_{st}	V_j
Total number of complex and simple types	T_t		Total number of complex types	T_{ct}	
Total number of reusable groups	T_g				

* Availability

used to generate the new version is of a lesser quality. If the indicator value is around or equal to zero, this implies that there is no change of quality between the schema versions. We will see how these indicators are calculated in the next example.

productOrder1.xsd and productOrder2.xsd shown in Figure 10a and b are two versions of the same schema. First, we calculate RI for the delta changes between the two versions as follows. Note that the last section that calculates R_g is omitted because neither groups nor attribute groups does exist in this example.

$$RI = \frac{1}{2}\left(\frac{1}{2}\left(\left(\frac{0}{2}-\frac{2}{4}\right)+\left(\frac{0}{7}-\frac{1}{7}\right)\right)+\frac{1}{2}\left(\left(\frac{2}{17}-\frac{2}{4}\right)+\left(\frac{0}{22}-\frac{3}{22}\right)\right)\right)\times 100$$
$$= \frac{1}{2}\left(\frac{1}{2}(-0.5-0.1429)+\frac{1}{2}(-0.3824-0.1364)\right)\times 100 = -29.04\%$$

As seen in this example, the subtracted parts of the equation ($^2/_4$, $^1/_7$, $^2/_4$, and $^3/_{22}$) are always larger than the values of parts that are subtracted from. Therefore, RI is given a negative

```
01 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
02  <xs:attribute name="country" type="xs:NMTOKEN"
fixed="US"/>
03  <xs:element name="pOrder" type="POType"/>
04  <xs:element name="street" type="xs:string"/>
05  <xs:element name="comment" type="xs:string"/>
06  <xs:complexType name="Items">
07   <xs:sequence>
08    <xs:element name="item" minOccurs="0"
maxOccurs="100">
09     <xs:complexType>
10      <xs:sequence>
11       <xs:element name="productName" type="xs:string"/>
12       <xs:element name="quantity">
13        <xs:simpleType>
14         <xs:restriction base="xs:positiveInteger">
15          <xs:maxExclusive value="50"/>
16         </xs:restriction>
17        </xs:simpleType>
18       </xs:element>
19       <xs:element name="USPrice"  type="xs:decimal"/>
20       <xs:element ref="comment"   minOccurs="0"/>
21      </xs:sequence>
22      <xs:attribute name="partNum" type="partNumType"/>
23     </xs:complexType>
24    </xs:element>
25   </xs:sequence>
26  </xs:complexType>
27  <xs:complexType name="POType">
28   <xs:sequence>
29    <xs:element name="shipTo" type="USAddress"/>
30    <xs:element name="billTo" type="USAddress"/>
31    <xs:element name="items"  type="Items"/>
32   </xs:sequence>
33   <xs:attribute name="orderDate" type="xs:date"/>
34  </xs:complexType>
35  <xs:complexType name="USAddress">
36   <xs:sequence>
37    <xs:element name="name"   type="xs:string"/>
38    <xs:element ref="street"/>
39    <xs:element name="city"  type="xs:string"/>
40    <xs:element name="state"   type="xs:string"/>
41    <xs:element name="zip"    type="xs:decimal"/>
42   </xs:sequence>
43   <xs:attribute ref="country"/>
44   <xs:attribute name="countryCode" type="xs:string"/>
45  </xs:complexType>
46  <xs:simpleType name="partNumType">
47   <xs:restriction base="xs:string">
48    <xs:pattern value="\d{3}-[A-Z]{2}"/>
49   </xs:restriction>
50  </xs:simpleType>
51 </xs:schema>
```

