Int. J. Web and Grid Services, Vol. X, No. Y, XXXX

XS-Diff: XML schema change detection algorithm

Abdullah Baqasah, Eric Pardede* and Wenny Rahayu

Department of Computer Science and Computer Engineering,
La Trobe University,
Victoria 3086, Australia
Email: ambaqasah@students.latrobe.edu.au
Email: e.pardede@latrobe.edu.au
Email: w.rahayu@latrobe.edu.au
*Corresponding author

Irena Holubova

Department of Software Engineering,
Charles University,
Prague, Czech Republic
Email: irena.mlynkova@gmail.com

Abstract: Detecting changes in XML data has emerged as an important research issue in the last decade, but the majority of change detection algorithms focus on XML documents rather than on their schemas because documents that contain data are deemed more significant than the schema itself. However, the XML schema change detection tool is essential, especially in situations where we need to maintain related XML documents with evolving schema, sustain relational schema generated by schema-conscious approach for storing XML data and provide support for XML versioning. This paper focuses on XML Schema (XSD) changes and provides a more meaningful description of the detected changes. Our proposed algorithm XS-Diff uses the technique of storing XML Schema versions in a relational database where the detection and storage of delta changes are employed on relational tables. We demonstrate the correctness of the proposed algorithm through both synthetic and real data sets without deteriorating the execution time.

Keywords: change detection; XML schema; XML data modelling; differencing algorithm.

Reference to this paper should be made as follows: Baqasah, A., Pardede, E., Rahayu, W. and Holubova, I. (XXXX) 'XS-Diff: XML schema change detection algorithm', *Int. J. Web and Grid Services*, Vol. X, No. Y, pp.xxx–xxx.

Biographical notes: Abdullah Baqasah is currently a PhD Candidate at the Department of Computer Science and Computer Engineering, La Trobe University, Australia, and completed the Master of Information Technology (MIT) at La Trobe University in 2009. He received his Bachelor's degree from the Department of Computer Science at King Abdulaziz University, Saudi Arabia, in 2003.

Copyright © 200X Inderscience Enterprises Ltd.

Eric Pardede is currently a Lecturer in the Department of Computer Science and Computer Engineering at La Trobe University, Australia. He completed PhD in Computer Science and Master of Information Technology from La Trobe University in 2007 and 2002, respectively.

Wenny Rahayu is currently a Professor and the Head of School of Engineering and Mathematical Sciences at La Trobe University. Prior to this appointment, she was the Head of Computer Science and Computer Engineering department. She received the PhD degree in Computer Science from La Trobe University in 2000. In the last ten years, she has published two authored books, three edited books and more than 150 research papers in international journals and proceedings. She has supervised to completion ten PhDs, around 30 Honours and ten Master's students.

Irena Holubova is currently an Assistant Professor in the Department of Software Engineering at Charles University, Prague. She received her PhD degree in Computer Science in 2007 and Master's degree in Computer Science in 2005. She is a Member of XML and Web Technology Research Group (XRG) since 2009.

This paper is a revised and expanded version of a paper entitled 'On change detection of XML schemas' presented at the '11th IEEE International Symposium on Parallel and Distributed Processing with Applications', Melbourne, 16–18 July 2012.

1 Introduction

EXtensible Markup Language (XML) has been widely used for representing, storing and manipulating data from different data sources. Due to its dynamic property, XML data tend to change from time to time in different ways, and a system is required to properly detect possible changes. As a consequence, there has been an increasing number of studies dedicated to the detection of changes in XML documents (Cobena et al., 2002b; Wang et al., 2003; Al-Ekram et al., 2005; Leonardi and Bhowmick 2005) or hierarchically structured documents in general (Chawathe et al., 1996; Chawathe and Garcia-Molina, 1997). In some situations, rules imposed by XML schema languages, such as Document Type Definitions (DTDs) (W3C, 2000) or XML Schemas (XSDs) (W3C, 2004a), are utilised to specify and enforce the XML document structure. However, these schemas may also change over time to reflect real-world changes and changes in the user's requirements, or to correct mistakes in the initial design. Moreover, complex structures (such as complexType in XML Schema) are often created but not fully configured to allow a user to restrict or expand them in future releases (Guerrini et al., 2005b).

A method to detect changes to XML Schema serves many purposes, such as the following:

- *Efficient maintenance (revalidation) of associated XML documents when their schema evolves*. In manipulating XML documents, it is common that documents need to be verified with respect to more than one schema and the cost of revalidating the whole document is known to be high (Raghavachari and Shmueli, 2004). This paper provides an alternative method for document revalidation that is much more efficient. We obtain this by identifying delta changes between two versions of the

schema so that only the document portions affected by those changes can be revalidated. For example, assume that XML Schema S_1 at time t_1 evolves to S_2 at time t_2. Let D be a set of XML documents associated with the schema S_1 where $d_i \in D$. Assume that S_1 evolves to S_2 due to errors in the initial design or the addition of new constructs. Consequently, $d_i \in D$ may no longer be valid against S_2. Therefore, it is important to detect changes between S_1 and S_2 automatically so that delta changes, i.e. $delta(S_1, S_2)$, can be used to facilitate the conversion between d_i and d_j such that d_j is valid against S_2.

- *Incremental maintenance of relational schema generated by the schema-conscious approach for storing XML data*. The usage of relational database systems to store and process XML data is a popular research area and several methods have been developed for this purpose. One of these methods is the *schema-conscious* approach where a DTD or XML Schema is used to create the relational schema (Shanmugasundaram et al., 1999). XML Schema change detection can be particularly useful in this approach where relational schema needs to be maintained incrementally due to XML Schema changes.
- *Traditional support for XML versioning*. Change detection is an integral part of version management for databases and document archives. Users and applications may want to query different versions of data and schema both retrospectively and prospectively. In this case, the delta generated by the change detection system works both forward and backward, allowing the user or the application to reverse the V_{i+1} version of the document to the V_i. This technique was first introduced by Marian et al. (2001) by calculating the 'completed delta' for XML documents. XML Schemas are written using different rules, so the delta used in XML document versioning is unlikely to be used in XML Schema versioning. The proposed change detection method, in this paper, creates delta records that are more meaningful with respect to XML Schema language.

In this paper, we address the problem of XML Schema change detection based on the three above-mentioned motivations. We propose a relational-based algorithm called XS-Diff for detecting the changes to XML Schemas. The relational model is utilised because previous researches, such as Leonardi et al. (2007) and Leonardi and Bhowmick (2007), have proven that DOM-based change detection methods (e.g. Cobena et al., 2002b; Wang et al., 2003) may often fail to find semantically correct and optimal changes. Moreover, they suffer from scalability problems since they are not able to handle large XML documents (due to lack of memory).

In our work, first, XS-Diff parses and stores the resulting XSD trees of two schema versions, say S_1 and S_2, in the relational tables. Then, by issuing SQL queries, it matches the identical nodes and stores them in temporary tables. Finally, it uses the information stored on those tables in order to detect changes and store them in delta tables.

The work in this paper extends our previous effort (Baqasah et al., 2013) on XML Schema change detection in the following ways:

- Two schema components, namely *group* and *attributeGroup*, are included in the change detection model as secondary components. This inclusion will improve the proposed algorithm by supporting components that are widely used. Thus, we redefine XML Schema tree based on the new enhancement (in Section 3).

- A detailed explanation of our delta model (XS-Rel delta) is given. This can be used to convert one version of the schema into another.
- A formal and generic definition for each phase of the proposed algorithm is provided. That is, we aggregate similar operations on each phase (e.g. moving of elements and attributes) in a more generic style.
- We conduct additional experiments by including a performance test. The performance in terms of execution time can be measured in different settings, such as varying the percentage of changes, switching between different design patterns and varying the number of nodes (scalability).

The organisation of this paper is as follows. Section 2 provides a literature review of related researches in this field. Section 3 introduces XML Schema components used along this paper and defines the XML Schema model. We then define the relational model (XS-Rel) used for the change detection process in Section 4. In Section 5, we describe our change detection algorithm (XS-Diff) and introduce XML Schema-specific changes. Section 6 conducts some experimental results to measure the correctness, performance and result quality of the proposed algorithm and, finally, Section 7 concludes the paper with remarks on the contribution of the paper and possible improvements on this topic in the future.

2 Related work

This section describes previous researches in the field that are closely related to our paper. More comprehensive descriptions of the topic on comparing different XML change detection techniques can be found in the literature (Cobena et al., 2002a; Peters, 2005).

2.1 Change detection in hierarchically structured data

As an early work, Chawathe et al. (1996) designed a differencing system called *LaDiff* for hierarchically structured information. It takes two versions of a LaTeX document as input and produces a marked-up version of the document with changes. LaDiff considers two sub-problems: a 'good matching' and a 'minimum cost edit script computing'. The four primary edit operations used to define the minimum cost edit script are *node delete*, *node insert*, *node update* and *subtree move*. Change detection is also studied in the context of nested-object documents in *MH-Diff* algorithm (Chawathe and Garcia-Molina, 1997). MH-Diff produces operations that allow it to describe changes semantically between two trees in a more meaningful way. In addition to the traditional insert, delete and update operations, MH-Diff also supports *move*, *copy* and *glue* (inverse of copy) operations. The introduction of new operations results in an even higher quality edit of scripts, especially when the copied or moved sub-trees are large. The main weakness of the previous approaches is that they cannot understand components of XML Schema. Furthermore, as the authors note, MH-Diff can detect changes between unordered trees, while XML Schema contains both ordered and unordered segments. In ordered trees, both ancestor (parent–child) relationships and left-to-right ordering among siblings are considered important, while in unordered trees, only ancestor relationships are considered significant.

2.2 Change detection in XML documents

In another group of studies, XML documents are processed. *XMLTreeDiff* (developed by IBM) finds changes between two XML documents using a set of java beans to compute the differences of Document Object Model (DOM) structures (IBM, 1998). This method mainly consists of two phases: in the first phase, hash values for the nodes of two document versions are computed using DOMHash (Maruyama et al., 2000); in the second phase, it computes the difference between the two simplified trees using Zhang and Shasha's (1989) algorithm. Despite its ease of use, Wang et al. (2003) reported that its result may not be optimal due to the conflicts with the cost model proposed by Zhang and Shasha's algorithm. Cobena et al. (2002b) proposes a main-memory algorithm called *XyDiff* for detecting changes between versions of XML documents. The algorithm detects changes in a bottom-up fashion of ordered documents. XyDiff also takes advantage of XML specifications, e.g. handling attributes and treating them differently from text and element nodes. In addition to the basic insert, delete and update operations, the algorithm supports a *move* operation. In contrast to XyDiff, the *X-Diff* algorithm proposed by Wang et al. (2003) is able to handle unordered XML documents in a top-down fashion, but it does not support the move operation. It achieves the optimal changes by integrating key XML structure characteristics with standard tree-to-tree correction methods.

