Conversion of XML schema design styles with StyleVolution

Conversion of XML Schema Design Styles with StyleVolution

Zouhaier Brahmia
University of Sfax, Sfax, Tunisia
Fabio Grandi
DISI, University of Bologna, Bologna, Italy
Rafik Bouaziz
University of Sfax, Sfax, Tunisia

Abstract
Purpose − Any XML schema definition can be organized according to one of the following *design styles*: "Russian Doll", "Salami Slice", "Venetian Blind", "Garden of Eden" (with the additional "Bologna" style actually representing *absence of style*). Conversion from a design style to another can facilitate the reuse and exchange of schema specifications encoded using the XML Schema language. Without any computer-aided engineering support, style conversions are difficult and error-prone operations that must be performed very carefully. The purpose of this work is to efficiently deal with such XML Schema design style conversions.
Design/methodology/approach − A general approach, named StyleVolution, for automatic management of XML schema design style conversions is proposed. StyleVolution is equipped with a suite of seven procedures: four for converting a valid XML schema from any other design style to the "Garden of Eden" style, which has been chosen as a *normalized* XML schema format, and three for converting from the "Garden of Eden" style to any of the other desired design styles.
Findings – Procedures, algorithms, and methods for XML Schema design style conversions are presented. The feasibility of the approach has been shown through the encoding (using the XQuery language) and the testing (with the Altova XMLSpy 2019 tool) of a suite of seven ready-to-use procedures. Moreover, four test procedures are provided for checking the conformance of a given input XML schema to a schema design style.
Originality/value − The proposed approach implements a new technique for efficiently managing XML Schema design style conversions, which can be used to make any given XML Schema file to conform to a desired design style.
Keywords XML database, XML Schema, XML schema design style, XML schema translation, Schema change, Schema evolution.
Paper type Research paper

1. Introduction

The eXtensible Markup Language XML (W3C, 2008) along with the XML Schema language (W3C, 2004; Van der Vlist, 2011) is likely the description and specification formalism that has had the most significant impact on the development of Web-related technologies and applications. In particular, although XML has been designed as a general-purpose data storage and exchange format, it has been widely exploited for the representation and management of (semistructured) data in Web-based information systems, for which XML Schema is the elective data modeling formalism (Abiteboul *et al*., 2011; Aiken & Allen, 2004; Chaudhri *et al*., 2003).

Nowadays, XML and all languages based on it, like XML Schema, XQuery (W3C, 2014), XQuery Update Facility, XPath, and XSLT, continue to be of great interest for the developers of modern XML-based applications (e.g., Web services, cloud computing applications, social network applications, e-commerce systems) and designers/administrators of XML repositories or databases (Bourret, 2005; Bourret, 2010). Among the most important requirements of these actors, we find those concerning application maintenance (application source code correction, application extension, XML schema changes, etc.). In general, changes performed on XML schema files are error-prone and time consuming tasks (Klímek *et al.*, 2015), since they should be accomplished manually, as there is often no technical support (e.g., stand-alone or IDE-embedded computer-aided engineering tools) for performing them.

In the present work, we continue our research work on changes to the overall XML schema design patterns, which already covered changes to XML namespaces in (Brahmia *et al.*, 2016a; Brahmia *et al.*, 2016b). In particular, we focus here on changes involving XML schema design styles. In fact, any XML schema could be designed according to one of the following five styles (Maler, 2002; McBeath *et al.*, 2004; Lämmel *et al.*, 2005; Darr *et al.*, 2011; RCC, 2015; xFront, 2018): “Russian Doll”, “Salami Slice”, “Venetian Blind”, “Garden of Eden”, and “Bologna”. They differ on the way they define, globally or locally, XML schema components: element declarations, attribute declarations, simple type definitions, and complex type definitions. A global component is an immediate child of the root <xsd:schema> element; it is also automatically associated with the target namespace of the XML schema, and therefore it could be re-used in other XML schemas. However, a local component is not an immediate child of the <xsd:schema> element and, thus, it is not visible from the outside of the schema definition. The “Russian Doll” design style means having only one global element declaration that nests all other possible element/attribute declarations and simple/complex type definitions. In the “Salami Slice” style all element/attribute declarations are globally defined and all simple/complex type definitions are locally defined. In the “Venetian Blind” style all element/attribute declarations are defined as local components and all simple/complex type definitions are defined as global components. The “Garden of Eden” style defines all element/attribute declarations and all possible simple/complex type definitions as global components. Otherwise, an XML schema can be by default considered as conforming to the “Bologna” design style that is actually not a defined style[1].

Considering an XML database (DB), that is a repository of XML-encoded documents each one conforming to an XML Schema specification, the XML DB designer could need to translate an XML schema from its current design style to another one or putting an XML schema, whose style is undefined or unknown, into any desired style

The reasons behind a style conversion may be different and involve the features of global versus local definitions as described in the following. On the one hand, the use of local definitions emphasizes decoupling (i.e., definitions are self-contained, without dependence on other components) and cohesion (i.e., related data are grouped together into self-contained definitions) of specifications, and supports namespace complexity hiding (according to the value of the elementFormDefault). Notice that complex schema definition details can be

deliberately maintained hidden via local definitions to namespace users, also in order to preserve the intellectual property of the design (e.g., by adopting a "Russian Doll" style). On the other hand, the use of global definitions improves the sharing and re-use of specifications, as the definition of some subcomponent can be referenced (without code duplication and, thus, reducing the verbosity of schema definitions) by several component specifications in the same schema or made available to the designers of other schemas. With global definitions, the full complexity of namespaces can be exposed and the collaborative and incremental definition of even more complex schemas and namespaces can be supported. Reusability of components can be made available at different levels, according to the adopted style: the "Salami Slice" style only allows the re-use of element/attribute declarations, the "Venetian Blind" style only allows the re-use of simple/complex type definitions, whereas the "Garden of Eden" style allows the re-use of both element/attribute declarations and simple/complex type definitions.

Nevertheless, schema translation operations are not straightforward and should be performed carefully, since, from one hand, there are no available tools that allow performing such style design changes, and, from the other hand, some style changes can be difficult and error-prone to manually apply on large schemas and may also have side effects on the underlying XML instance documents (in such a case, they modify not only the XML schema presentation but also the XML schema specification). Therefore, our purpose is to allow the XML DB designers/administrators to automatically (i) change the design style of any XML schema to another design style, (ii) put an XML schema whose style is undefined or unknown into any desired style, (iii) correct design style errors in some existing XML schemas supposed to be in a given design style but not completely conformant (e.g., because developed by non-expert designers), and (iv) make available for re-use definitions that are local in a schema, via transformation into global definitions in a new schema. This should be done by means of automatic design style conversion tools, which should reduce the intervention of the schema designer to the minimum and generate, in a transparent manner, a new version of a given schema, semantically equivalent to it but conformant to the target style.

To this purpose, in this paper, we propose StyleVolution, a suite of seven procedures for efficiently managing design styles of XML schemas: putting an XML schema whose design style is undefined or unknown into a desired style, or converting an XML schema that is designed according to a given style into a different design style.