(a) productOrder1.xsd

```
01 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
02  <xs:element name="pOrder">
03   <xs:complexType>
04    <xs:sequence>
05     <xs:element name="shipTo" type="USAddress"/>
06     <xs:element name="billTo" type="USAddress"/>
07     <xs:element name="orderInfo">
08      <xs:complexType>
09       <xs:simpleContent>
10        <xs:extension base="xs:positiveInteger">
11         <xs:attribute name="orderID" type="xs:token"/>
12        </xs:extension>
13       </xs:simpleContent>
14      </xs:complexType>
15     </xs:element>
16     <xs:element name="items"  type="Items"/>
17    </xs:sequence>
18    <xs:attribute ref="orderDate"/>
19   </xs:complexType>
20  </xs:element>
21  <xs:element name="comment" type="xs:string"/>
22  <xs:element name="productName" type="xs:string"/>
23  <xs:attribute name="orderDate" type="xs:date"/>
24  <xs:complexType name="Items">
25   <xs:sequence>
26    <xs:element name="item" minOccurs="0"
maxOccurs="100">
27     <xs:complexType>
28      <xs:sequence>
29       <xs:element ref="productName"/>
30       <xs:element name="quantity">
31        <xs:simpleType>
32         <xs:restriction base="xs:positiveInteger">
33          <xs:minInclusive value="1"/>
34         </xs:restriction>
35        </xs:simpleType>
36       </xs:element>
37       <xs:element name="USPrice"  type="xs:decimal"/>
38       <xs:element ref="comment"   minOccurs="0"/>
39      </xs:sequence>
40      <xs:attribute name="partNum">
41       <xs:simpleType>
42        <xs:restriction base="xs:string">
43         <xs:pattern value="\d{3}-[A-Z]{2}"/>
44        </xs:restriction>
45       </xs:simpleType>
46      </xs:attribute>
47     </xs:complexType>
48    </xs:element>
49   </xs:sequence>
50  </xs:complexType>
51  <xs:complexType name="USAddress">
52   <xs:sequence>
53    <xs:element name="fullname"   type="xs:string"/>
54    <xs:element name="street" type="xs:string"/>
55    <xs:element name="state"  type="xs:string"/>
56    <xs:element name="city"   type="xs:string"/>
57    <xs:element name="zip"    type="xs:integer"/>
58   </xs:sequence>
59   <xs:attribute name="country" type="xs:NMTOKEN"
fixed="US"/>
60  </xs:complexType>
61 </xs:schema>
```

(b) productOrder2.xsd

Figure 10 Two successive versions of productOrder schema (changes are highlighted)

number (−29.04 %). It is clear that the large number of components migrated to the local definition or inserted locally, with respect to the total number of schema components, will negatively affect the value of RI. Because of the negative number given in this example, a possible suggestion to the designer is to redesign the second version to avoid a low level of reusability.

Now, we apply the EI in our running example.

$$EI = \frac{1}{2}\left(\left(\frac{0+0}{2}\right) + \left(\frac{0+1}{5}\right)\right) \times 100 = \frac{1}{2}\left(0 + \frac{1}{5}\right) \times 100 = 0.1\%$$

EI for the delta in this example is affecting the generated schema version with a positive value of 0.1 %. This can be explained by the insertion of a complex type with extension to orderInfo element to give it a type as seen in Figure 10b. The new insertion allows inheriting from another built-in type positiveInteger, thus, increasing the extensibility feature. Although the value of EI is positive, it is still very low and it is recommended that the designer consider enhancing the extensibility in the new schema version.

5 Experimental evaluation

In this section, we first evaluate the *correctness* of the versioning system. The correctness can be measured by comparing deltas produced by the system with the optimal deltas manually calculated for XML Schema. Then, we measure the *functionality* and the *correctness* of the delta quality indicators proposed in the previous sections. This is done by applying RI and EI indicators to datasets with several versions and checking how the indicator values can affect the design of new versions.

5.1 Experimental settings and datasets

To prove the correctness of the proposed versioning method, the optimality of the produced delta and the usability of delta quality indicators, we develop a tracking tool called XML Schema Monitor (XSM) using JAVA programming and XSOM parser. The prototyped tool uses SQL queries on MySQL 5.5.24-log RDBMS to store and retrieve XS-Rel-Deltas. We conduct all experiments on a computer running an Intel Core i7 2.30 GHz processor with 8 GB of memory and Windows 7 Home Premium as the operating system.