Although XyDiff and X-Diff are able to detect differences between versions of XML documents, Leonardi et al. (2007) argue that they may often fail to detect semantically correct and optimal changes. (The semantic correctness is further explained with examples in Section 6.) Furthermore, they suffer from scalability problems, as they cannot handle large XML documents (more than 5000 nodes) due to the use of DOM to represent the compared trees.

To address the scalability problems of XML document change detection described above, a number of algorithms have been proposed using relational database systems. The idea is to exploit the relational environment to store XML data, and then apply a set of SQL queries to find changes between the stored documents. A set of modules called XANADUE is proposed by Leonardi and Bhowmick (2007) to detect changes to unordered and ordered XML documents. In Al-Ekram et al. (2005), diffX algorithm is also proposed for detecting changes between XML document versions. This method aims to minimise the size of the resulting edit script and optimise the runtime of mapping the nodes. To achieve these targets, it performs the 'isolated tree fragment mapping' technique to identify the largest matching fragments between the two trees. Therefore, it handles ordered elements and unordered attributes, and supports primitive operations, such as insert, delete and move in addition to replace, which is introduced to reduce the size of the edit script (i.e. insert followed by the delete operation). DeltaXML is a commercial tool that provides a comparison for XML documents and also represents changes in XML. The principal idea behind the tool can be found in Robin (2001). Its technique is based on 'longest common subsequence computations'. Operations supported by DeltaXML are insert, delete and update, while move is not supported. As a result, in case the XML document node, for instance element or attribute, moved from one part of the tree to another, the tool will detect that change as a deletion from the first tree followed by an insertion to the second.

2.3 Change detection in XML schemas

XML schemas (i.e. XML Schema or DTD instances) have not been extensively studied in the area of change detection. As far as we know, the *DTD-Diff* algorithm published by

Leonardi et al. (2007) is the only work in this context and is used to find changes between versions of a DTD. The algorithm takes two DTD files representing the first and the second versions as input and returns a list of changes containing the differences between the two versions. It defines types of changes for the following DTD components: *Element Type Declaration (ETD)*, *Attribute Declaration (AD)* and *Entity Declaration (ED)*. Leonardi et al. (2007) claim that converting DTD to XML Schema and detecting the changes using one of the existing XML document change detection tools are not a practical option because they are expensive and may yield semantically incorrect or non-optimal changes. As the XML Schema format is different from that of a DTD, the proposed changes between the old and the new versions are different from the ones that occur in DTD versions. As a matter of fact, XML Schema has a richer variety of changes compared to DTD. For example, in XML Schema, a `complexType` is used to define the content type of the element. The complex type definition can migrate (by changing its scope) from the top level of the schema tree to be locally defined under its element.

2.4 XML schema evolution

Apart from the change detection tools discussed above, XML Schema change detection cannot be handled without the investigation of XML Schema evolution. The concept of evolution is important because our proposed change operations are based on the evolution primitives introduced by the evolution techniques. For example, Guerrini et al. (2005a) studied the impact of XML Schema updates on the validity of related documents. The authors first devised a set of *atomic evolution primitives* to be applied to the basic components of the schema. Then, they presented some *high-level evolution primitives* that are a composite of the atomic primitives. The evolution primitives suggested by this work have been classified into three main categories: *insertion*, *modification* and *deletion* of the main XML Schema components (`element`, `simpleType` and `complexType`). We enhance the proposed classification by considering changes for other XML Schema components, such as `attribute`, model group (i.e. `sequence`, `choice` and `all`), as we will see in the next section.

3 XML Schema model

XML Schemas and their components can be better described by using XML Schema Object Model (XSOM). XSOM is a Java library (https://xsom.java.net/) used to parse XSDs and retrieve information inside them. The library is a straightforward implementation of XML Schema components defined by W3C (2004b). In this section, we begin by showing the most important components and define XML Schema tree model based on XSOM.

3.1 XML schema components

According to W3C (2004b), there are 13 kinds of components divided into *primary*, *secondary* and *helper* components. The primary components include attributes (`attribute`), elements (`element`), simple types (`simpleType`) and complex types (`complexType`). Components in this category are most likely to be used by the common XML Schemas. The secondary components are used but less than the primary ones. Components that fall in this category are attribute group definitions

(`attributeGroup`), model group definitions (`group`), identity-constraint definitions (i.e. `ID/IDREF` and `key/keyref`) and notation declarations (`notation`). The third category refers to the helper components, which include annotations (`annotation`), model groups (`sequence`, `choice` and `all`), particles, wildcards (`any` and `anyAttribute`) and attribute uses. They are called helpers because they do not stand by themselves but as child nodes of other components. Since most of the previous components are not an integral part of every schema, we only consider the dominant ones: `attribute`, `element`, `simpleType`, `complexType`, `attributeGroup`, `group` and model group as `sequence`, `choice` or `all`. We also consider facets as important nodes that are commonly used to restrict simple types and the simple content of the complex type. Examples of facets are `minInclusive`, `maxInclusive`, `length`, `pattern` and `enumeration`. Note that the terms 'component' and 'node' are used interchangeably throughout this paper and both relates to one of the dominant components listed above.

3.2 XML schema tree representation

In this section, we formally define the XML Schema tree in order to use it during the parsing and change detection phases.

Definition 1 *(XSD Tree): An XSD is a tree T = (AD, ED, ST, CT, AG, GD, MG, F) that has different types of nodes, where:*

1. *AD is a set of attribute declaration nodes of the form* $ad = (ad_n, ad_t, ad_f, ad_d, ad_u)$, *where* ad_n *is the name of the node,* ad_t *is the data type of the node (i.e. built-in or user-derived simple type),* ad_f, ad_d *and* ad_u *are values of fixed, default and use attributes of the attribute declaration node, respectively;*
2. *ED is a set of element declaration nodes of the form* $ed = (ed_n, ed_t, ed_{mno}, ed_{mxo}, ed_o)$, *where* ed_n *is the name of the node,* ed_t *is the data type of the node (elements may have simple or complex types),* ed_{mno} *and* ed_{mxo} *are values of the* `minOccurs` *and* `maxOccurs` *attributes representing the cardinality of the element declaration node, respectively, and* ed_o *is the order of the element node among its siblings;*
3. *ST is a set of simple type definition nodes of the form* $st = (st_n, st_d, st_b)$, *where* st_n *is the name (if defined globally) of the simple type node,* $st_d = \{restriction, list, union\}$ *is the derivation type of the node and* st_b *is the value of either the* `base`, `itemType` *or* `memberTypes` *attribute;*
4. *CT is a set of complex type definition nodes of the form* $ct = (ct_n, ct_c, ct_d, ct_b)$, *where* ct_n *is the name (if defined globally) of the complex type node,* $ct_c = \{complexContent, simpleContent, \theta\}$, $ct_d = \{restriction, extension, \theta\}$ *and* ct_b *are the content type, the derivation type and the base type of the complex type node, respectively;*
5. *AG is a set of attribute group definition nodes of the form* $ag = (ag_n)$, *where* ag_n *is the name of the node;*
6. *GD is a set of model group definition nodes of the form* $gd = (gd_n, gd_{mno}, gd_{mxo}, gd_o)$, *where* gd_n *is the name of the node,* gd_{mno} *and* gd_{mxo} *represent the* `minOccurs` *and* `maxOccurs` *attributes of the model group definition, respectively, and* gd_o *is the order of the referenced node;*

7 *MG is a set of model group nodes of the form mg =* $(mg_c, mg_{mno}, mg_{mxo}, mg_o)$*, where* mg_c *= {sequence,choice,all} is the compositor of the model group defined under a particular complex type or group definition node,* mg_{mno} *and* mg_{mxo}*, respectively, are the* `minOccurs` *and* `maxOccurs` *attributes of the model group node and* mg_o *is the order of the model group node among its siblings;*

8 *F is a set of restriction facet nodes of the form f =* (f_n, f_v)*, where* f_n *= {minExclusive, minInclusive, maxExclusive, maxInclusive, length, minLength, maxLength, totalDigits, fractionDigits, enumeration, pattern, whitespace, explicitTimezone} is the name of the facet and* f_v *is the value assigned to the facet.*

Based on Definition 1, the two input XSD versions S_1 and S_2 in Table 1 are translated into their respective tree representations T_1 and T_2 in Figure 1. As a preparatory step for the change detection process, we match identical nodes. To identify nodes in the tree, there are several techniques such as XPath expressions and XID signatures. W3C (1999) recommend XPath as a language for addressing XML document parts (i.e. used to navigate through elements and attributes in an XML document). To select nodes and node-sets, XPath uses 'path expressions'. In our XML Schema context, we cannot use path expressions because they return a set of nodes. Thus, identifying one node through different schema versions is not possible. XIDs is an alternative solution introduced by Marian et al. (2001) to identify XML document nodes in the context of Xyleme project. The technique solves the previous problem of XPath for identifying a single node through a set of XML versions. However, it is still not a perfect method since the system has to maintain the assignment of XIDs to nodes through a function called 'XID-map'.

To identify node paths that best fit our requirements, we use a 'node signature', which has the same spirit of path expressions in XPath language. The unique path in our XSD tree model is defined as follows:

Definition 2 *(XSD unique path): Given an XML Schema with a tree representation T, the unique path for each node in the tree is a concatenation of node tags starting from the root node (schema) that ends with one of the following node variations:*

- *If the node type is one of the components AD, ED, ST, CT, AG or GD (listed in Definition 1) and the attribute name of the component is known (e.g.* $ad_n \neq \theta$*, where* $ad_n \in \# AD$*), the path ends by the node type followed by the value of the attribute name of the node between two brackets ([]), otherwise the path ends by the node type only.*
- *If the node type is MG and is a child of a* `sequence` *model group node, then the child order is embedded instead of the attribute name.*
- *If the node type is F, then the path ends by the facet name followed by facet value between two brackets ([]).*

Examples (from Figure 1) illustrating the idea of identifying schema nodes based on Definition 2 are the following:

`#/schema#/attribute[A1],`

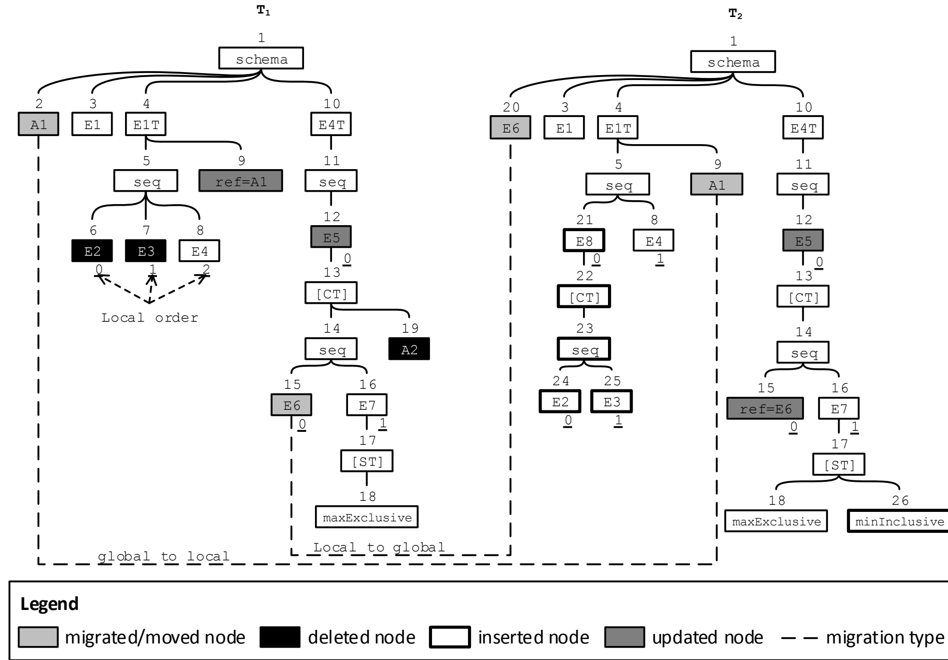
`#/schema#/CT[E1T]#/sequence[]#/element[E2]` and

`#/schema#/CT[E4T]#/sequence[]#/element[E5]#/CT[].`

In these examples, the first line represents the unique path for the global attribute declaration named *A1*. The second line represents *E2* element declaration, whereas the final line represents the anonymous complex type under *E5* element declaration.

To easily maintain node paths during the example in this paper, we assign a *unique ID* for each node in the compared trees (appears above each node in Figure 1).

Figure 1 XML Schema trees T_1 and T_2 of the schema versions in Table 1 (T_1 represents S_1 and T_2 represents S_2)



3.3 *Matching schema versions*

Matching identical nodes between trees is an essential part of any change detection procedure. In the XML Schema context, the schema component nodes, which fall in the *same type* and with the *same unique path* between the two compared tree versions, are considered identical. For example, node `3` in both T_1 and T_2 in Figure 1, representing element declaration node `E1`, shares the same unique path: `#/schema#/element[E1]`. Similarly, node `12` in both T_1 and T_2 represents element declaration node `E5` and shares the same unique path: `#/schema#/CT[E4T]#/sequence[]#/element[E5]`. Observe that the uniqueness of the node path between the two versions does not necessarily require its attributes to be the same. For example, although the path of node `12` is unique between the two trees T_1 and T_2, the node is considered *updated* because the value of its attribute `maxOccurs` changes from `5` to `10` as seen in the corresponding S_1 and S_2 in line `14` Table 1(a) and line `20` Table 1(b), respectively.

Table 1 Two versions of XML Schema S_1 and S_2 (changes are highlighted)

(a) S_1	*(b) S_2*
01 <xs:schema ...>	01 <xs:schema ...>
02 <xs:attribute name="A1" type="xs:string"/>	02 <xs:element name="E6" type="xs:string"/>
03 <xs:element name="E1" type="E1T"/>	03 <xs:element name="E1" type="E1T"/>
04 <xs:complexType name="E1T">	04 <xs:complexType name="E1T">
05 <xs:sequence>	05 <xs:sequence>
06 <xs:element name="E2" type="xs:string"/>	06 <xs:element name="E8">
07 <xs:element name="E3" type="xs:string"/>	07 <xs:complexType>
08 <xs:element name="E4" type="E4T"/>	08 <xs:sequence>
09 </xs:sequence>	09 <xs:element name="E2" type="xs:string"/>
10 <xs:attribute ref="A1" use="required"/>	10 <xs:element name="E3" type="xs:string"/>
11 </xs:complexType>	11 </xs:sequence>
12 <xs:complexType name="E4T">	12 </xs:complexType>
13 <xs:sequence>	13 </xs:element>
14 <xs:element name="E5" maxOccurs="5">	14 <xs:element name="E4" type="E4T"/>
15 <xs:complexType>	15 </xs:sequence>
16 <xs:sequence>	16 <xs:attribute name="A1" type="xs:string"/>
17 <xs:element name="E6" type="xs:string"/>	17 </xs:complexType>
18 <xs:element name="E7">	18 <xs:complexType name="E4T">
19 <xs:simpleType>	19 <xs:sequence>
20 <xs:restriction base="xs:integer">	20 <xs:element name="E5" maxOccurs="10">
21 <xs:maxExclusive value="100"/>	21 <xs:complexType>
22 </xs:restriction>	22 <xs:sequence>
23 </xs:simpleType>	23 <xs:element ref="E6"/>
24 </xs:element>	24 <xs:element name="E7">
25 </xs:sequence>	25 <xs:simpleType>
26 <xs:attribute name="A2" type="xs:string"/>	26 <xs:restriction base="xs:integer">
27 </xs:complexType>	27 <xs:maxExclusive value="100"/>
28 </xs:element>	28 <xs:minInclusive value="1"/>
29 </xs:sequence>	29 </xs:restriction>
30 </xs:complexType>	30 </xs:simpleType>
31 </xs:schema>	31 </xs:element>
	32 </xs:sequence>
	33 </xs:complexType>
	34 </xs:element>
	35 </xs:sequence>
	36 </xs:complexType>
	37 </xs:schema>

4 XML schema storage in relational model

The design of the relational database schemas used by the XS-Diff algorithm plays a key role in the change detection process. In this section, we build our relational model XS-Rel based on a scalable technique *XRel* (Yoshikawa et al., 2001) to store and retrieve XML documents using relational databases. We show how XRel model can be used to accommodate XML Schema components considered in this work.

XML documents can be stored in relational databases using two approaches: *structure-mapping* and *model-mapping*. In the structure-mapping approach, a database schema is created for each XML document structure XSD or DTD (e.g. a relational table is created for each element type in the XML document). Therefore, it is more suitable for storing a large number of documents that conform to a limited number of XML Schemas or DTDs. On the other hand, in the model-mapping approach, there is no need to have an XSD or DTD in order to design the relational schemas. Instead, we create database schemas based on the constructs of the XML document model. We select the model-mapping approach because there is no schema attached to XSD versions.

As stated earlier, we adopt XRel technique to store XML paths. Unfortunately, this technique, which uses *path expressions* as a unit of decomposition of XML trees, may not be suitable for storing XML Schema trees. For example, given a schema S_1 in Table 1(a), when we apply the XRel approach, the path from the root node `schema` to the first element node tagged `attribute` (from XML document perspective) can be denoted as `#/schema#/attribute`. Similarly, the path from the root node schema to the first attribute node tagged name is `#/schema#/attribute#/@name`. As can be seen here, the previous paths are more appropriate for describing traditional XML document nodes: *elements*, *attributes* and *text nodes*. In the XML Schema language, there exists a set of rules (syntax) to write the XML Schema component definitions and declarations (e.g. only `element`, `attribute`, `simpleType`, `complexType`, `group` and `attributeGroup` nodes may exist at a top level of the schema tree). Based on the previous observation, we alter the XRel storage technique to our relational model called XS-Rel (for **X**ML **S**chema to **Rel**ational Storage) to fulfil our requirements of storing and later querying XML Schema components. For example, unlike XRel where tables are created on the basis of the type of XML document nodes (i.e. element, attribute or text node), XS-Rel creates tables based on the most popular schema components emphasised in Definition 1.

4.1 XS-Rel relational schema

To detect changes in XML Schema, we redesign XS-Rel schema to include *basic tables*, *temporary tables* and *delta tables*. Basic tables represent the eight XML Schema components introduced in the previous section. Each table in this category stores all information about the two parsed versions of the schema. As we intend to maintain XML Schema versions and their respective repositories (as a future extension of this work), `version` and `repository` tables are also created. This group also includes `path` table that stores all the unique paths from the root node `schema` to every node in the tree. Figure 2(a) shows an excerpt of XML Schema running examples stored in XS-Rel basic tables.

Temporary tables are used to store the matched components between the two schema versions and will be used at later stages to speed up the change detection process. As the basic tables group, eight temporary tables correspond to the schema components discussed earlier. For example, the element components have a temporary table named `tmp_`

`element` with the properties of `pathid`, `name`, `type1`, `type2`, `minO1`, `minO2`, `maxO1`, `maxO2`, `locO1`, `locO2`, `isRef1`, `isRef2`, `isGlob1`, `isGlob2`. We explain SQL queries used with the temporary tables in the next section (XS-Diff algorithm).

Delta tables are a set of XML Schema change operations stored as records in the relational tables. The purpose of delta tables is obvious; they are mainly designed to store XML Schema changes (XS-Rel delta) in the form of a 'completed delta' (Marian et al., 2001). The completed delta has the advantage of reversing change operations so that we can use the same delta record to retrieve either the old schema version or the new one. For instance, delta records containing deleted and inserted elements in Figure 1 are stored in `del_ins_element` table as shown in Figure 2(b). Due to space limitation in this paper, we have attached the relational schema for all XS-Rel tables and queries to a public domain in the 'XS-Diff google docs' page (https://drive.google.com/file/d/0By77KsJuXVdcRjg1N0p6WDVNVlU/edit?usp=sharing). We now define the relational delta (XS-Rel delta) that is used throughout this paper.