In order to define such procedures, we started by choosing the "Garden of Eden" design style as a normalized style, thanks to the fact that it maximizes reusability of specifications, since it globally exposes all element/attribute declarations and all simple/complex type definitions. After that, we have defined seven translation procedures that will be presented in this paper: four procedures, named RD2GE, SS2GE, VB2GE and BO2GE, for translating any "Russian Doll", "Salami Slice", "Venetian Blind" or "Bologna" XML schema to an equivalent schema designed according to the normalized style, respectively; and three procedures, named GE2RD, GE2SS, and GE2VB, for translating a "Garden of Eden" XML schema to an equivalent schema in any (desired) one of the three other non-default design styles, namely "Russian Doll", "Salami Slice", or "Venetian Blind", respectively.

Any design style conversion could then be performed with the direct use of either one procedure (when the target or the source design style is "Garden of Eden"), or with a combination of two procedures, one from the first subset {RD2GE, SS2GE, VB2GE, BO2GE} and the other from the second subset {GE2RD, GE2SS, GE2VB} (i.e., passing through the "Garden of Eden" style as an intermediate step of the conversion).

The rest of this paper is structured as follows. The next section describes the main XML Schema design styles found in the literature and used in XML-based application development. Section 3 introduces the procedures that we propose for converting any XML Schema, having

a defined (“Russian Doll”, “Salami Slice”, “Venetian Blind”, “Garden of Eden”) or a non-defined (“Bologna”) design style, into any other desired defined style. Section 4 deals with effects of changes to XML schema design styles on underlying XML document instances. Section 5 discusses related work and clarifies our contribution with respect to the state of the art. Section 6 summarizes the paper and gives some remarks about our future work. Furthermore, a generalized BO2GE procedure listing and test queries to detect the design style of an XML schema can be found as Appendices.

2. Background on XML Schema Design Styles

The main XML Schema design styles that have been proposed by the XML Schema community (McBeath *et al.*, 2004; Darr *et al.*, 2011; RCC, 2015; xFront, 2018) are four design styles: “Russian Doll”, “Salami Slice”, “Venetian Blind”, and “Garden of Eden”. When an XML schema does not conform to any one of these four styles, it is considered to be in the “Bologna” design style.

In this paper, we only refer to XML documents as usually considered for structured or semistructured data management (Abiteboul *et al.*, 2000), that is having a tree structure with elements as inner nodes and data values, children of elements or attributes, as leaves. Leaf data values have a predefined XML Schema type or a user-defined simpleType. Hence, all the procedures introduced in the paper assume to deal with XML schemas conformant to such an XML file structure. This choice will make the proposed algorithms easier to understand and the code shown in the paper shorter and more readable, with respect to conversion procedures working on general XML Schema definitions, which would be much more complex indeed. Although this could be seen as a limitation, such an approach is anyway significant, as data management is one of the most important application fields of XML, and database schema design is a fundamental activity of an information system lifecycle. In this context, the adoption of schema design styles embodies a disciplined attitude in the design of an XML DB schema. Moreover, our approach can also be extended to capture the most general case (exploiting the full XML syntax) without great additional difficulties, as it will be shown in Appendix A, where the code of a BO2GE general normalization procedure will be presented.

2.1. Russian Doll

In an XML schema designed according to the “Russian Doll” style, there is a single global complex element declaration that nests local components. Figure 1 presents an example of an XML schema of employees, in the “Russian Doll” style.

2.2. Salami Slice

In an XML schema designed according to the “Salami Slice” style, all (simple and complex) element declarations are defined as global components (i.e., as immediate children of the `<schema/>` element) and referenced where appropriate. Figure 2 presents the same example of XML schema of employees, already shown in Figure 1, but in the “Salami Slice” style.

Notice here that the XML DB schema designer should be careful of the problem of element/attribute name collisions. In fact, during the production of a new “Salami Slice” XML schema, one could find two or more global element/attribute declarations with the same name. Such a problem could be resolved for instance by (i) using namespaces, or (ii) appending some suffix (a string or a number) at the end of the name of each new global component (element or attribute) declaration having the same name of another already existing global component declaration, and inserting a comment that follows each one of these global components in order to specify their provenances. Thanks to these comments, this second solution allows obtaining the same source XML schema when applying the reverse operation on a produced Salami Slice XML schema.

2.3. Venetian Blind

In an XML schema designed according to the "Venetian Blind" style, all complex type definitions are globally defined and used when needed. Figure 3 presents the same example of XML schema of employees, already shown in Figure 1 and Figure 2, but in the "Venetian Blind" style.

The same notice mentioned above and dealing with the problem of element/attribute name collisions applies here but for complex type definition names. In fact, two or more global complex type definitions cloud have the same name, during the construction phase of a "Venetian Blind" XML schema. Obviously such a problem must be resolved if it appears.

Besides, if we inspire from the "Extreme Salami Slice" style proposed in (Lämmel, 2007), we could also propose the "Extreme Venetian Blind" style that provides also global simple type definitions based on XML Schema built-in simple types (e.g., `xsd:string`, `xsd:float`, `xsd:integer`). Indeed, in our example presented above, if we will consider such a style, we will have also three global simple type definitions derived by restriction from the XML Schema built-in simple types corresponding to the "name" and "salary" elements and to the "id" attribute. Our example of Figure 3 will become as shown by Figure 4.

In this work, since we aim at proposing procedures that automatically, that is without any interaction between the XML DB designer/administrator and the system, generate a new XML schema version according to a new design style which is different from that of the previous/source XML schema version, we do not consider the "Extreme Venetian Blind" style since the new simple type definitions that are generated actually do not make any restriction on the corresponding XML Schema built-in type. However, if we will consider an environment in which the new XML schema version is semi-automatically generated, that is there is some interaction between the XML DB designer/administrator and the system during the production of the new version, the "Extreme Venetian Blind" style could be of interest, since it can be used to prepare the ground for the designer/administrator to define his/her own simple type specifications by extending the proposed ones with some facets (e.g., `xsd:minInxclusive`, `xsd:maxInxclusive`, `xsd:minExclusive`, `xsd:maxExclusive`, `xsd:enumeration`).

2.4. Garden of Eden

In an XML schema designed according to the "Garden of Eden" style, both element/attribute declarations and complex type definitions are globally defined and referenced or used, respectively, when needed. Thus, this style combines both the "Salami Slice" and the "Venetian Blind" styles. Figure 5 presents the same example of XML schema of employees, already shown in Figure 1, Figure 2, Figure 3 and Figure 4, but in the "Garden of Eden" style.

Table 1 compares these four styles with regard to the scope (i.e., local or global) of XSD element/attribute declarations and XSD type definitions.

It is clear from Table 1 that from a reusability point of view, (i) the "Garden of Eden" is the best one as it allows reusing all element/attribute declarations and all type definitions, (ii) the "Russian Doll" style is the worst one as it defines locally all XML Schema components, (iii) the "Salami Slice" style allows reusing only element and attribute declarations, and (iv) the "Venetian Blind" style allows reusing only type definitions.