At this time, we are not aware of any available tools for XML Schema versioning or change control that can be compared with our tool. We do not compare our method to XML document versioning methods for the following reasons:

1. The space used to store XML Schema changes is not significant unlike XML documents. Our exploration of 40 real-world XML Schemas and standards[4] reveals that 85 % of those schemas are below 200 kb in size, 5 % are between 200 and 800 kb, and only 10 % are above 800 kb. Obvious, the complexity of space and memory used in the versioning process is not a real concern in the case of XML Schemas.

[4] Available at: https://docs.google.com/document/d/1mkJmt28f100DwPqCa8QKtrKWWhDilC1oBUI1pH9xh-M/edit?usp=sharing

2. The speed to perform the versioning is not important too because XML Schemas, in general, have few versions unlike XML documents. For example, a papinet standard[5] for the paper and forest supply chain releases two versions of their schemas every year starting from 2009. Similarly, the total number of schema versions published by OpenTravel standards[6] starting from 2001 is estimated to be 26. As seen in the previous examples, the demand of better management and understanding of schema changes is more important than the speed and storage in XML Schema versioning.

On considering the other methods that manipulate XML schemas, we only found the change detection method for the DTD schema language called DTD-Diff [22]. This approach is designed for a different purpose that aims to find changes between DTD files and use them to revalidate related XML documents. This means that the versioning aspect is absent in the approach, i.e., this method does not address how a set of deltas is stored and how a version is retrieved. Moreover, the generated DTD changes are different to those in XML Schema.

In order to evaluate the correctness and optimality of our versioning approach, a group of XSD datasets have been used. The original versions of these datasets are taken from the examples in [39] and [42]. These examples are considered complete since they reflect a high level of practicing of XML Schema recommendation. For each dataset we focus on specific component changes and manually generate five versions representing all changes to that component. A summary and characteristics of this group is shown in Table 10.

5.2 Versioning evaluation

5.2.1 Versioning correctness

The concept of correctness is critical in the design of XSM tool. The correctness in our context of versioning means that the versioning tool can retrieve the complete version using just the first version and a set of deltas to the required version. For that purpose, we first accommodate the six versions for all datasets (listed in Table 10) by running the change detector algorithm XS-Diff. Then, for each dataset we query the stored versions by running the version retrieval algorithm. For all datasets, XSM is able to reconstruct versions with no missing information. That is, in each particular retrieve, we compare the resulting version with the original one. Regardless of some differences in the appearance of schema global components, the compared schemas will still have the same meaning for describing XML documents.

5.2.2 Delta optimality

In this test, we move step further by evaluating the optimality of the delta used for the versioning. Using the same XSD datasets listed in Table 10, we investigate how deltas generated by XSM are close to optimal deltas. The optimal deltas for XML Schema are calculated manually by adopting the traditional operations *insert*, *delete*, *update*, and *move* that are supported by the majority of the previous works on XML change detection. The results of this comparison are depicted in Figure 11.

The results show that in the majority of datasets, XSM tool produces deltas that are optimal or near optimal. The equality between XSM operations and the optimal operations gives a

[5] Available at http://www.papinet.org/#the-standard/previous-versions/ajax.html
[6] Available at http://www.opentravel.org/Specifications/PastSpecs.aspx