Figure 2 XS-Rel basic and delta tables populated by XML Schema versions S1 and S2 in Table 1. (a) Basic tables including element and attribute tables as examples for schema components; (b) delta tables including insertion and deletion of element and attribute as examples

`repository`

rid	rname	active	initVer	currV	noVs
1	S	1	S1.xsd	S2.xsd	2

`version`

vid	vn	vpn	vdoc	date	rid
1	1	0	S1.xsd	2-4	1
2	2	1	S2.xsd	2-5	1

`path`

pathid	pathexp	rid
2	#/schema#/attribute[A1]	1
3	#/schema#/element[E1]	1
…	…	…
7	#/schema#/CT[E1T]#/sequence#/element[E3]	1
8	#/schema#/CT[E1T]#/sequence#/element[E4]	1
9	#/schema#/CT[E1T]#/attribute[A1]	1
…	…	…
19	#/schema#/CT[E4T]#/sequence#/element[E5]#/CT[]#/attribute[A2]	1
21	#/schema#/CT[E1T]#/sequence#/element[E8]	1
22	#/schema#/CT[E1T]#/sequence#/element[E8]#/CT[]	1
23	#/schema#/CT[E1T]#/sequence#/element[E8]#/CT[]#/sequence	1

`attribute`

vid	pathid	pPathid	name	type	fixedV	defaultV	useV	isRef	isGlob
1	2	1	A1	string	Null	Null	Null	0	1
1	9	4	A1	Null	Null	Null	required	1	0
1	19	13	A2	string	Null	Null	Null	0	0
…	…	…	…	…	…	…	…	…	…
2	9	4	A1	string	Null	Null	Null	0	0

`element`

vid	pathid	pPathid	name	type	minO	maxO	locO	isRef	isGlob
1	3	1	E1	E1T	Null	Null	0	0	1
1	7	5	E3	string	Null	Null	1	0	0
1	12	11	E5	Null	Null	5	0	0	0
…	…	…	…	…	…	…	…	…	…
2	3	1	E1	E1T	Null	Null	0	0	1
2	12	11	E5	Null	Null	10	0	0	0
2	20	1	E6	string	Null	Null	0	1	1
2	21	5	E8	Null	Null	Null	0	0	0

(a)

`del_ins_attribute`

id	pathid	path	cType	name	type	fixed	default	use	isRef	isGlob	vid
1	19	#/schema…#/attribute[A2]	del	A2	string	null	null	null	0	0	1

`del_ins_element`

id	pathid	path	cType	name	type	minO	maxO	locO	isRef	isGlob	vid
1	7	#/schema…#/element[E3]	del	E3	string	Null	Null	1	0	0	1
2	21	#/schema…#/element[E8]	ins	E8	Null	Null	Null	0	0	0	2

(b)

Definition 3 (XS-RelDelta): *Given any two successive versions S_1 and S_2 of an XML Schema S, where $S_1 \neq S_2$, XS-RelDelta is a set of relational tables that record the changes of the schema from one version to another. XS-RelDelta consists of a set of operations O {migrate, move, delete, insert, update}, which, upon execution on the schema version S_1, will produce the schema version S_2.*

Note that the terms 'delta' and 'XS-RelDelta' share the same concept defined as per Definition 3 and are used interchangeably throughout this paper.

5 XS-Diff algorithm

In the previous section, we described and defined the relational model (XS-Rel) used to store XML Schema changes. In this section, we present *XS-Diff*, the algorithm to detect changes between the XSD versions and store the changes in the relational delta XS-RelDelta. For simplicity and better understanding, the algorithm is split into four different phases: (i) *matching schema components*, (ii) *detecting migrated and moved components*, (iii) *detecting deleted and inserted components* and (iv) *detecting updated components*. Each phase of the proposed algorithm is discussed in detail in this section using examples where appropriate. The pseudo-code of the XS-Diff algorithm is presented in Table 2.

Table 2 XS-Diff algorithm

```
01  Input: XSDs s1 and s2
02  Output: DELTA tables (mig_X, mov_X, del_ins_X, upd_X), where
X is
            a node type according to Definition 1
03  /* Phase 1: find matching components */
04  For each node n in s1 and s2 parsed schemas, do
05      If ((class(n) == class(n)) and (path(n) == path(n))),
then
06          tmp_n = find_unique_n_path();
07  /* Phase 2: find migrated and moved components */
08  For each node n in s1 and s2, do {
09      If (isElement(n)), then {
10          mig_ed = find_gtl_ed() + find_ltg_ed(); //gtl: global
to local
11      } Else if (isAttribute(n)), then {          //ltg: local
to global
12          mig_ad = find_gtl_ad() + find_ltg_ad();
13      } Else if (isComplexType(n)), then {
14          mig_ct = find_gtl_ct() + find_ltg_ct();
15      } Else if (isSimpletype(n)), then {
16          mig_st = find_gtl_st() + find_ltg_st();
17      } }
```

Table 2 XS-Diff algorithm (continued)

```
18  For each node n in s1 and s2, do {
19      If (isComplexType(n)), then mov_ct = find_moved_ct();
20      Else if (isModelgroup(n)), then mov_mg = find_moved_mg();
21      Else if (isElement(n)), then mov_ed = find_moved_ed();
22      Else if (isAttribute(n)), then mov_ad = find_moved_ad();
23      Else if (isSimpletype(n)), then mov_st = find_moved_st();
24      Else if (isFacet(n)), then mov_f = find_moved_f();
25      Else if (isAttributeGroup(n)), then mov_ag =
find_moved_ag();
26      Else if (isGroup(n)), then mov_g = find_moved_g(); }
27  /* Phase 3: find deleted and inserted components */
28  For each node n in s1 and not in s2, do {
29      If (isElement(n)), then del_ins_ed = find_deleted_ed();
30      Else if (isAttribute(n)), then del_ins_ad =
find_deleted_ad();
31      Else if (isComplextype(n)), then del_ins_ct =
find_deleted_ct();
32      Else if (isSimpletype(n)), then del_ins_st =
find_deleted_st();
33      Else if (isModelgroup(n)), then del_ins_mg =
find_deleted_mg();
34      Else if (isFacet(n)), then del_ins_f = find_deleted_f();
35      Else if (isAttributeGroup(n)), then del_ins_ag =
find_deleted_ag();
36      Else if (isGroup(n)), then del_ins_g = find_deleted_
g(); }
37  For each node n in s2 and not in s1, do {
38      If (isElement(n)), then del_ins_ed = find_inserted_ed();
39      Else if (isAttribute(n)), then del_ins_ad =
find_inserted_ad();
40      Else if (isComplextype(n)), then del_ins_ct =
find_inserted_ct();
41      Else if (isSimpletype(n)), then del_ins_st =
find_inserted_st();
42      Else if (isModelgroup(n)), then del_ins_mg =
find_inserted_mg();
43      Else if (isFacet(n)), then del_ins_f = find_inserted_f();
44      Else if (isAttributeGroup(n)), then del_ins_ag =
find_inserted_ag();
45      Else if (isGroup(n)), then del_ins_g = find_inserted_
g(); }
```

Table 2 XS-Diff algorithm (continued)

```
46  /* Phase 4: find updated components */
47  For each node n in s1 and s2, do {
48      If (n ∈ tmp_n table), then {                // n is
populated by Phase 1
49          If (isElement(n)), then upd_ed = find_updated_ed();
50          Else if (isAttribute(n)), then upd_ad =
find_updated_ad();
51          Else if (isComplextype(n)), then upd_ct =
find_updated_ct();
52          Else if (isSimpletype(n)), then upd_st =
find_updated_st();
53          Else if (isModelgroup(n)), then upd_mg =
find_updated_mg();
54          Else if (isGroup(n)), then upd_g = find_updated_g();
55      } }
```

For the reader's convenience, a number of SQL queries used during the XS-Diff algorithm are mentioned as examples to show how the schema versions and delta changes are stored in XS-Rel relational tables.

5.1 Phase 1: matching schema components

In the first phase, we parse two XML Schema trees and *match* their respective components one by one. The matched components (pairs with the same unique path) are then stored in a corresponding temporary table introduced in Section 4.1. In addition, the system generates an *id* for the unique (shared) path and stores it along with its unique path (as per Definition 2) in the `path` table. This step is necessary to improve the efficiency and speed of the algorithm. The temporary tables are also used later by SQL queries to find moved, inserted, deleted and updated components.

In what follows, we formally define the first phase of the change detection algorithm.

Definition 4 (find matching components): *Let N_1 and N_2 be two identical sets of nodes in the first and second trees, T_1 and T_2, of XML Schema, respectively. $n_i \in N_1$ and $n_j \in N_2$ are matching components iff n_i and n_j have the same unique path [i.e. path(n_i) = path(n_j)].*

Example: The unique path for the element `E5` (node `12` in Figure 1), shared between the two trees T_1 and T_2, is `#/schema#/CT[E4T]#/sequence#/element[E5]`.

Hence, we store the resulting match in a temporary table corresponding to the type of the matched components (in this example, `tmp_element` is the temporary table for the matched element nodes). Simultaneously, we store the unique path and the path id for the matched nodes in the `path` table. For parsing and matching the components, the algorithm runs a loop (from line `4` to line `6` of the algorithm in Table 2) that inspects the component type and, based on the type, invokes an appropriate SQL query. For instance, the query for matching element components is

```
INSERT INTO tmp_element
SELECT e1.pathid, e1.name, e1.type, e2.type, e1.minO,
e2.minO, e1.maxO, e2.maxO, e1.locO, e2.locO, e1.isRef,
e2.isRef, e1.isGlob, e2.isGlob
FROM element AS e1, element AS e2
WHERE e1.vid = <vi> AND e2.vid = <vj> AND e1.pathid =
e2.pathid
```

where `vi` and `vj` represent the first and second versions, respectively, of the matched schema. After parsing the two schemas, storing components information and their matching paths, we proceed to the detection of migration and move phase.

5.2 Phase 2: detection of migrated and moved components

In this phase, migrated components are classified into (i) declaration components including `element` and `attribute` and (ii) type components including `complexType` and `simpleType`. For each category, migration occurs in two directions: *global-to-local* and *local-to-global*. Then, all schema components (under study) are subject to move from their old positions in the first tree to new positions in the second, thus requiring move operations. As a result, this phase consists of 12 operations maintained in lines `8–26` of the algorithm in Table 2.

5.2.1 Migration changes

In XML Schema, if a component (e.g. `element`) moves from the top level of the schema to be locally defined under a content model (or vice versa), then it changes its scope. A *migrate* change does not exist between traditional XML document versions. Therefore, we devise it as a new type of change.

Migration changes can be captured in element and attribute declarations (lines `9–12` in the algorithm). For the global-to-local change, the element/attribute node declared at a top level of the schema tree is deleted and all its information (attributes such as `name` and `type`) will be mapped to the relevant local element/attribute node. On the contrary, the local (reference) of the component is not deleted in the local-to-global change of the element/attribute node. Instead, its `name` attribute will be replaced with the `ref` attribute, which refers to the name of the globally declared component. Formal definitions of declaration components are given in Definitions 5 and 6.

Definition 5 (detect global-to-local migration of declaration component): *Let DC_1 and DC_2 be two sets of declaration component nodes (`element` or `attribute`) from the first and second trees T_1 and T_2, respectively. Node $dc \in DC_1$ is migrated from a global to a local definition iff dc has the same `name` and `ref` attributes in both T_1 and T_2, and it is global in T_1 and local in T_2.*

Example (attribute migration): An attribute A1 [node 2 in Figure 1(T_1)] is migrated from a global definition to a local definition at node `9` in T_2. To detect global-to-local migration of the previous attribute node, we execute the following SQL query.

```
INSERT INTO mig_attribute
SELECT a1.pathid, a2.pathid, p1.pathexp, p2.pathexp,
a1.name, a1.type, a2.type, a1.fixedV, a2.fixedV,
a1.defaultV, a2.defaultV, a1.useV, a2.useV, a1.isGlob,
a2.isGlob, a2.vid
FROM attribute AS a1, attribute AS a2, path AS p1, path
AS p2
WHERE a1.vid = <vi> AND a2.vid = <vj> AND a1.name = a2.name
AND a1.isGlob = 1 AND a2.isGlob = 0 AND a1.isRef = a2.isRef
AND p1.pathexp IN (SELECT pathexp FROM path WHERE pathid =
a1.pathid)
AND p2.pathexp IN (SELECT pathexp FROM path WHERE pathid =
a2.pathid)
```

Definition 6 (detect local-to-global migration of declaration component): *Let DC_1 and DC_2 be two sets of declaration component nodes (`element` or `attribute`) from the first and second trees T_1 and T_2, respectively. Node $dc \in DC_1$ is migrated from a local to a global definition iff dc has the same `name` and `ref` attributes in both T_1 and T_2, and it is local in T_1 and global in T_2.*

Example (element migration): An element `E6` [node `15` in Figure 1(T_1)] is migrated from a local definition (under a complex type at node `13`) to a global definition at node `20` in T_2.