2.5. Bologna

With "Bologna" style (McBeath *et al.*, 2004), we mean any kind of valid and well-formed XML schema that does not fit into the "Russian Doll", "Venetian Blind", "Salami Slice" or "Garden of Eden" style formats. Hence, the "Bologna" style has been proposed basically to denote *absence of style* or "everything goes" (i.e., XSD files which store XML Schemas that work but are unstructured or messy). Figure 6 presents the same example of XML schema of

employees, already shown in Figure 1, Figure 2, Figure 3, Figure 4 and Figure 5, but in the "Bologna" style.

3. Design Style Conversion Procedures

In this section, we define the seven procedures making up the StyleVolution conversion kit, which allow designers to apply any desired design style to any valid XML Schema (whatever its initial design style was).

Notice that all the procedures introduced below assume to deal with XML schemas conformant to the XML file structure mentioned at the beginning of Section 2, i.e., the XML schema of any XML file with a tree structure, with element inner nodes and text content of elements or attribute values as leaves.

In the following, we start by choosing a normalized XML schema design style, which helps us defining the translation procedures. After that, we propose these procedures. Moreover, XQuery queries that test the conformance to design styles of a schema stored in a given XSD file can be found in Appendix B.

Notice that all our procedures are written in XQuery (W3C, 2014) and have been tested using the well-known Altova XMLSpy 2019 Enterprise Edition (rel. 3 sp1) tool[2]. Moreover, we have created a public GitHub project[3] in which we have made available our style conversion procedures along with the XSD files examples that we have used in this paper for testing them.

3.1. Choosing a normalized design style of any XML schema

In this work, we have chosen the "Garden of Eden" design style as a sort of *normalized version* of any XML schema, thanks to its characteristics mentioned above. This choice has allowed us to propose only seven conversion procedures instead of twelve, namely "Russian Doll" to "Garden of Eden" (RD2GE), "Salami Slice" to "Garden of Eden" (SS2GE), "Venetian Blind" to "Garden of Eden" (VB2GE), "Bologna" to "Garden of Eden" (BO2GE), from one hand, and "Garden of Eden" to "Russian Doll" (GE2RD), "Garden of Eden" to "Salami Slice" (GE2SS), "Garden of Eden" to "Venetian Blind" (GE2VB), from the other hand. Therefore, all the other translations can be defined as compositions of two of the procedures mentioned above, passing through the "Garden of Eden" style as an intermediate step. In particular, the style conversion procedure XX2YY, where XX, YY $\in$ {RD, SS, VB} (XX$\neq$YY), can be defined as the composition XX2GE $\circ$ GE2YY (e.g., the conversion from "Venetian Blind" to "Russian Doll" can be defined as the application of the "Venetian Blind" to "Garden of Eden" VB2GE conversion followed by the application of the "Garden of Eden" to "Russian Doll" GE2RD conversion).

The semantics of the XX2YY conversion procedures provided in the following is based on the assumption that the input schema is a correct XML Schema conforming to the design style XX. In practice, conformance to the XX design style can be tested, before the application of the style conversion, by means of the test procedures listed in Appendix B.

The proposed style conversion procedures are reversible, that is XX2GE and GE2XX, with XX $\in$ {RD, SS, VB}, are the inverse of each other (except for the ordering of global definitions, which we consider irrelevant as long as equivalent XML schemas are generated). As a consequence, reversibility is guaranteed also in the case of composition, that is XX2YY and YY2XX, where XX, YY $\in$ {RD, SS, VB}, are also the inverse of each other.

An issue we had to cope with in order to support reversibility is the resolution of naming conflicts that can arise when making global the definitions that were local in origin. In fact, when there are several components of the same type (e.g., `xsd:element` or `xsd:attribute`) that have the same name (i.e., having the same value of the attribute "name"), this situation would give rise to a naming collision (homonymity) when the definitions of such local

components are made global during the translation to a “Salami Slice” or “Garden of Eden” design style. Our solution to this problem consists in performing the translation in two passes, as described in the following:

1. In the first pass, the translation adds “_n” suffixes to equal names, where n is the occurrence number of the homonym; full XPath paths of the renamed components in the original schema are then added to the converted schema within XML comments, in order to include a sort of *provenance links* to their origin.

2. In the second pass, it checks whether the types of renamed with “_n” components are the same or compatible. In case their types are equal, suffixes and duplicate definitions are simply removed. In case their types are not exactly the same but compatible (e.g., strings with different lengths/constraints), the XML DB designer is interactively asked if he/she wants to introduce a unifying definition or to maintain the definitions distinct.

For example, let us assume to have the following XSD code snippet:

```
<xsd:schema ... >
 <xsd:element name="employees">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="employee">
     <xsd:compexType>
      <xsd:sequence>
       <xsd:element name="name" type="xsd:string" />
        ...
      </xsd:sequence>
     </xsd:complexType>
    </xsd:element>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="departments">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="department">
     <xsd:compexType>
      <xsd:sequence>
       <xsd:element name="name" type="xsd:string" />
        ...
      </xsd:sequence>
     </xsd:complexType>
    </xsd:element>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

The locally defined `<name/>` subelements of `<employee/>` and `<department/>` give rise to a naming conflict when their declaration is made global. Hence, during the first pass, the conflict is resolved by introducing suffixes (and provenance links in comments) as follows:

```
<xsd:element name="name_1" type="xsd:string" />
<!--
name_1=/xsd:schema/xsd:element[name="employees"]/xsd:complexType/xsd:sequen
ce/xsd:element[name="employee"]/xsd:complexType/xsd:sequence/xsd:element[na
me="name"] -->

<xsd:element name="name_2" type="xsd:string" />
```

```
<!--
name_2=/xsd:schema/xsd:element[name="departments"]/xsd:complexType/xsd:sequ
ence/xsd:element[name="department"]/xsd:complexType/xsd:sequence/xsd:elemen
t[name="name"] -->
```

In this way, the information contained in the provenance links enables full reversibility of the transformation: when translating back to the origin style, thanks to the path information stored in the comments, the global definitions of `<name_1/>` and `<name_2/>` elements become `<name/>` subelement definitions local to the `<employee/>` and `<department/>` element definitions, respectively.

Then, during the second pass, the translation procedure can check that the global definitions of `<name_1/>` and `<name_2/>` elements resulting from a renaming are exactly the same (i.e., they both have an xsd:string type). Hence, in this case, their definitions can be unified and the renaming canceled, while the provenance comments are merged. The final result is as follows:

```
<xsd:element name="name" type="xsd:string" />
<!--
name=/xsd:schema/xsd:element[name="employees"]/xsd:complexType/xsd:sequence
/xsd:element[name="employee"]/xsd:complexType/xsd:sequence/xsd:element[name
="name"],
name=/xsd:schema/xsd:element[name="departments"]/xsd:complexType/xsd:sequen
ce/xsd:element[name="department"]/xsd:complexType/xsd:sequence/xsd:element[
name="name"] -->
```

Notice that the resolution of naming conflicts can also be optimized by making a single pass (e.g., in the example above, there would be no need to produce the intermediate results involving the global definitions of `<name_1/>` and `<name_2/>` elements). However, we preferred to split it into two separated steps, which in general are both required, where the interaction with the user is required during the second pass only.

Furthermore, we have proposed an XQuery program, in Figure 20 of Appendix C, to technically solve this problem of naming conflicts; such a program is also put online within the aforementioned GitHub project.

For the sake of simplicity and in order to focus on the proper logic of the design style conversion only, in the rest of the work concerning the conversion procedures, we assume naming conflict never occur.

3.2. Procedures for translating an XML Schema from a given style to the "Garden of Eden" style

To convert an XML schema from a defined design style (i.e., "Russian Doll", "Salami Slice", "Venetian Blind", or "Garden of Eden") or an undefined style (i.e., "Bologna") to another different defined style, our proposal is based on using the "Garden of Eden" style as an intermediary. We propose in this Section four procedures that allow translation from any design style that is different from the "Garden of Eden" style to this latter (i.e., XX2GE translation procedures with XX $\in$ {RD, SS, VB, BO}). The procedures are defined as XQuery queries taking the schema to be converted as input and producing the converted schema as a result (notice that XML schemas themselves are basically encoded as XML files that, thus, can be processed with a standard XML transformation language like XQuery or XSLT).

Notice that since these four proposed procedures are simplified for the reasons provided above, they allow converting XML schema files that include only a subset of the components of the W3C XML Schema specification (W3C, 2004), as shown in Table 2. Nevertheless, the general BO2GE procedure presented in Figure 14 of Appendix A covers all XSD constructs of the XML Schema language, and allows the conversion of any valid XML Schema file, designed according to any style, to the "Garden of Eden" normalized style.

3.2.1. Procedure for translation from the "Russian Doll" to the "Garden of Eden" style

The RD2GE procedure could be defined in XQuery as shown in Figure 7.

The first for loop (on $ct) is aimed at making global the complex type definitions found at any level of nesting. The new global complex type is assigned a name obtained by adding a "Type" suffix to the container construct name (e.g., it becomes "employeesType" for the complex type definition inside the "employees" element definition in Figure 1). Subelement and attribute declarations making part of the complex type are processed by replacing local definitions with references to definitions that will become global.

The second loop (on $st) is aimed at making global the simple type definitions found at any level of nesting.

The rest of the loops (on $el and $at) are used to make global all the element and attribute declarations (either with a nested type definition or not) found anywhere in the input schema.

3.2.2. Procedure for translation from the "Salami Slice" to the "Garden of Eden" style

The SS2GE procedure could be expressed in XQuery as shown in Figure 8.

The first two loops (on $elc) are aimed at processing the declarations of element defined with a complex type (such elements are already globally declared in the SS style). The former loop extracts the inner complex type definition making it global (names are generated with a "Type" suffix as described for the RD2GE conversion), whereas the latter rewrites the global elements with a reference to the newly made global types.

The next two loops (on $eas and $els) are aimed at processing the declarations of elements and attributes defined with a simple type (such elements and attributes are already globally declared in the SS style). The former loop extracts the inner simple type definition making it global (names are generated with a "Type" suffix as described for the RD2GE conversion), whereas the latter rewrites the global containers (elements or attributes) with a reference to the newly made global types.

The last two loops (on $el and $at) simply copies the global declarations of elements and attributes having an XMLSchema predefined type.

3.2.3. Procedure for translation from the "Venetian Blind" to the "Garden of Eden" style

The VB2GE procedure could be formalized in XQuery as shown in Figure 9.

The first and last loops (on $el and $st, respectively) simply copy the (already global) element declarations and simple type definitions to the output schema.

The other loops (on $ct) are used to normalize the complex type definitions. In particular, the first one substitutes the local declarations of elements and attributes with a reference to definitions that will be made global. The second and the third one are responsible for making global the declaration of such elements and attributes (having an XML Schema predefined type), respectively.

3.2.4. Procedure for translation from the "Bologna" to the "Garden of Eden" style

In addition to the RD2GE, SS2GE and VB2GE translation procedures seen above, we have also defined a "Bologna" to "Garden of Eden" (BO2GE) translation procedure that can be used to normalize whatever kind of XSD file the XML DB designer can supply as input, by putting it into the "Garden of Eden" format, which is the normalized design style. Then, by composition of BO2GE with one of the three GE2YY translation procedures, with YY $\in$ {RD, SS, VB} (that will be presented in Sec. 3.3 below), the XML DB designer can convert his/her former Bologna XSD file into any desired (and desirable) design style. Notice that applying a "BO2YY" translation, with YY $\in$ {RD, SS, VB, GE}, is just a way to put the input XSD file in YY design style, whatever its initial contents might be.

We think that also having "BO2YY" conversion procedures at the disposal of the XML schema designer is important for the following reasons:

1) If an XML schema designer or database administrator does not exactly know which is the format of his/her source XSD schema file, a "BO2YY" conversion is the only procedure he/she can safely apply to normalize his/her XML schema definition and put it into his/her desired target YY style.

2) Most of the XSD files one may find around (including legacy XSD files, third-party XSD files, XSD files downloaded from the internet, large XSD files developed and modified by several authors over time) are likely to be in a true Bologna (i.e., "mixed") format.

3) One could also use them as correction tools for XSD files supposed to be in a given style but developed by non expert designers (e.g., to correct XML schemas which almost conform to a given style but contain some style errors). For instance, applying a "BO2GE" conversion to an XSD file supposedly written in the "Garden of Eden" style but containing style errors could be used to fix such errors[4].

The BO2GE procedure could be defined in XQuery as shown in Figure 10. It assumes to deal only with XML schemas conforming to structure of XML files mentioned at the beginning of Section 2, that is having a tree structure with element inner nodes and element or attribute textual values as leaves. An extension of this procedure to capture the most general case of XML Schema definition (e.g., including extension/restriction-based type derivations, choice/all/any/group constructs) is provided in Figure 14 of the appendix A. Such a definition is designed to work on any kind of XML Schema (W3C, 2004) with complex constructs. It is more general and complete but also much more long and complex than the procedure presented in this subsection. In fact, with the assumption made above on the construct present in the XSD input file, the definition of BO2GE in Figure 10 is not very complex and, basically, resumes the conversion operations previously seen in the definitions of the other XX2GE procedures.