Table 10 XSD datasets

Dataset	Number of components during 6 versions (V_b-V_1-V_2-V_3-V_4-V_5)								Focus area
	ED	AD	CT	F	ST	MG	GD	AG	
library	14-23-15-14-14-14	3-8-4-2-2-2	4-5-5-5-5-5	0-0-0-0-0-0	0-0-0-0-0-0	4-4-4-4-4-4	0-0-0-0-0-0	0-0-0-0-0-0	Element and attribute changes
namedGroup	9-9-11-9-10-10	6-5-7-5-4-4	3-2-2-2-2-2	0-0-0-0-0-0	0-0-0-0-0-0	6-6-7-6-6-6	7-7-9-7-7-7	5-5-8-6-6-6	Group and attribute group changes
complexType	13-14-16-16-14-14	9-9-10-10-9-9	9-9-11-11-9-9	2-2-4-4-2-2	0-0-0-0-0-0	5-5-6-6-5-5	0-0-0-0-0-0	0-0-0-0-0-0	Complex type changes
simpleType	6-8-6-7-6-6	1-1-1-1-1-1	0-1-1-1-1-1	9-9-9-11-11-11	6-6-6-7-6-6	0-1-1-1-1-1	0-0-0-0-0-0	0-0-0-0-0-0	Simple type and facet changes
modelGroup	5-5-9-5-5-5	0-0-0-0-0-0	1-1-2-1-1-1	0-0-0-0-0-0	0-0-0-0-0-0	1-1-3-2-2-2	0-0-0-0-0-0	0-0-0-0-0-0	Model group changes
shiporder	8-10-16-17-17-17	4-3-5-6-6-6	3-3-4-5-5-5	2-2-2-2-2-2	2-2-2-2-2-2	3-3-4-5-6-6	0-0-0-0-2-2	0-0-0-0-2-2	Mixed changes for all components

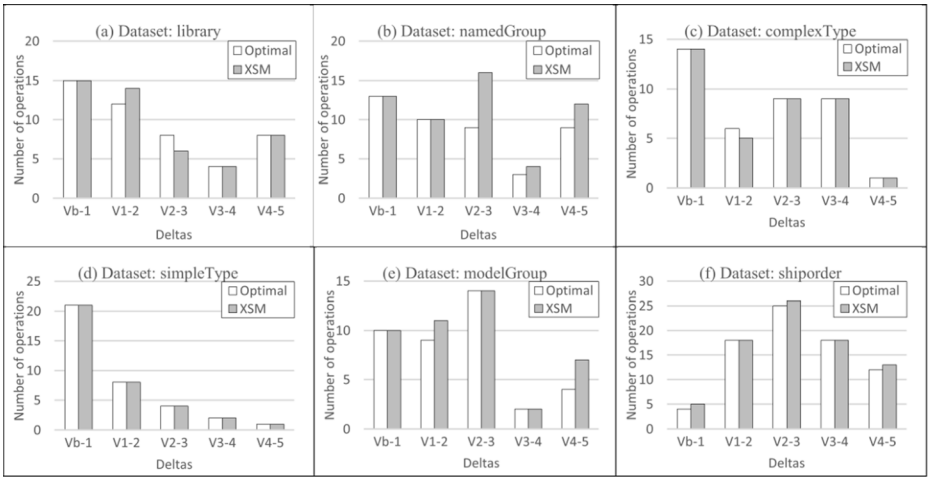


Figure 11 Results of the quality for five datasets listed in Table 10

positive indication that our tool can effectively be used to version XML Schemas, though XSM has some deficiencies. For example, deltas produced in datasets *namedGroup* and *modelGroup* (Figure 11b and e, respectively) in some situations exceed the optimal deltas by few operations (i.e., V_{2-3}, V_{3-4}, and V_{4-5} in *namedGroup* dataset and V_{1-2} and V_{4-5} in *modelGroup* dataset). In these datasets we focus on named groups (group and attributeGroup components) and model groups (sequence, choice, and all), which implicitly contain a sequence model group compositor in the component structure. Recall that in XML Schema context, we maintain order changes only for sequence child nodes. Consequently, the operations produced by XSM in both (b) and (e) cases include extra update operations to maintain those order changes.

On the other hand, XSM generates optimal deltas in case of complex and simple types as seen in datasets *complexType* and *simpleType* (Figure 11c and d). This is because in both datasets, child components, such as facets for simpleType and model group for complexType, always move in case a migration of simple/complex type is encountered. In other words, there is no need for extra insertions or deletions to complete the transformation. In the case of element and attribute operations including migration (as seen in Figure 11a and f), the delta generated by XSM moves above and below the optimal delta. Extra operations in most deltas relate to an order update and are caused by the insert/delete of the neighbour preceding the components. In contrast, XSM deltas are less than optimal delta as seen in delta V_{2-3} in Figure 11a, where move and update operations are summed up in one operation *migrate*.