Migration changes can also be identified for simple and complex type nodes (lines `13`–`16` in the algorithm). For instance, to detect global-to-local type of migration, we find a type that is defined globally in the first tree and locally (as anonymous type) under element or attribute in the second tree. Generally, a formal definition of global-to-local type is as follows:

Definition 7 (detect global-to-local migration of type component): *Let TC_1 and TC_2 be two sets of type component nodes (simpleType or complexType) from the first and second trees T_1 and T_2, respectively. Let ED be a set of element nodes from T_2, AD be a set of attribute nodes from T_2, ED_M be a set of temporary matched elements and AD_M be a set of temporary matched attributes. Node $tc \in TC_1$ is migrated from a global definition to local as a child of $ed \in ED$ iff:*

- *tc is global in T_1 and local in T_2;*
- *its name in T_1 is equal to the type definition of at least one element in ED [i.e. name(tc) = type(ed)];*
- *its parent node in T_2 exists in the set of matched element nodes ED_M [i.e. pPath(tc) = path(ed_m) where $ed_m \in ED_M$].*

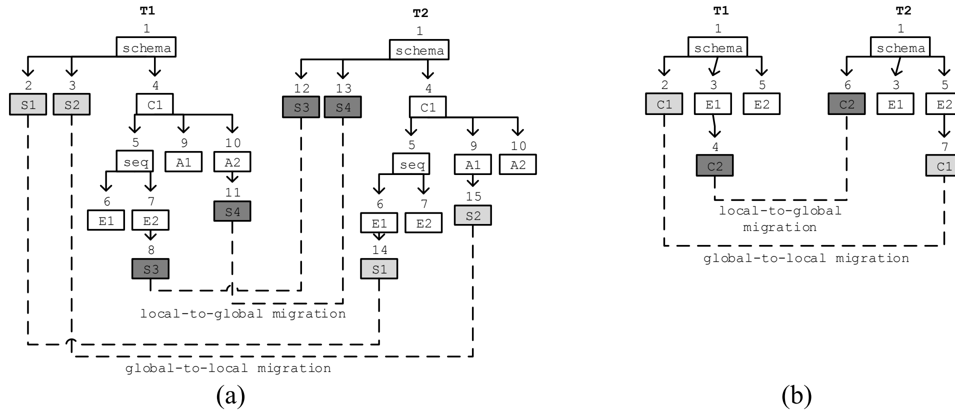
Similarly, node $tc \in TC_1$ is migrated from a global to a local definition as a child of $ad \in AD$ *iff:*

- *tc* is global in T_1 and local in T_2;
- its name in T_1 is equal to the type definition of at least one attribute in T_2 (i.e. *name*(*tc*) = *type*(*ad*)];

- its parent node in T_2 exists in the set of matched element nodes AD_M [i.e. *pPath*(*tc*) = *path*(ad_m), where $ad_m \in AD_M$].

Example (simpleType migration): Consider T_1 and T_2 schema tree versions as depicted in Figure 3(a). Nodes `S1` and `S2` are simple types migrated from the global definitions in T_1 to be locally defined under `E1` element and `A1` attribute in T_2, respectively.

Figure 3 Examples of (a) simple type and (b) complex type migrations



Detecting the opposite migration changes (i.e. local-to-global migration) to types components is also possible. This time, the detection is done by reversing the previous definition as follows.

Definition 8 (detect local-to-global migration of type component): *Let TC_1 and TC_2 be two sets of type component nodes (simpleType or complexType) from the first and second trees T_1 and T_2, respectively. Let ED be a set of element nodes from T_1, AD be a set of attribute nodes from T_1, ED_M be a set of temporary matched elements and AD_M be a set of temporary matched attributes. Node $tc \in TC_1$ is migrated from a local definition as a child of $ed \in ED$ to a global definition iff:*

- *tc is local in T_1 and global in T_2;*
- *its name in T_2 is equal to the type definition of at least one element in ED [i.e. name(tc) = type(ed)];*
- *its parent node in T_1 exists in the set of matched element nodes ED_M [i.e. pPath(tc) = path(ed_m), where $ed_m \in ED_M$].*

Similarly, node $tc \in TC_1$ is migrated from a local definition as a child of $ad \in AD$ to global definition *iff:*

- *tc* is local in T_1 and global in T_2;
- its name in T_2 is equal to the type definition of at least one attribute in *AD* [i.e. *name*(*tc*) = *type*(*ad*)];
- its parent node in T_1 exists in the set of matched attribute nodes AD_M [i.e. *pPath*(*tc*) = *path*(ad_m), where $ad_m \in AD_M$].

Example (complexType migration): Consider node `C2` depicted in Figure 3(b). This node is a complex type migrated from the local definition (as anonymous type under `E1` element) in T_1 to become a global definition in T_2 (node `6`).

5.2.2 Move changes

Move is a critical operation in the majority of XML change detection methods. In our context, it can be classified into: (i) *move among siblings under the same parent node* (also named order change) and (ii) *move to different parent nodes*. Here, we study the second type and postpone the first one to the last phase of the algorithm. Note that the move to different parent nodes does not include the move from/to the top level of the schema tree because these changes are considered as migration changes discussed earlier.

Any component of XML Schema can be moved either by itself or as a result of its parents moving. Table 3 shows the affected children nodes from a component migration/move. This observation is based on XML Schema syntax. As seen in this table, all components, except facet, are possibly internal nodes, so their child nodes are affected by the component change.

Table 3 Affected children components from a component migration/move

		Direct affected child components									
		`element`	`attribute`	`Simple Type`	`Complex Type`	`group`	`Attribute Group`	`sequence`	`choice`	`all`	`<facet>`
Moved/migrated components	`element`			√	√						
	`attribute`			√							
	`simpleType`										√
	`complexType`		√			√	√	√	√	√	√
	`group`							√	√	√	
	`attributeGroup`		√				√				
	`sequence`	√				√		√	√		
	`choice`	√				√		√	√		
	`all`	√				√					
	`<facet>`										

The idea behind the move operation, in general, is to check that the nodes (of the same type) being tested satisfy the following conditions:

1. They are located in different positions.
2. They have the same name.
3. None of them falls in the migrated set.
4. The parent path of the old one is part of the migrated/moved operation set of possible main components, listed in Table 3.

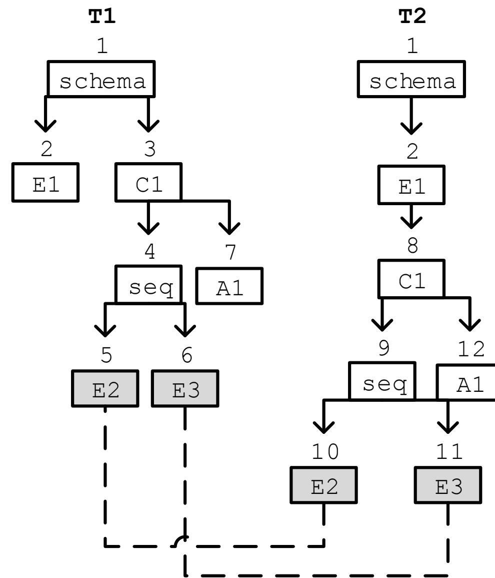
Based on the above conditions, we generically define the component move detection, then give one example for element component move with its SQL query.

Definition 9 (detect moved component): *Let C_1 and C_2 be two sets of nodes of the same component type (e.g. element or attribute) from the first and second trees T_1 and T_2, respectively. Let M be a set of migrated nodes of the same component type between T_1 and T_2, M_P be a set of possible migrated parent nodes of $c_1 \in C_1$ or $c_2 \in C_2$ (based on Table 3) and MV_P be a set of possible moved parent nodes of $c_1 \in C_1$ or $c_2 \in C_2$ (based on Table 3). Node c_1 is moved from an old position in T_1 to a new one in T_2 iff:*

- *c_1 and c_2 have different paths [i.e. $path(c_1) \neq path(c_2)$];*
- *c_1 and c_2 have the same name;*
- *c_1 and c_2 do not exist in M [i.e. $c_1 \notin M$ and $c_2 \notin M$];*
- *parent node of either c_1 or c_2 does not exist in M_P;*
- *parent node of either c_1 or c_2 does not exist in MV_P.*

Example: Figure 4 demonstrates the element move operation as an example of the component move. As seen in the figure, elements `E2` and `E3` moved from nodes `5` and `6` in T_1 into `10` and `11` in T_2, respectively. These two move operations are the results of the model group sequence (node `4`) move operation. Note that `sequence`, in this example, is considered as an immediate parent node (which represents MV_P in Definition 9) of the moved elements `E2` and `E3`.

Figure 4 Example of element move



5.3 Phase 3: detection of deleted and inserted components

Intuitively, the deleted components are only available in the old tree version T_1, whereas inserted components are only available in the new one T_2. Based on the types of schema components introduced in Section 3.1, there are 12 types of operations for deleted and inserted components. Therefore, generic definitions for deleting or inserting the XML Schema components are provided. Deletions and insertions are maintained in lines `28–45` of the algorithm in Table 2.

5.3.1 Deletion changes

The idea behind detecting deletion changes for schema components is to eliminate the set of components existing in the old version T_1 from three sets: *matched nodes*, *migrated nodes* and *moved nodes*. All components in this paper are considered deleted.

We formally define a generic detecting of deleted components as follows.

Definition 10 (detect deleted component): *Let C_1 and C_2 be two sets of nodes of the same component type (e.g. element or attribute) from the first and second trees T_1 and T_2, respectively. Let M_T be a set of matched nodes of the same component type between T_1 and T_2, M_G be a set of migrated nodes of the same component type between T_1 and T_2, and M_V be a set of moved nodes of the same component type between T_1 and T_2. Node $c_1 \in C_1$ is deleted from T_1 iff:*

- *c_1 does not have a matched node in C_2 (i.e. $c_1 \notin M_T$);*
- *c_1 does not exist in the set of migrated components M_G (i.e. $c_1 \notin M_G$);*
- *c_1 does not exist in the set of moved components M_V (i.e. $c_1 \notin M_V$).*

Example: Element `E2` with node id `6` in Figure 1 is a deleted element node from T_1 tree, since all deletion conditions are satisfied.