In particular, the first loop (on $ct) is used to make global all complex type definition and is a simple extension of the code already seen for complex type processing in RD2GE and VB2GE. The name assigned to the complex type is the name it already had in the input schema (since its definition was already global) or is generated from the container name by adding the suffix "Type".

The second loop (on $st) is aimed at dealing with simple types making all their definitions global, similarly to how it is done in RD2GE. Also in this case, the name assigned to the simple type is the name it already had in the input schema or is generated from the container name by adding the suffix "Type".

The last two loops (on $el and $at) make global all the declarations of elements and attributes, respectively, which could already be global or not in the input schema. In this case, if the component was already declared as global (i.e., it had a type attribute), its definition is basically copied to the output schema. If it was declared as local, a reference to the name of its type (whose definition the first two loops ensure that will be global in the output schema) has to be generated from the container name by adding the suffix "Type".

Notice that, since an XML Schema in RD, SS or VB style can also be considered to be in BO style, the BO2GE procedure could be always used in place of the other XX2GE procedures previously described in Sections 3.2.1, 3.2.2, and 3.2.3 to normalize the schema. However, providing and using the leaner and more specific conversion procedures (that can be implemented in an optimized form) is better for efficiency reasons.

3.3. Procedures for translation from the "Garden of Eden" style to a defined design style

As we have already defined in Section 3.2 the four procedures that allow converting an XML Schema from any given design style to the "Garden of Eden" style, we propose in this section three procedures that allow translation from the "Garden of Eden" style to any other different defined design style (i.e., GE2YY translation procedures with YY $\in$ {RD, SS, VB}). As we

already observed, the composition of a "normalization" XX2GE conversion procedure with a "denormalization" GE2YY conversion procedure makes it possible to convert any XX design style into any other YY design style. Notice also that it does not make any sense to define XX2BO conversion procedures (including GE2BO), since any arbitrary transformation applied to a given schema always produces an XML Schema that can be considered to be in BO style (as every XML Schema can be considered to be in BO style).

Besides, it is worth mentioning that since these three procedures are defined in a simplified way for the reasons specified above, they allow converting XML schema files that include only the following components: `<xsd:complexType>`, `<xsd:element>` (having a simple or a complex type, including that with a "ref" attribute), `<xsd:attribute>` (including that with a "ref" attribute), `<xsd:sequence>`, `<xsd:simpleType>` with both `<xsd:restriction>` and `<xsd:pattern>` or both `<xsd:restriction>` and `<xsd:length>`.

3.3.1. Procedure for translation from the "Garden of Eden" to the "Russian Doll" style

The GE2RD procedure could be defined in XQuery as shown in Figure 11. The semantics of the XQuery code of this procedure is mainly based on a recursive function `typeFold`, that "folds" global type definitions into a local type definition. Recursion is used for nesting all (sub)element declarations into a single global element declaration by means of the type definitions and following the linking between element/type names and references.

In particular, if $t is the reference to the type of an element, the function `typeFold` returns the definition of the type. If $t is an XMLSchema predefined type, it simply returns the definition of an attribute "type" with name $t (base of recursion). Else, it finds the global declaration of the element named $t and returns an element declaration with the same name and, if it has a complex type, it makes local the declaration of its subelements and attributes and then makes recursive calls for the type of its subelements.

Hence, the procedure is based on a loop (on $el), searching for the declaration of elements non referenced by any other element, which generates the only global element declaration in the resulting schema and whose type definition is built as return value of the `typeFold` function.

3.3.2. Procedure for translation from the "Garden of Eden" to the "Salami Slice" style

The GE2SS procedure could be defined in XQuery as shown in Figure 12.

The first loop (on $el) is aimed at processing element declarations. For each element declaration $el, the variable $el1 is bound to the global definition of its type (i.e., a complex or simple type definition). Such a definition is made local by nesting it into $el in the output schema. In case its type is an XMLSchema predefined one, the element declaration is simply copied to the output schema.

The second loop (on $at) is aimed at processing attribute declarations. For each attribute declaration $at, the variable $at1 is bound to the global definition of its type (i.e., a simple type definition). Such a definition is made local by nesting it into $at in the output schema. In case its type is an XMLSchema predefined one, the attribute declaration is simply copied to the output schema.

3.3.3. Procedure for translation from the "Garden of Eden" to the "Venetian Blind" style

The GE2VB procedure could be defined in XQuery as shown in Figure 13.

The first loop (on $el) simply copies to the output schema the declaration of elements which are not referenced as subelements in any other element definition: this generates the only global element declaration in the resulting schema.

The second loop (on $ct) is aimed at processing complex type definitions, which remain global in the output schema. For each subelement and attribute declared inside $ct, its

declaration is made local with reference to its XMLSchema predefined type or to its type name if it has a complex or simple type (defined as $ct1 or $st1, respectively).

The last loop (on $st) simply copies the simple type definitions (which remain global) to the output schema.

4. Effects of XML Schema Design Styles Changes on XML Instances

When the XML DB designer wants to change the XML schema design style, in fact he/she basically aims at changing the presentation format of the schema but not its specification. As long as the XML schema specification does not change, there is no need for changing the XML document instances (i.e., even though its specification has been reorganized, the schema remains the same, so the instances should). Therefore, changing the schema design style is transparent to the instance management and leaves XML instance files unchanged. However, in practice, there are some design style changes that require some amendment to the schema specification and, thus, require propagation to the instances. Such style changes correspond to XSD element (attribute, respectively) renamings which are required when transforming local XSD element (attribute, respectively) components to global ones, while there are at least two XSD element (attribute, respectively) components that satisfy one of the following conditions:

- they have the same name but different types;
- they have the same name, and compatible types (e.g., types derived from the same base type but having different facets).

To better explain this issue, suppose having the following extract of an XML schema document:

```
...
  <xsd:element name="product">
   <xsd:compexType>
    <xsd:sequence>
     <xsd:element name="code" type="xsd:string" />
     ...
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
  <xsd:element name="supplier">
   <xsd:compexType>
    <xsd:sequence>
     <xsd:element name="code" type="xsd:positiveInteger" />
     ...
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
...
```

If the XML DB designer wants to transform the current design style of such a schema to the "Garden of Eden" or the "Salami Slice" style, the local `<xsd:element name="code" type="xsd:string" />` and `<xsd:element name="code" type="xsd:positiveInteger" />` element declarations must become global. Since these two XSD elements have the same name but different types, the two-step naming conflict resolution procedure presented in Section 3.1 has to be applied. As a result, the first step causes a renaming with suffix "_n" of the homonyms in the element declarations which are made global and generation of comments with provenance links as follows:

```
<xsd:element name="code_1" type="xsd:string" />
<!-- code_1= ... xsd:element[name="product"]/xsd:complexType/
                                  xsd:sequence/xsd:element[name="code"] -->
<xsd:element name="code_2" type="xsd:positiveInteger"/>
```

```
<!-- code_2= ... xsd:element[name="supplier"]/xsd:complexType/
                                    xsd:sequence/xsd:element[name="code"] -->
```

In this case, the second step has no effects as the types of `<code_1/>` and `<code_2/>` are incompatible. Hence, in all the instance XML documents referencing the converted schema, all the `<code/>` elements which are subelements of `<product/>` have to be renamed to `<code_1/>` and all the `<code/>` elements which are subelements of `<supplier/>` have to be renamed to `<code_2/>`, in order to maintain conformance to the converted schema.

On the contrary, notice that when the step 2 takes place and the types of homonyms are the same, equal homonym declarations are replaced by a single declaration and renaming is canceled. Therefore, propagation to XML document instances is avoided. The same happens when homonym types are compatible and the interactively asked XML DB designer opts for the introduction of a unifying type definition. Only in the case the XML DB designer wants to maintain the definitions distinct, the renaming cannot be canceled and requires propagation to instances.

5. Related Work Discussion

Several previous works have dealt with defining and presenting XML Schema design styles, like (Maler, 2002), (McBeath *et al.*, 2004), (Lämmel *et al.*, 2005), (Khan & Sum, 2006), (Lämmel, 2007), (Jordan & Waldt, 2010), (Darr *et al.*, 2011), (RCC, 2015), and (xFront, 2018), but none of them has studied style changes and style conversions of an XML schema.

In the realm of information systems, XML has been largely adopted as a semistructured data model and XML Schema as a semistructured data modeling formalism (Abiteboul *et al*., 2011; Aiken & Allen, 2004; Chaudhri *et al*., 2003). In this context, although a lot of research work has been done on XML schema evolution (Klettke, 2007; Guerrini & Mesiti, 2008; Cavalieri *et al.*, 2011; Domínguez *et al.*, 2011; Nečaský *et al.*, 2012; Amavi *et al.*, 2014; Klímek *et al.*, 2015) and on XML schema versioning (Dyreson *et al.*, 2006; Brahmia & Bouaziz, 2008; Snodgrass *et al.*, 2008; Malý *et al.*, 2011; Baqasah *et al.*, 2014; Brahmia *et al.*, 2014a; Brahmia *et al.*, 2016b; Brahmia *et al.*, 2018a), none of such approaches has dealt with changes involving XML schema design styles.

Beyond its use as a data storage and exchange format, XML has also been used as a modeling language (Fishwick, 2002; Huang *et al.*, 2005; Cortellessa *et al.*, 2014) and XML Schema, in addition to having been used as a general-purpose data modeling formalism (Mani *et al.*, 2001; Yan *et al.*, 2009; Hacherouf *et al.*, 2019), has also been adopted as a metamodeling language in different software engineering contexts (Bordbar & Staikopoulos, 2004; Kensche *et al.*, 2007; Wang *et al.*, 2011). However, design style changes and conversions have not been considered in such works either.

The authors of (La Fontaine & Nichols, 2003) have used the “Russian Doll” format to propose a multi-versions XML file, that is an XML file that stores multiple versions of the same file.

Lämmel (2007) has worked on the transformation of an XML Schema file to an object model. Contrarily to him, we have proposed in this paper a set of translation procedures that transform an XML schema to another equivalent schema but having a different design format.

In (Brahmia *et al.*, 2014b), the authors have proposed (among others) a large set of high-level operations for changing XML schemas, in a schema versioning (Brahmia *et al.*, 2015) environment, but no operation that acts on the design style of an XML schema has been considered. In order to extend such a framework, in this work, we have proposed seven procedures that could be used as high-level operations for performing any desired design style conversion involving a given XML schema.

Brahmia *et al.* (2016a, 2016b) have proposed an approach for managing changes to XML namespaces in XML schemas and their effects on underlying XML instances, in a setting that

supports schema versioning, but they have not studied the support of XML Schema design styles and of their evolution. In this paper, we complete the picture by dealing with such evolution and showing the use of our proposed translation procedures for changing design styles of XML schemas while schema versioning is being supported.

Costello and Utzinger (2018) have proposed a set of eight recommendations to minimize the impact of XML schema versioning on underlying XML instance documents, running applications and related XML schemas. In fact, the authors consider a system as a set of three components: XML schemas, XML instance documents, and applications and try to provide an answer to the question "how designing a system so that the effects of creating each new XML schema version on the other components of this system (i.e., XML instances, applications, and the other schemas) could be reduced to their minimum level?" They also define six categories of XML schema changes (Namespace, Location, Change, Shuffle, Remove, and Add), but they do not deal with changes to XML schema design styles.

As far as available (commercial) XML management tools are concerned, Altova XMLSpy 2019 Enterprise Edition[5], Liquid XML Studio 2018[6], and Oxygen XML Editor 20.0[7] do not provide any support for XML schema design styles.

Only the free and open-source integrated development environment NeatBeans IDE 8.2[8], extended with the XMLTools4NetBeans[9] plugin, provides support of design styles to a limited extent. In particular, it provides some support for applying a design pattern to an XSD file under development, without any support for managing design pattern transformations. In fact, while playing with it, we have noticed the following limitations:

- It does not define attributes as "ref" components; `<attribute/>` components are always defined as local components to their parent-components.
- It does not provide any solution for the problem that happens when making global elements that have the same name but different types; on the contrary, it provides a schema which is not "faithful" to the source schema or not reversible (i.e., it does not allow generating the source schema). Indeed, we noticed that it considers only the last type of the corresponding element in the new schema. For example, suppose that our XSD file includes a subelement (of the `<employee/>` element) "Id" with "`xsd:string`" type, a subelement (of the `<product/>` element) "Id" with "`xsd:int`" type, and a subelement (of the `<customer/>` element) "Id" with "`xsd:date`" type. When putting such a schema into the "Garden of Eden" style, NetBeans keeps, in the new schema, only a single "Id" element with the last type found in the source schema (i.e., "`xsd:date`"), which overrides the type definitions of the other two homonyms.