5.3 Delta indicators correctness

Two datasets are used in this set of experimentations *mails*[7] and *lib*.[8] We apply RI and EI delta quality indicators using Eqs. (4) and (5), respectively, on both datasets to see how the delta affects the versioning process. In the first dataset *mails*, we generate a set of eight versions manually since we are not aware of any publicly available XML Schema version generator

[7] Available at https://svn.osgeo.org/geotools/branches/2.2.x_js15/module/sample-data/src/org/geotools/test-data/xml/mails.xsd

[8] The base schema is available at http://docstore.mik.ua/orelly/xml/schema/ch03_01.htm

XSDs	Parameters													% Of change[a]
	M_{gt}	M_c	M_{gt}	M_{ti}	I_{gc}	I_{ic}	I_{gt}	I_{ti}	I_{ig}	S_t	S_{ti}	C_t	C_e	
mails1-2	0	2	1	0	0	2	1	0	1	1	0	0	0	5.48
mails1-3	1	3	1	1	0	11	4	0	2	2	1	0	0	17
mails1-4	2	8	1	4	0	12	6	0	2	2	2	0	1	30.65
mails1-5	2	6	1	5	0	15	6	0	3	2	0	2	1	44.23
mails1-6	2	6	1	6	0	25	13	2	3	7	4	2	1	58.96
mails1-7	2	10	1	15	0	40	17	6	3	10	4	3	3	80.67
mails1-8	2	10	1	15	4	125	20	22	10	13	4	6	7	92.52

[a] Percentage of change from the original version "mails1"

(a) Delta parameters for "mails" dataset

XSDs	Parameters													Switch[b]
	M_{gt}	M_c	M_{gt}	M_{ti}	I_{gc}	I_{ic}	I_{gt}	I_{ti}	I_{ig}	S_t	S_{ti}	C_t	C_e	
lib1-2	0	92	0	0	0	0	0	0	0	0	0	0	0	GE-VB
lib1-3	0	0	0	37	0	0	0	0	0	0	0	0	0	GE-SS
lib1-4	0	92	0	1	0	0	0	34	0	12	0	0	3	GE-RD
lib2-1	92	0	0	0	0	0	0	0	0	0	0	0	0	VB-GE
lib2-3	92	0	0	1	0	34	0	33	0	12	0	0	4	VB-SS
lib2-4	0	0	0	1	0	38	0	34	0	11	0	0	4	VB-RD
lib3-1	0	0	37	0	0	0	0	0	0	0	0	0	0	SS-GE
lib3-2	0	92	1	0	0	0	0	0	0	0	0	0	0	SS-VB
lib3-4	0	92	0	0	0	0	0	0	0	0	0	0	0	SS-RD
lib4-1	92	0	1	0	0	0	0	0	0	0	0	0	0	RD-GE
lib4-2	0	0	1	0	0	22	26	0	0	12	0	0	2	RD-VB
lib4-3	92	0	0	0	0	0	0	0	0	0	0	0	0	RD-SS

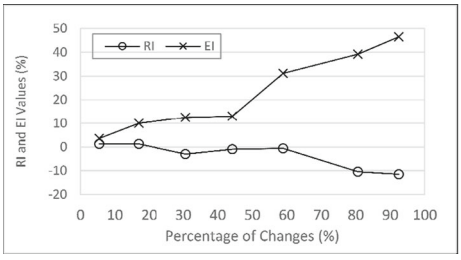
[b] Switch between design patterns: Garden of Eden (GE), Venetian Blind (VB), Salami Slice (SS), and Russian Doll (RD)