Note that we store both delete and insert operations for the same component in one delta table as seen in Figure 2(b) (e.g. for elements, it is called `del_ins_element`) since both delete and insert operations share the same table structure. That is, it holds all information of the deleted/inserted component including the path where the node should be deleted from/inserted to, the type of an operation (`del` denotes to a deleted operation and `ins` means inserted one) and the second version id `<vj>` (of the compared schema versions) representing delta id.

5.3.2 Insertion changes

We can think of the insertion changes as the reverse of deletion operations. Inserted components are only available in the new version of the schema tree T_2; therefore, the same technique is used here to find inserted components. We formally define the detection of insertion as follows.

Definition 11 (detect inserted component): *Let C_1 and C_2 be two sets of nodes of the same component type (e.g. element or attribute) from the first and second trees T_1 and T_2, respectively, M_T be a set of matched nodes of the same component type between T_1 and T_2, M_G be a set of migrated nodes of the same component type between T_1 and T_2, and M_V be a set of moved nodes of the same component type between T_1 and T_2. Node $c_2 \in C_2$ is inserted to T_2 iff:*

- *c_2 does not have a matched node in C_1 (i.e. $c_2 \in M_T$);*
- *c_2 does not exist in the set of migrated components M_G (i.e. $c_2 \notin M_G$);*
- *c_2 does not exist in the set of moved components M_V (i.e. $c_2 \notin M_V$).*

Example: Nodes `21`, `22`, `23` and `26` in T_2 in Figure 1 are examples of element, complexType, sequence and facet insertions, respectively.

5.4 Phase 4: detection of updated components

The concept of an updated component in XML Schema has some similarities to an updated element in XML document. Some XML document change differencing methods consider the update operation for attributes and text nodes only because they are leaf nodes (Wang et al., 2003). For instance, attributes from both old and new versions are considered updated if their values are modified, meaning that the update operation is only measured on the attribute level. In our context of XML Schema, we define the update operation based on the node paths located in the path table. In other words, the path for the updated node should be available in both the old and the new versions of the schema tree, but here one or more of the node properties (XML Schema component attributes) might also have changed.

To detect the update changes for any type of schema components, we check its corresponding temporary table (e.g. `tmp_element` is the temporary matching table for the element component) to see if there are any changes to its properties. Note that properties in this check do not include the node's name since it is considered as a pair of insertion and deletion. We detect updated components in lines `47`–`55` of the algorithm in Table 2.

The following definition is for detecting the updated components.

Definition 12 (detect updated component): *Let M be a set of matched nodes of the same component type given by Definition 4, P_1 be a set of properties (e.g.* `type`*,* `minOccurs`*,* `maxOccurs` *or* `use`*) for each node in the schema tree T_1 and P_2 be a set of properties for each node in the schema tree T_2. Node $m \in M$ is updated iff $\forall p_i \in P_1$ and $p_j \in P_2$, $p_i \neq p_j$.*

Example: As a cardinality of `E5` element changes, the value of its attribute `maxOccurs` at line `14` in Table 1(a) is updated from '`5`' to become '`10`' in the same node in Table 1(b).

5.4.1 Order changes as updates

As stated earlier, order change for schema components under the `sequence` model group can be captured in this phase. We simply compare (using the tree information generated by the XSOM parser) the position of the old component (e.g. `element`) with the position of the new one. If the positions are different, we report this as a component update.

6 Experimental results

In this section, we evaluate XS-Diff algorithm by measuring three different factors: *correctness*, *performance* and *result quality*. The criterion of correctness means that the proposed algorithm can detect all the expected changes and generate the delta operations that are sufficient to transform the old tree into the new one. In other words, we measure how far the XS-Diff algorithm can capture XML Schema changes. Next, in the performance test, we measure the speed of the algorithm compared to the other available approaches. Finally, in the result quality, we test the minimality and the semantic correctness of the generated delta.

We compare our approach to the Java version of XyDiff (called jxydiff; www.github.com/tanob/jxydiff) (Cobena et al., 2002b), X-Diff (www.cs.wisc.edu/~yuanwang/xdiff.html) (Wang et al., 2003) and DeltaXML commercial tool (www.deltaxml.com/products/core/demo.html). The selection of these methods is based on the availability of the source/tool, the similarity of the input format (XML in our case) and the aim of the resulting delta (to translate one version of XML into another). Although methods for XML schema differencing, such as DTD-Diff (Leonardi et al., 2007) and DiffDog (AltovaDiffDog; Altova.com, 2014), are available, we do not compare the algorithm with them for the following reasons. In the case of the DTD-Diff algorithm, the types of changes of DTD are different from the ones in XML Schema, while in DiffDog, the output file (XSLT) that is produced by the tool cannot be compared to our resulting delta. XSLT file can only be used to update the related XML data files, which is the main purpose of the tool.

6.1 Experimental settings and data sets

We implement the XS-Diff algorithm using SQL queries on MySQL 5.5.24-log RDBMS. XS-Rel tables are created to store both XML Schemas and their delta changes. We use the Java programming language with an XSOM parser (https://xsom.java.net) to parse and store the input schemas. We conduct all experiments on a computer running an Intel Core i7 2.30 GHz processor with 8 GB of memory and Windows 7 Professional as the operating system.

We divide the data sets into *synthetic* and *real-world* XSDs. The synthetic XSDs listed in Figure 5 are modified from the ones available on W3C (www.w3.org/TR/xmlschema-0) and DATYPIC (www.datypic.com/books/defxmlschema/examples.html) websites. The second versions of synthetic XSDs are generated manually since we are not aware of any publicly available XML Schema generator tools. It is quite difficult to find two XSD versions that represent all types of XML Schema changes. For this reason, we select samples that cover all XML Schema changes produced in this work. We also consider different schema design patterns (e.g. Garden of Eden or Venetian Blind) to examine the performance when XSD transforms from one pattern into another. For real-world data sets, seven XSDs with their second versions are downloaded from online open standards (www.oasis-open.org, www.idealliance.org, and www.oagi.org) and their characteristics are listed in Figure 6.

Figure 5 Synthetic data sets

Dataset	Number of components								Focus area
	ED	AD	CT	ST	MG	F	GD	AG	
C01	18	6	4	2	4	2	0	0	ED and AD
C02	6	1	0	6	0	9	0	0	ST and F
C03	15	9	9	0	7	2	0	0	CT and MG
C04	14	1	3	0	4	0	0	0	MG and ED
C05	23	8	5	0	4	0	0	0	ED and AD
C06	8	6	2	0	5	0	6	4	GD and AG

Dataset	Number of components				File size (kb)	# of nodes	% of changes
	ED	AD	CT	ST			
MAILS01	84	47	36	14	19	274	0.00
MAILS02	84	46	36	14	20	273	5.48
MAILS03	87	45	35	13	23	275	16.39
MAILS04	86	44	35	15	26	277	31.22
MAILS05	70	41	29	12	28	247	44.15
MAILS06	72	45	27	12	30	247	56.81
MAILS07	67	41	28	14	33	251	77.14
MAILS08	129	55	35	17	46	398	90.03

Dataset	Design pattern	Number of components				File size (kb)	# of nodes
		ED	AD	CT	ST		
LIB01	Garden of Eden	121	48	25	12	14	267
LIB02	Venetian Blind	65	28	25	12	12	191
LIB03	Salami Slice	121	48	25	12	13	267
LIB04	Russian Doll	65	28	25	12	21	191

Figure 6 Real-world data sets. (a) Real XSD characteristics; (b) execution time (sec)

Dataset	Version	Number of components				File size (kb)	AVG nodes
		ED	AD	CT	ST		
3gpp	1	30	0	9	5	6	86
	2	33	0	10	6	7	
mismo	1	16	71	16	15	43	870
	2	177	1	23	21	59	
pidx	1	1133	62	153	73	112	2332
	2	1150	93	164	83	127	
AML	1	1432	46	343	123	311	5098
	2	1381	46	346	126	309	
OAGi	1	4543	208	964	377	455	8205
	2	4551	208	965	379	457	
papinet	1	5687	617	926	433	710	15042
	2	5863	658	965	448	737	
TXLife	1	12526	1732	1144	598	1149	18525
	2	12680	1753	1158	606	1164	

(a)

Dataset	XS-Diff	X-Diff	XyDiff
3gpp	0.055	0.016	0.112
mismo	0.126	0.88	2.289
pidx	0.3	0.107	3.739
AML	0.447	0.538	10.192
OAGi	0.961	0.588	8.418
papinet	4.17	2.274	14.522
TXLife	2.49	5.167	19.932

(b)

Table 4 Proposed edit operations (grouped by component) and their application on data sets (C: covered change, EO: expected occurrences, DO: detected occurrences)

Component	*Operation*	*C01*			*C02*			*C03*			*C04*			*C05*			*C06*		
		C	*EO*	*DO*	*C*	*EO*	*DO*	*C*	*EO*	*DO*	*C*	*EO*	*DO*	*C*	*EO*	*DO*	*C*	*EO*	*DO*
Attribute	Migration	✓	2	2															
	Move							✓	2								✓	1	1
	Deletion	✓	2	2						2				✓	1	1	✓	2	2
	Insertion	✓	2	2	✓	1	1			2				✓	1	1	✓	2	2
	Update	✓	3	3	✓	1	1												
Element	Migration	✓	2	2															
	Move							✓	3								✓	1	1
	Deletion	✓	3	3	✓	2	2	✓	2	5	✓	3	3	✓	2	2	✓	2	2
	Insertion	✓	4	4	✓	1	1	✓	2	5	✓	5	5	✓	2	2	✓	4	4
	update	✓	6	6	✓	3	3	✓	4	4	✓	1	1	✓	3	3	✓	3	3
complexType	Migration							✓	3	3							✓	1	1
	Move																		
	Deletion							✓	1	1									
	Insertion	✓	1	1				✓	1	1									
	Update							✓	1	1									
simpleType	Migration				✓	4	4												
	Move																		
	Deletion	✓	1	1	✓	1	1												
	Insertion	✓	1	1	✓	1	1												
	Update				✓	1	1												
Group	Move																✓	1	1
	Deletion																✓	2	2
	Insertion																✓	2	2
	Update																✓	1	1

Table 4 Proposed edit operations (grouped by component) and their application on data sets (C: covered change, EO: expected occurrences, DO: detected occurrences) (continued)

Component	*Operation*	*C01*			*C02*			*C03*			*C04*			*C05*			*C06*		
		C	*EO*	*DO*	*C*	*EO*	*DO*	*C*	*EO*	*DO*	*C*	*EO*	*DO*	*C*	*EO*	*DO*	*C*	*EO*	*DO*
attributeGroup	Move																✓	1	1
	Deletion																✓	2	2
	Insertion																✓	2	2
Model group	Move							✓	1	1							✓	1	1
Sequence	Deletion							✓	1	1	✓	1	1				✓	1	1
Choice	Insertion	✓	1	1				✓	1	1	✓	2	2				✓	1	1
All	Update							✓	1	2									
Facet, e.g. enumeration pattern	Move				✓	7	7	✓	2	2									
	Deletion	✓	1	1															
	Insertion	✓	2	2	✓	2	2												
Total		14	31	31	11	24	24	14	25	31	5	12	12	5	9	9	18	30	30

6.2 Correctness analysis

Correctness, as suggested by Cobena et al. (2002a), can be defined as a property which ensures that the differencing algorithm can find a set of operations that is sufficient to transform the old version into the new version of the XML document. This definition holds true in the XS-Diff algorithm because the inputs (XML Schemas) are mainly in XML document format. In order to verify the correctness of our algorithm, we run the experiments on the data sets (https://drive.google.com/file/d/0By77KsJuXVdcRjg1N0p6WDVNVlU/edit?usp=sharing) listed in Figure 5(a). On each data set, we target a particular segment of XML Schema components and simulate the relevant changes. The list of change operations grouped by the schema component is introduced in Table 4.