- It does not support/know the "Bologna" design style; it automatically considers that every well-formed and valid XSD file is under one of the four design styles: "Russian Doll", "Salami Slice", "Venetian Blind", or "Garden of Eden". For example, it considers the "Bologna"-style XSD file of Figure 6 to be in the "Salami Slice" style, which is actually wrong.
- In some cases, it provides some results that are not consistent with its definitions of the design styles. Indeed, we could consider them as erroneous results although NetBeans generates a well-formed and valid XSD file. In the following, we just provide two examples of this misbehavior:
  - It considers the XSD files of Figure 3 and Figure 4, which are actually in the "Venetian Blind" style, to be in the "Garden of Eden" (according to its definition of the "Garden of Eden" style: all elements and types are defined in the global namespace with the elements referenced as needed) whereas there are four elements that are locally defined in these XSD files: `<employee/>`, `<name/>`, `<salary/>` and `<password/>`.
  - Moreover, our schemas in Figure 3 and Figure 4 satisfy its definition of the "Venetian

Blind" style (in the true "Venetian Blind" design, there is a single global element; all other elements are local. Element declarations are nested within a single global declaration, using named complex types and element groups; complex types and element groups can be reused throughout the schema; only the root element must be defined within the global namespace), since there is only one single global element and some global complex/simple types.

- It considers that the XSD file of Figure 6 is in the "Salami Slice" style (according to its definition of the "Salami Slice" style: in the Salami Slice design, all elements are global; there is no nesting of element declarations and element declarations can be reused throughout the schema; all elements must be defined within the global namespace), whereas there are three elements that are locally defined in this XSD file: `<name/>`, `<salary/>` and `<password/>`.

Notice that our proposals overcome all the limitations of NetBeans, which are sketched above.

In (Brahmia *et al.*, 2018b), we have presented a preparatory work for the current proposal. In fact, we introduced a "Normalize" procedure, which can be used to convert any given XML schema to the "Garden of Eden" style, by automatically transforming and rearranging all declarations and definitions it contains. With respect to that paper, in the present work, we completed the style-conversion picture by also defining the procedures to be used to denormalize a "normalized" XML schema. Furthermore, in this Section, we framed and discussed our approach with respect to the state-of-the-art of the related works.

6. Conclusion

In this paper, we have mainly dealt with the following problem: how to make any given XML Schema file to conform to a desired design style? As a solution to this problem, we have proposed a suite of procedures, collectively named StyleVolution, for applying and changing design styles to an XML schema. In order to define such procedures, we started by choosing the "Garden of Eden" style, thanks to its advantages, as a normalized design style. After that, we have defined (in a formal but also ready-to-use way, using the XQuery language) seven translation procedures: four procedures for translating an XML schema, whatever its style is (i.e., "Russian Doll", "Salami Slice", "Venetian Blind" or "Bologna"), to the "Garden of Eden" design style, and three procedures for translating an XML schema from the "Garden of Eden" style to any one of the three other defined design styles (i.e., "Russian Doll", "Salami Slice", or "Venetian Blind"). We have also studied the effects of changes to XML schema design styles on the corresponding XML instances and showed that these changes, which in general act only on XML schema presentation, could have an impact also on XML schema specifications (in case of renaming of homonyms generated by the conversion) and, therefore, on all XML document instances that are valid to the changed XML schema. Notice that the provided procedures allow designers to re-use both existing XML schemas, by effectively reorganizing them according to a new style, and local XSD definitions in old schemas, by automatically transforming them into global ones in new schemas. In this way, XML-encoded specifications developed in the context of an information system design or application software engineering project can be more easily exchanged and shared among designers. In practice, since XML and XML Schema languages have often been used as modeling and metamodeling formalisms, we could consider our contribution in this work as the proposal of a software engineering tool that can be used for facilitating the computer-aided reuse and exchange of models and metamodels for data and software specifications.

In addition, in order to safely choose the most specific procedure needed to apply the conversion to a desired design style in the most convenient and efficient way, we also

proposed in Appendix B four XQuery test queries designed for checking the conformance of a given input XML schema to a schema design style.

In the future, we plan to develop a tool that demonstrates the usability of our proposal in a user-friendly integrated development framework. Moreover, since in our previous work (Brahmia *et al.*, 2016a; Brahmia *et al.*, 2016b), we have dealt with XML namespace changes in an environment that supports schema versioning, which is an aspect that has been ignored in this paper, although it is closely related to the issues studied in the current work (the proper use of namespaces also facilitates the modularity and the reuse and exchange of specifications; XML namespaces are involved when importing (through the `<xsd:import>` construct) or including (through the `<xsd:import>` construct) XML schemas), we also aim at investigating the combination of changes to XML namespaces with changes to XML schema design styles, in a multi-version XML context.

Notes

1. The name comes from the Bologna sausage, a finely ground pork salami, for which it is (unsubstantiated) folklore that everything could be put inside.
2. https://www.altova.com/xmlspy-xml-editor (accessed 26 June 2019)
3. https://github.com/ZouhaierBrahmia/StyleVolution
4. Notice that we only consider here *style* errors. Wrong XML Schema definitions cannot be fixed by changing the design style by means of our procedures.
5. https://manual.altova.com/xmlspy/spyenterprise/xmlspy_content.htm (accessed 26 June 2019)
6. https://www.liquid-technologies.com/xml-studio (accessed 26 June 2019)
7. https://www.oxygenxml.com/doc/versions/20.0/ug-editor/ (accessed 26 June 2019)
8. https://netbeans.org/ (accessed 26 June 2019)
9. http://plugins.netbeans.org/plugin/40292/xmltools4netbeans (accessed 26 June 2019)

References

Abiteboul, S., Buneman, P. and Suciu, D. (2000), *Data on the Web: From Relations to Semistructured Data and XML*, Morgan Kaufmann, San Francisco, CA, USA.

Abiteboul, S., Manolescu, I., Rigaux P., Rousset, M.-C. and Senellart, P. (2011), *Web Data Management*, Cambridge University Press, Cambridge, UK.

Aiken, P., Allen, D. (2004), *XML in Data Management: Understanding and Applying Them Together*, Morgan Kaufmann, Burlington, MA, USA.

Amavi, J., Chabin, J., Ferrari, M. H. and Réty, P. (2014), "A ToolBox for Conservative XML Schema Evolution and Document Adaptation", *Proceedings of the 25th International Conference on Database and Expert Systems Applications (DEXA'2014), Munich*, *Germany*, Part I, pp. 299-307.

Baqasah, A., Pardede, E. and Rahayu, W. (2014), "XSM - A Tracking System for XML Schema Versions", *Proceedings of the 28th IEEE International Conference on Advanced Information Networking and Applications (AINA 2014), Victoria*, *BC*, *Canada*, pp. 1081-1088.