(b) Delta parameters for "lib" dataset

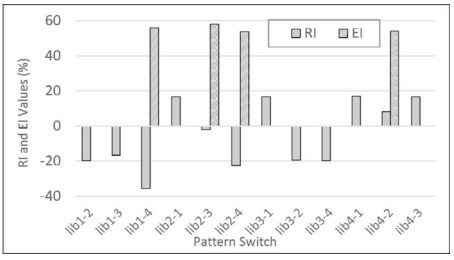
Figure 12 Delta parameter values

tool. Values of delta parameters in this test are shown in Figure 12a. The aim of this test is to see how RI and EI are influenced by the percentage and type of changes in the schema version. Results are plotted in Figure 13a. As clearly seen in this figure, RI and EI have positive values and are almost similar at the beginning of the test (5.48 % of change). As the schema evolved, more changes are incorporated which increase EI and decrease RI. The reusability of delta used to create the version at 44.23 % of change becomes slightly higher as shown by the rise in the RI indicator at that point. After 58.96 % of changes, RI moves into the negative direction while EI moves the opposite way. That means, while the extensibility of the schema enhances during its lifespan, a careful attention must be paid to the reusability of the schema as it tends to diminish.

We also consider the transformation between different schema design patterns: *Garden of Eden*, *Venetian Blind*, *Salami Slice*, and *Russian Doll* discussed by [42]. We want to examine the impact of pattern switch on RI and EI indicator values. The *lib* dataset is used in this test. The values of the delta parameters are listed in Figure 12b. The lowest value of RI (as seen in Figure 13) is −35 % at *lib1-4* where a lot of schema components are either migrating from global to local or are inserted as local definitions (see underlined values in Figure 12b). On the other hand, RI has a positive value with 17 % at *lib2-1*, *lib3-1*, *lib4-1*, and *lib4-3*. This is because the transformation between any pattern to the *Garden of Eden* will result in moving all schema components from the local to global definitions (values shaded in Figure 12b), which means that reusability will increase. We notice that EI is equal to ZERO in all switches except *lib1-4*, *lib2-3*, *lib2-4*, and *lib4-2*. EI indicator, in particular, shows a maximum value at *lib2-3*



(a) RI and EI values vs percentage of changes



(b) RI and EI values vs pattern switch

Figure 13 RI and EI results

switch, where the largest number of type restrictions and extensions are performed, thus, increasing the extensibility of the schema.

The benefit from the usage of RI and EI indicators is twofold. First, they provide an overview of the quality of the delta used to generate the version. Second, RI and EI indicators help the schema designer to understand the factors that impact the decrease or increase of schema quality. For example, in our previous dataset *mails*, the first drop of RI is at 30.65 % of change; at that point, I_{lc} (with 12) is the parameter that most affects RI followed by M_{lc} (with 8), as seen in *mails1-4* in Figure 12a. As a guideline, it is recommended that the designer alter the version until it reaches a reasonable positive value.

6 Conclusion and future work

In recent years, the maturity of cloud technology had influenced more collaborative works. Many providers of cloud collaboration tools have created solutions to solve collaboration needs, such as real-time editing, synchronisation, and retrieval of shared files. Documents written in XML Schema language require manipulation and consistency check during its development lifecycle. In this work, we take the cloud collaboration a level up by developing an approach to analyse XML Schema versions in the cloud, maintain their compatibility, and provide a useful information about version changes to the development team. We propose a versioning model and an algorithm for XML Schema versions. The developed algorithm is based on the schema object model that understands the unique structure of XML Schemas. To proof its feasibility, we design XSM, a tool capable of storing and monitoring XML Schema versions. It is also capable of detecting changes between successive versions, and measuring the quality of delta generated. Our target scope in this work is schema designers and groups who work on developing XML Schema standards through cloud environment. The main tasks of XSM include essential functions to any versioning system, such as *adding*, *comparing*, and *retrieving* a version from the repository. The versioning tasks also include a method for *measuring the quality* of the generated delta. We defined the two indicators of RI for reusability and EI for extensibility for that purpose. These indicators were shown to be useful for enhancing the version i.e., they were able to show when the processed delta needs to be further enhanced to generate a good quality schema.

We experimentally evaluated the system with a group of XSDs. The results showed the success of the proposed framework in generating the correct version from the initial version and a set of deltas. In addition, the deltas stored by XSM framework were found to be optimal or near optimal, which is important for recognizing and easily maintaining XML Schema changes. As future work, we plan to add more functions to the system such as providing suggestions on how to alter the schema to produce a better version. We will also consider more parameters for measuring the delta quality and readability.