In C01 and C05 data sets, we focus on element and attributes changes. C01 uses the schema versions S_1 and S_2 in Table 1. The test contains 14 kinds of change operations with the expected occurrences of those operations at 31. As can be seen in column C01(DO) in Table 4 (DO denotes to detected changes), XS-Diff detects all expected changes $\left(\sum EO = \sum DO\right)$. Thus, XS-Diff succeeds in finding changes to this particular part of the schema. C05, on the other hand, demonstrates element and attribute changes with the existence of references. Again, the algorithm finds all the changes and reports them correctly.

In the C02 data set, we demonstrate *simple type* and *restriction facet* changes. The results here were also promising since XS-Diff was able to detect all targeted changes and report 24 total occurrences of these changes [see column C02(DO)].

We focus on *complex type* and *model group* changes in the C03 data set. As shown in column C03, the total amount of 25 expected changes (EO) are detected in addition to six more operations. This is because move operations in certain element and attribute nodes are counted as a pair of insertion and deletion operations. Although this will reduce the optimality of the algorithm, it will not affect the correctness of the algorithm, meaning that the resulting changes can still be used to generate a new version.

The C04 data set is dedicated to detect changes in a specific *model group* nesting structure. The consequences on other operations, such as *element order change*, are also tested. It mainly studies the situation where the model group (i.e. `sequence` or `choice`) is inserted as a child under another model group in the second version of the schema. The possibility of detecting deleted nested groups is also studied. It can be seen in column C04 in Table 4 that the XS-Diff algorithm detects 12 change occurrences correctly.

Attribute group and *model group definitions* are considered in the C06 data set. In this case, we test the ability of XS-Diff to find changes to *attributeGroup* and *group* nodes and their related components. As seen in column C06, the system clearly detects all the related changes in the groups.

6.3 Performance

6.3.1 Execution time vs. percentage of changes

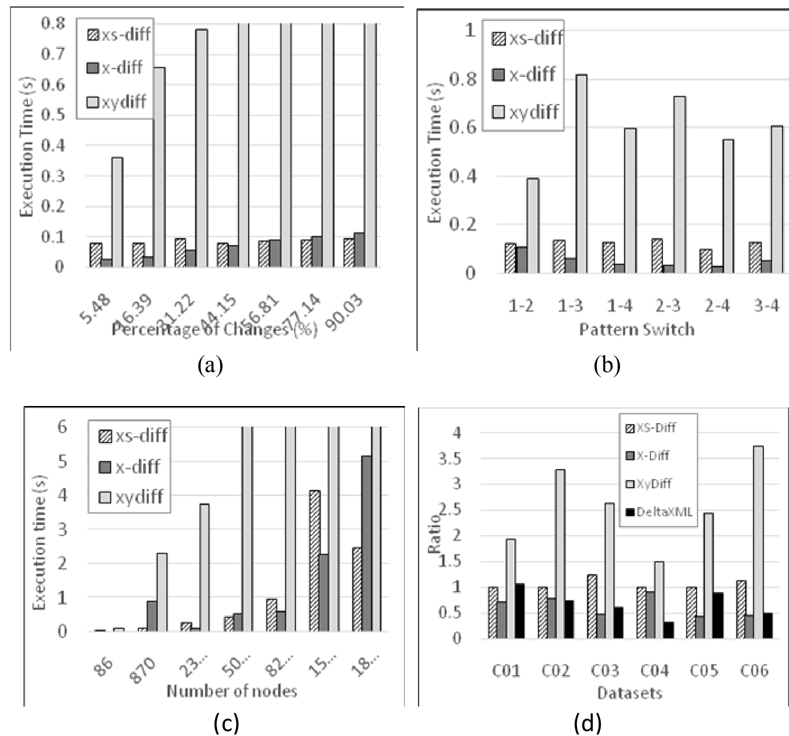
In this set of experiments, we study the effects of the percentage of changes on the execution time of XS-Diff, XyDiff and X-Diff. We do not compare this with DeltaXML since the source code and the algorithm are not available. The `MAILS01` data set (shown in Figure 5b) is used as a first version of the XSD. We vary the percentage of changes

from ‘5.48%’ to ‘90.03%’. Figure 7(a) depicts the performances of the compared algorithms. Recall that in XS-Diff, we use relational tables to store the parsed XML Schema components. The storage can be achieved by using different database systems; therefore, it is not affected by the database performance itself. For this reason, we exclude the time for the parsing phase from all compared approaches. We notice that XS-Diff is faster than XyDiff during this test. XS-Diff is slower than X-Diff for the first four data sets when the percentage of changes are ‘5.48%’, ‘16.39%’, ‘31.22%’ and ‘44.15%’. When ‘56.81%’ of XSD is changed, XS-Diff is nearly similar to the X-Diff approach. After this, XS-Diff outperforms X-Diff in the remaining data sets.

6.3.2 Execution time vs. pattern switch

XML Schema can be designed using four different patterns: *Garden of Eden*, *Venetian Blind*, *Salami Slice* and *Russian Doll*. The aim of this test is to study the performance of XS-Diff, X-Diff and XyDiff when the same XSD is transformed from one design pattern to another. The synthetic data sets ‘LIB01’ to ‘LIB04’ depicted in Figure 5(c) are used for this purpose, where only migration and move of type definitions and element/attribute declarations are applicable. The performance of this test is plotted in Figure 7(b). Note that ‘1-2’ in the horizontal axis represents the switch from ‘LIB01’ to ‘LIB02’ and so forth. Again, XS-Diff outperforms XyDiff along all data set switches but is not as fast as X-Diff.

Figure 7 Experimental results. (a) Time vs. percentage of changes; (b) time vs. pattern switch; (c) time vs. number of nodes; (d) result quality



We observe that the performance of XS-Diff and X-Diff is quite similar at the '`1-2`' pattern switch, i.e. when XSD is transformed from the *Garden of Eden* to *Venetian Blind* fashion. As we mentioned before (in the literature survey chapter), X-Diff supports seven types of operations. In the pattern switch test, the main operations detected by X-Diff are sub-tree delete followed by sub-tree insert, which group a set of atomic operations (e.g. possible internal `name`, `type` or `minOccurs` attribute changes). As a result, the number of operations is small compared to XS-Diff, which goes further and explore more types of changes convenient to XML Schemas. For example, in the '`1-4`' case (*Garden of Eden* to *Russian Doll*), all global definitions and declarations of the schema change to local. X-Diff records changes as `114` sub-tree element deletions followed by `1` sub-tree element insertion of the root element containing all changes. Thus, X-Diff is much faster but less expressive than XS-Diff (in terms of providing more detailed delta).

6.3.3 Execution time vs. number of nodes

The performance of XS-Diff and other algorithms is also studied using real-world XSDs. Figure 6(a) depicts the characteristics of seven real-world data sets. For each data set, we download two successive versions 1 and 2. To measure the performance against the number of nodes, the average number of nodes from the two versions is taken. The results of this test are given in Figure 7(c).

Figure 7(c) shows the performance of XS-Diff compared to X-Diff and XyDiff. At the beginning, all the algorithms display almost the same level of performance. This is because XSDs in this data set have few nodes (less than 100), so they do not need much time to be processed. At the '`mismo`' data set (see row 2 in Figure 6b), XS-Diff outperforms X-Diff and XyDiff. The number of inserted elements and restriction facets in that data set is relatively high. XS-Diff treats these insertions as an atomic operation, whereas X-Diff handles it as sub-tree insertions for both elements and facets. Thus, XS-Diff requires less time to execute. For up to 5000 nodes, XS-Diff and X-Diff show the same performance, but after this, X-Diff becomes better than XS-Diff. Surprisingly, in the last data set, '`TXLife`', with an average of 18,525 nodes, XS-Diff outperforms X-Diff with less execution time by around 2.5 seconds. Again, this is mainly because of the characteristics of the XSD versions being compared. For instance, the majority of elements and attributes are defined globally with no built-in types in the `TXLife` data set. For the XS-Diff algorithm, it is easy to store and match nodes of the same class, especially if they do not have sub-tree nodes. In contrast, the X-Diff algorithm handles each element declaration change as a sub-tree insertion and deletion. Given this fact, X-Diff is best suited to detect changes occurring at the leaf nodes (the content) of the trees but not the internal nodes (the structure).

6.4 Result quality

In this set of experiments, we study the result quality of the delta generated by XS-Diff from two different perspectives: *minimality* and *semantic correctness* of the generated operations. Minimality can be defined by the minimum number of edit operations versus the optimal number. Semantic correctness, on the other hand, is the property that ensures that the algorithm can generate delta relevant or close to XML Schema changes. Although the notion of minimality is important in some applications to save space required for storing changes, it becomes less significant if the generated changes are

semantically incorrect, especially in our context of XML Schemas. Therefore, both minimality and semantic correctness are discussed in the following subsections to analyse result quality.

6.4.1 Minimality

With the previous definition of minimality in mind, we calculate the result quality from the ratio between the number of change operations found by XS-Diff and the optimal number. In order to count the optimal operations for each test, we propose a set of minimal operations in Table 5, based on the basic operations of insert, delete, update and move that are supported by the majority of previous works on XML change detection. These operations are not only applicable on XML Schema changes but also on XML documents in general. Therefore, we use it as a guideline for introducing the optimal operations for the schema. In other words, the set of optimal operations guarantees the ideal number of operations required to perform each change. Based on this, we test the result quality of the generated operations through the synthetic data sets in Figure 5(a) and the ratio is plotted in Figure 7(d).