References

1. acord.org: ACORD Standards, https://www.acord.org/standards/Downloads/Pages/default.aspx
2. Altova: Authentic – XML authoring tool, http://www.altova.com/authentic/xml-authoring-tool.html
3. Baqasah, A., Pardede, E., Holubova, I., Rahayu, W.: On change detection of XML Schemas. In: Proceedings of the 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom'13). pp. 974–982. IEEE Computer Society (2013)

4. Baqasah, A., Pardede, E., Rahayu, W.: XSM - A tracking system for XML Schema versions. In: 2014 I.E. 28th International Conference on Advanced Information Networking and Applications (AINA'14). pp. 1081–1088. IEEE (2014)
5. Basci, D., Misra, S.: Measuring and evaluating a design complexity metric for XML schema documents. J. Inf. Sci. Eng. **25**, 1405–1425 (2009)
6. Brahmia, Z., Bouaziz, R., Grandi, F., Oliboni, B.: Schema versioning in tXSchema-based multitemporal XML repositories. In: Proceedings of fifth International Conference on Research Challenges in Information Science (RCIS 2011). pp. 1–12, Gosier, Guadeloupe, France (2011)
7. Brahmia, Z., Bouaziz, R.: Schema versioning in multi-temporal XML databases. In: Proceedings of the Seventh IEEE/ACIS International Conference on Computer and Information Science (ICIS 2008). pp. 158–164. IEEE Computer Society (2008)
8. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F. cois: Extensible Markup Language (XML) 1.0 (Third Edition), W3C Recommendation. W3C, http://www.w3.org/TR/2004/REC-xml-20040204 (2004)
9. Camacho-Rodriguez, J., Colazzo, D., Manolescu, I.: Building large XML stores in the Amazon cloud. In: IEEE 28th International Conference on Data Engineering Workshops (ICDEW'12). pp. 151–158. IEEE (2012)
10. Camacho-Rodríguez, J., Colazzo, D., Manolescu, I.: Web data indexing in the cloud: efficiency and cost reductions. In: Proceedings of the 16th International Conference on Extending Database Technology (EDBT'13), pp. 41–52. ACM Press, New York (2013)
11. Chang, Y.-S., Yang, C.-T., Luo, Y.-C.: An ontology based agent generation for information retrieval on cloud environment. J Univers Comput Sci **17**, 1135–1160 (2011)
12. Chien, S.-Y., Tsotras, V.J., Zaniolo, C.: Efficient schemes for managing multiversion XML documents. VLDB J.—Int. J. Very Large Data Bases **11**, 332–353 (2002)
13. Currim, F., Currim, S., Dyreson, C., Snodgrass, R.T.: A tale of two schemas: creating a temporal XML schema from a snapshot schema with tXSchema. Advances in Database Technology-EDBT 2004. pp. 348–365. Springer (2004)
14. ebxml.org: ebXML Specifications, http://www.ebxml.org/specs/index.htm
15. Ignat, C.-L., Norrie, M.: Flexible collaboration over XML documents. In: Luo, Y. (ed.) Cooperative Design, Visualization, and Engineering, pp. 267–274. Springer, Berlin (2006)
16. JAVA: XML Schema Object Model (XSOM), http://xsom.java.net/
17. Klettke, M., Schneider, L., Heuer, A.: Metrics for XML document collections. XML-Based Data Management and Multimedia Engineering—EDBT 2002 Workshops. pp. 15–28. Springer (2002)
18. Leonardi, E., Bhowmick, S.S.: Xandy: a scalable change detection technique for ordered XML documents using relational databases. Data Knowl Eng **59**, 476–507 (2006)
19. Leonardi, E., Bhowmick, S.S., Madria, S.: Xandy: detecting changes on large unordered XML documents using relational databases. Database Systems for Advanced Applications. pp. 711–723 (2005)
20. Leonardi, E., Bhowmick, S.S.: Detecting changes on unordered XML documents using relational databases: a schema-conscious approach. In: Proceedings of the 14th ACM International Conference on Information and Knowledge Management (CIKM'05). pp. 509–516 (2005)
21. Leonardi, E., Bhowmick, S.S.: Oxone: a scalable solution for detecting superior quality deltas on ordered large XML documents. In: Proceedings of the 25th International Conference on Conceptual Modeling. pp. 196–211 (2006)
22. Leonardi, E., Hoai, T.T., Bhowmick, S.S., Madria, S.: DTD-Diff: a change detection algorithm for DTDs. Data Knowl Eng **61**, 384–402 (2007)
23. Marian, A., Abiteboul, S., Cobena, G., Mignet, L.: Change-centric management of versions in an XML warehouse. In: Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01). pp. 581–590. Morgan Kaufmann Publishers Inc. (2001)
24. McDowell, A., Schmidt, C., Yue, K.: Analysis and metrics of XML Schema. In: Proceedings of the International Conference on Software Engineering Research and Practice (SERP'04). pp. 538–544. CSREA Press (2004)
25. Microsoft: XML Data (SQL Server), http://technet.microsoft.com/en-us/library/bb522446.aspx
26. OpenTravel: OpenTravel Specifications, http://www.opentravel.org/Specifications/Default.aspx
27. Oracle.com: Using Oracle XML DB, http://docs.oracle.com/cd/B28359_01/appdev.111/b28369/xdb03usg.htm
28. oxygenxml.com: oXygen XML editor: collaborative authoring using subversion, http://www.oxygenxml.com/demo/Collaborative_Authoring_Using_Subversion.html
29. Rönnau, S., Borghoff, U.M.: XCC: change control of XML documents. Comput Sci Dev **27**, 95–111 (2012)

30. Rusu, L.I., Rahayu, W., Taniar, D.: Maintaining versions of dynamic XML documents. Web Inf. Syst. Eng. – WISE **3806**, 536–543 (2005)
31. Saracco, C.M., Chamberlin, D., Ahuja, R.: DB2 9 pureXML overview and fast start. IBM Redbooks (2006)
32. sdl.com: SDL LiveContent create: all the power of XML, http://www.sdl.com/products/livecontent/create.html
33. serna-xmleditor.com: SERNA XML editor: improve collaboration, http://www.serna-xmleditor.com/benefits/improve-collaboration/
34. Skaf-Molli, H., Molli, P., Rahhal, C., Naja-Jazzar, H.: Collaborative writing of XML documents. In: 3rd International Conference on Information and Communication Technologies: From Theory to Applications (ICTTA'08). pp. 1–6. IEEE (2008)
35. Sun, Y., Lambert, D., Uchida, M., Remy, N.: Collaboration in the cloud at Google. In: Proceedings of the 2014 ACM conference on Web science (WebSci'14), pp. 239–240. ACM Press, New York (2014)
36. Sundaram, S., Madria, S.K.: A change detection system for unordered XML data using a relational model. Data Knowl Eng **72**, 257–284 (2012)
37. Thao, C., Munson, E.V.: Using versioned tree data structure, change detection and node identity for three-way XML merging. In: Proceedings of the 10th ACM Symposium on Document Engineering (DocEng'10), pp. 77–86. ACM Press, New York (2010)
38. Thaw, T.Z., Khin, M.M.: Measuring qualities of XML Schema documents. J Softw Eng Appl **6**, 458–469 (2013)
39. Vlist, E.: XML Schema: The W3C's Object-Oriented Descriptions for XML. O'Reilly Media, Inc., Cambridge (2002)
40. W3C: W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures, http://www.w3.org/TR/xmlschema11–1/
41. W3C: XML Schema Part 0: Primer Second Edition, http://www.w3.org/TR/xmlschema-0/
42. Walmsley, P.: Definitive XML Schema. Prentice Hall (2012)
43. Wong, R.K., Lam, N.: Managing and querying multi-version XML data with update logging. In: Proceedings of the 2002 ACM Symposium on Document Engineering, pp. 74–81. ACM, McLean (2002)