Table 5 Minimal operations required for XML schema changes (abbreviations ED, AD, CT, ST, MG, F, GD, AG are based on Definition 1)

No.	*XML Schema operation*	*Basic operation/s*	*Description*
1	ED or AD global-to-local **migration**	1 **delete** and 1 **update**	Global node is deleted and local node is updated
2	ED or AD local-to-global **migration**	1 **insert** and 1 **update**	Global node is inserted and local node is updated
3	CT or ST global-to-local **migration**	1 **move** and 1 **update**	Global node is moved to be under another node in the tree and local parent node property is updated
4	CT or ST local-to-global **migration**	1 **move** and 1 **update**	Local node is moved to the top level in the schema tree and the property of the parent of the local node is updated
5	ED, AD, CT, ST, F, MG, GD, or AG **move**	1 **move**	Node is moved from the old position to the new position between the schema tree versions
6	ED, AD, CT, ST, F, MG, GD, or AG **deletion** (does not include deleted nodes in no. 1)	1 **delete**	Node is deleted from the old schema tree version
7	ED, AD, CT, ST, F, MG, GD, or AG **insertion** (does not include inserted nodes in no. 2)	1 **insert**	Node is inserted to the new schema tree version
8	ED, AD, CT, ST, MG, GD, or AG **update** (does not include updated nodes in no. 1, 2, 3, or 4)	1 **update**	Node is updated at the new schema tree version

We observe that XS-Diff is capable of detecting optimal or near-optimal delta changes. The ratio of X-Diff ranges from `0.4` to `1.0`. This is because X-Diff supports two composite operations on non-leaf nodes: insert sub-tree and delete sub-tree. Although

composite operations can clearly minimise the delta by combining atomic operations, such as insert and delete, it may lead to deltas that are semantically incorrect. Semantic correctness is further discussed in the next section. In XS-Diff, such operations are not supported since the proposed changes are limited to schema component nodes. The ratio of XS-Diff resembles the optimal one in the C01, C02, C04 and C05 cases, and slightly above the optimal one in C03 and C06. This can be explained by generating a move (in certain situations) as a sequence of deletion and insertion operations. At the time of writing, XS-Diff supports a move operation when there is no copy of the same node at the target schema or the node still has the same parent at the target schema.

6.4.2 Semantic correctness

We provide an explanation of the semantic correctness using the following example. Consider the two element declarations `E2` and `E3` at nodes `6` and `7`, respectively, in T_1 in Figure 1. They are joined together under the new element named E8 at node `21` in T_2. X-Diff does not perform any particular investigation for this part of the tree. Instead, it deletes the sub-tree rooted at node `4` (E1T complex type) affected by these changes from the old version T_1 and inserts the updated sub-tree to the new version T_2. Similarly, DeltaXML updates the name of the first element E2 in the old version to the new value E8 in the new one and inserts the complex type sub-tree rooted at node `22` to the new version. However, the changes detected by these methods are semantically incorrect. More meaningful operations (with respect to XML Schema) would have been able to accomplish a host of other functions, such as insert the element declaration `E8` at node `21`, insert its complex type at node `22`, insert the sequence model group at node `23`, insert both elements `E2` and `E3` into their new positions `24` and `25`, respectively, and finally, remove `E2` and E3 from their old positions `6` and `7` in T_1.

7 Conclusion

Since existing tools for differencing general trees or XML documents are not designed for detecting changes to XML Schemas, they do not take the semantic and structural issues of such input schemas into account. In this paper, we present an approach for detecting meaningful changes to XML Schemas (XSDs) using relational databases. Our method is based on building an efficient relational model (XS-Rel) to store and compare XML Schemas. We also propose the XS-Diff algorithm as a new technique for XML Schema differencing. The algorithm is shown to be useful in several applications including (i) the revalidation of XML documents when their schema evolves, (ii) the incremental maintenance of relational schema generated by schema-conscious approach for storing XML data and (iii) the traditional support of XML versioning. XS-Diff takes two XSDs representing the old and the new version as input and generates a set of changes (stored in delta tables) containing the differences between the two versions. The key feature of XS-Diff is that it computes the changes by considering the tree structure of XML Schema.

In this study, we have proved the correctness of the proposed algorithm by testing it on both synthetic and real-world XSDs. In addition, we examined the performance and result quality in comparison to other XML document change detection tools, such as X-Diff, XyDiff and DeltaXML. We observed that the performance of XS-Diff is

comparable to X-Diff and much better than XyDiff. This can be illustrated by the new suggested operations (*global-to-local* and *local-to-global migrations*) that aggregate the atomic move and update operations. We also noticed that both X-Diff and DeltaXML produce deltas that fall below the optimal mark. This is because both tools are mainly designed for XML documents but not for schemas. On the other hand, XS-Diff produces optimal or near-optimal deltas and the resulting deltas allow the description of XML Schema changes in a more meaningful way.

As a future work, we aim to investigate the problem of XML Schema versioning. We will use the change detection method produced in this paper to develop a more robust versioning model for XML Schemas. We also aim to solve related issues of schema merging and conflict resolution.

References

Al-Ekram, R., Adma, A. and Baysal, O. (2005) 'diffX: an algorithm to detect changes in multi-version XML documents', *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research*, 17–20 October, Toronto, Ontario, Canada, pp.1–11.

Altova.com (2014) *Comparing XML schemas with DiffDog*. Available online at: http://www.altova.com/technote20.html (accessed on 24 May 2014).

Baqasah, A., Pardede, E., Holubova, I. and Rahayu, W. (2013) 'On change detection of XML Schemas', *Proceedings of the 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 16–18 July, Melbourne, VIC, pp.974–982.

Chawathe, S.S. and Garcia-Molina, H. (1997) 'Meaningful change detection in structured data', *Proceedings ACM SIGMOD International Conference on Management of Data*, 13–15 May, Tucson, AZ, pp.26–37.

Chawathe, S.S., Rajaraman, A., Garcia-Molina, H. and Widom, J. (1996) 'Change detection in hierarchically structured information', *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, 4–6 June, Montreal, Quebec, Canada, pp.493–504.

Cobena, G., Abdessalem, T. and Hinnach, Y. (2002a) *A comparative study for XML change detection*, National Institute for Research in Computer Science and Control, Rocquencourt, France. Available online at: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.58.3647 (accessed on 24 May 2014).

Cobena, G., Abiteboul, S. and Marian, A. (2002b) 'Detecting changes in XML documents', *Proceedings of the 18th International Conference on Data Engineering*, 26 February–1 March, San Jose, CA, pp.41–52.

Guerrini, G., Mesiti, M. and Rossi, D. (2005a) 'Impact of XML schema evolution on valid documents', *Proceedings of the 7th Annual ACM International Workshop on Web Information and Data Management*, 31 October–5 November, Bremen, Germany, pp.39–44.

Guerrini, G., Mesiti, M. and Rossi, D. (2005b) *XML schema evolution*, Department of Computer and Information Science, University of Genoa. Available online at: ftp://ftp.disi.unige.it/pub/.person/GuerriniG/reports/tr-schev-06.pdf (accessed on 24 May 2014).

IBM (1998) *XML TreeDiff 1998*. Available online at: http://www.xml.com/pub/r/536 (accessed on 24 May 2014).

Leonardi, E. and Bhowmick, S.S. (2005) 'Detecting changes on unordered XML documents using relational databases: a schema-conscious approach', *Proceedings of the 14th ACM International Conference on Information and Knowledge Management*, 31 October–5 November, Bremen, Germany, pp.509–516.

Leonardi, E. and Bhowmick, S.S. (2007) 'XANADUE: a system for detecting changes to XML data in tree-unaware relational databases', *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, 12–14 June, Beijing, China, pp.1137–1140.

Leonardi, E., Hoai, T.T., Bhowmick, S.S. and Madria, S. (2007) 'DTD-Diff: a change detection algorithm for DTDs', *Data and Knowledge Engineering*, Vol. 61, No. 2, pp.384–402.

Marian, A., Abiteboul, S., Cobena, G. and Mignet, L. (2001) 'Change-centric management of versions in an XML warehouse', *Proceedings of the 27th International Conference on Very Large Data Bases*, 11–14 September, Roma, Italy, pp.581–590.

Maruyama, H., Tamura, K. and Uramoto, N. (2000) *Digest values for DOM (DOMHASH)*, Available online at: http://www.research.ibm.com/trl/projects/xml/xss4j/docs/rfc2803.html (accessed on 24 May 2014).

Peters, L. (2005) 'Change detection in XML trees: a survey', *Proceedings of the 3rd Twente Student Conference on IT*, June, Enschede, the Netherlands, pp.1–8.

Raghavachari, M. and Shmueli, O. (2004) 'Efficient schema-based revalidation of XML', *Proceedings of the 9th International Conference on Extending Database Technology*, Heraklion, 14–18 March, Crete, Greece, pp.639–657.

Robin, L.F. (2001) *A delta format for XML: identifying changes in XML files and representing the changes in XML*, *XML Europe 2001*. Available online at: http://www.deltaxml.com/attachment/455-dxml/deltaxml-xml-europe-2001.pdf (accessed on 24 May 2014).

Shanmugasundaram, J., Tufte, K., He, G., Zhang, C., DeWitt, D. and Naughton, J. (1999) 'Relational databases for querying XML documents: limitations and opportunities', *Proceedings of the 25th International Conference on Very Large Data Bases*, 7–10 September, Edinburgh, Scotland, pp.302–314.

W3C (1999) *XML Path Language (XPath) Version 1.0*. Available online at: http://www.w3.org/TR/xpath/ (accessed on 24 May 2014).

W3C (2000) *Extensible Markup Language (XML) 1.0*. Available online at: http://www.w3.org/TR/2000/REC-xml-20001006 (accessed on 24 May 2014).

W3C (2004a) *XML Schema Part 0: primer second edition*. Available online at: http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/ (accessed on 24 May 2014).

W3C (2004b) *XML Schema Part 1: structures second edition*. Available online at: http://www.w3.org/TR/xmlschema-1/ (accessed on 24 May 2014).

Wang, Y., DeWitt, D.J. and Cai, J-Y. (2003) 'X-Diff: an effective change detection algorithm for XML documents', *Proceedings of the 19th International Conference on Data Engineering*, 5–8 March, Bangalore, India, pp.519–530.

Yoshikawa, M., Amagasa, T. and Shimura, T. (2001) 'XRel: a path-based approach to storage and retrieval of XML documents using relational databases', *ACM Transaction on Internet Technology*, Vol. 1, No. 1, pp.110–141.

Zhang, K. and Shasha, D. (1989) 'Simple fast algorithms for the editing distance between trees and related problems', *SIAM Journal on Computing*, Vol. 18, No. 6, pp.1245–1262.

Website

http://www.deltaxml.com/products/core/demo/.