Bordbar, B. and Staikopoulos, A. (2004), "Automated Generation of Metamodels for Web Service Languages", *Proceedings of the 2nd European Workshop on Model Driven Architecture (EWMDA-2)*, *Canterbury*, *England*, *UK*, available at: https://www.cs.kent.ac.uk/projects/kmf/mdaworkshop/submissions/Bordbar.pdf (accessed 26 June 2019)

Bourret, R. (2005), "XML and Databases", available at: http://www.rpbourret.com/xml/XMLAndDatabases.htm (accessed 26 June 2019)

Bourret, R. (2010), "XML Database Products", available at: http://www.rpbourret.com/xml/XMLDatabaseProds.htm (accessed 26 June 2019)

Brahmia, Z. and Bouaziz, R. (2008), "An approach for schema versioning in multi-temporal XML databases", *Proceedings of the 10th International Conference on Enterprise Information Systems (ICEIS'2008)*, *Barcelona*, *Spain*, Volume DISI, pp. 290-297.

Brahmia, Z., Grandi, F., Oliboni, B., Bouaziz, R. (2014a), "Schema Change Operations for Full Support of Schema Versioning in the τXSchema Framework", *International Journal of Information Technology and Web Engineering*, Vol. 9 No. 2, pp. 20-46.

Brahmia, Z., Grandi, F., Oliboni, B. and Bouaziz, R. (2014b), "High-level Operations for Creation and Maintenance of Temporal and Conventional Schema in the τXSchema Framework", *Proceedings of the 21st International Symposium on Temporal Representation and Reasoning (TIME'2014)*, *Verona*, *Italy*, pp. 101-110.

Brahmia, Z., Grandi, F., Oliboni, B. and Bouaziz, R. (2015), "Schema Versioning", in Khosrow-Pour, M. (Ed.), *Encyclopedia of Information Science and Technology (Third Edition)*, IGI Global, Hershey, PA, USA, pp. 7651-7661.

Brahmia, Z., Grandi, F. and Bouaziz, R. (2016a), "Changes to XML Namespaces in XML Schemas and their Effects on Associated XML Documents under Schema Versioning", *Proceedings of the 11th International Conference on Digital Information Management (ICDIM'2016)*, *Porto*, *Portugal*, pp. 43-50.

Brahmia, Z., Grandi, F. and Bouaziz, R. (2016b), "A Systematic Approach for Changing XML Namespaces in XML Schemas and Managing their Effects on Associated XML Documents under Schema Versioning", *Journal of Digital Information Management*, Vol. 14 No. 5, pp. 275-289.

Brahmia, Z., Grandi, F., Oliboni, B. and Bouaziz, R. (2018a), "Supporting Structural Evolution of Data in Web-Based Systems via Schema Versioning in the τXSchema Framework", in Elçi, A. (Ed.), *Handbook of Research on Contemporary Perspectives on Web-Based Systems*, IGI Global, Hershey, PA, USA, pp. 271-307.

Brahmia, Z., Grandi, F. and Bouaziz, R. (2018b), "Normalization of XML Schema Definitions", *Proceedings of the 7th International Conference on Software Engineering and New Technologies (ICSENT'2018)*, Hammamet, Tunisia, Article No. 2.

Cavalieri, F., Guerrini, G. and Mesiti, M. (2011), "Updating XML Schemas and Associated Documents through EXup", *Proceedings of the 27th International Conference on Data Engineering (ICDE'2011)*, *Hannover*, *Germany*, pp. 1320-1323.

Chaudhri, A. B., Rashid, A. and Zicari, R. (2003), *XML Data Management: Native XML and XML-Enabled Database Systems*, Addison-Wesley Professional, Boston, MA, USA.

Cortellessa, V., Di Marco, A. and Trubiani, C. (2014), "An approach for modeling and detecting software performance antipatterns based on first-order logics", *Software & Systems Modeling*, Vol. 13 No. 1, pp. 391-432.

Costello, R. L. and Utzinger, M. (2018), "Impact of XML Schema Versioning on System Design: Strategies for Facilitating System Evolution", xFront document, available at: http://www.xfront.com/SchemaVersioning.html (accessed 26 June 2019)

Darr, T., Hamilton, J., Fernandes, R. and Jones, C. H. (2011), "Design Considerations for XML-Based T&E Standards", *Proceedings of the 47th Annual International Telemetering Conference and Technical Exhibition – Telemetry: Blending the Art with Science and Technology (ITC 2011)*, *Las Vegas*, *NV*, *USA*, available at: http://arizona.openrepository.com/arizona/bitstream/10150/595666/1/ITC_2011_11-10-01.pdf (accessed 26 June 2019)

Domínguez, E., Lloret, J., Pérez, B., Rodríguez, Á., Rubio, A. L. and Zapata, M. A. (2011), "Evolution of XML Schemas and documents from stereotyped UML class models: A traceable approach", *Information & Software Technology*, Vol. 53 No. 1, pp. 34-50.

Dyreson, C. E., Snodgrass, R. T., Currim, F., Currim, S. and Joshi, S. (2006), "Validating Quicksand: Schema Versioning in τXSchema", *Proceedings of the 22nd International Conference on Data Engineering Workshops (ICDE Workshops 2006)*, Atlanta, GA, USA, p. 82.

Fishwick, P. A. (2002), "XML-based modeling and simulation: using XML for simulation modeling", *Proceedings of the 34th Winter Simulation Conference: Exploring New Frontiers (WSC 2002)*, *San Diego*, *CA*, *USA*, pp. 616-622.

Guerrini, G. and Mesiti, M. (2008), "X-Evolution: A Comprehensive Approach for XML Schema Evolution", *Proceedings of the 19th International Conference on Database and Expert Systems Applications Workshops (DEXA Workshops 2008)*, *Turin*, *Italy*, pp. 251-255.

Hacherouf, M., Nait-Bahloul, S. and Cruz, C. (2019), "Transforming XML schemas into OWL ontologies using

formal concept analysis", *Software & Systems Modeling*, Vol. 18 No. 3, pp. 2093-2110.

Huang, C. C., Tseng, T. L. and Kusiak, A. (2005), "XML-based modeling of corporate memory", *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, Vol. 35 No. 5, pp. 629-640.

Jordan, C. D. and Waldt, D. (2010), "Schema scope: Primer and best practices - Understand a crucial aspect of schema design", IBM developerWorks, available at: https://www.ibm.com/developerworks/library/x-schemascope/x-schemascope-pdf.pdf (accessed 26 June 2019)

Kensche, D., Quix, C., Chatti, M. A. and Jarke, M. (2007), "*GeRoMe*: A Generic Role Based Metamodel for Model Management", in Spaccapietra, S. et al. (Eds.), *Journal on Data Semantics VIII*, LNCS Vol. 4380, Springer-Verlag, Berlin, Heidelberg, Germany, pp. 82-117.

Khan, A. and Sum, M. (2006), "Introducing Design Patterns in XML Schemas", Oracle Technology Network document, available at: http://www.oracle.com/technetwork/java/design-patterns-142138.html (accessed 26 June 2019)

Klettke, M. (2007), "Conceptual XML Schema Evolution - the CoDEX Approach for Design and Redesign", *Proceedings of the 12th Conference on Database systems for Business, Technology and Web Workshops (BTW Workshops 2007)*, Aachen, Germany, pp. 53-63.

Klímek, J., Malý, J., Nečaský, M. and Holubová, I. (2015), "eXolutio: Methodology for Design and Evolution of XML Schemas using Conceptual Modeling", *Informatica (Lithuanian Academy of Sciences)*, Vol. 26 No. 3, pp. 453-472.

La Fontaine, R. and Nichols, T. (2003), "Russian Dolls and XML: Handling Multiple Versions of XML in XML", *Proceedings of the XML 2003 Conference (XML 2003)*, *Philadelphia*, *PA*, *USA*, available at: https://docs.deltaxml.com/support/files/latest/2130869/2130872/1/1522331772123/deltaxml-paper-xml-2003.pdf (accessed 26 June 2019)

Lämmel, R., Kitsis, S. and Remy, D. (2005), "Analysis of XML schema usage", *Proceedings of the XML 2005 Conference (XML 2005)*, *Atlanta*, *GA*, *USA*, available at: http://www.pdfpower.com/XML2005Proceedings/ship/49/paper.PDF (accessed 26 June 2019)

Lämmel, R. (2007), "Style normalization for canonical X-to-O mappings", *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM 2007)*, *Nice*, *France*, pp 31-40.

Maler, E. (2002), "Schema Design Rules for UBL...and Maybe for You", *Proceedings of the XML 2002 Conference and Exposition (XML 2002)*, *Baltimore*, *Maryland*, *USA*, available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.199.5993&rep=rep1&type=pdf (accessed 26 June 2019)

Malý, J., Mlýnková, I. and Nečaský, M. (2011), "XML Data Transformations as Schema Evolves", *Proceedings of the 15th International Conference on Advances in Databases and Information Systems (ADBIS'2011)*, *Vienna*, *Austria*, pp. 375-388.

Mani, M., Lee, D. and Muntz, R. R. (2001), "Semantic data modeling using XML schemas", *Proceedings of the 20th International Conference on Conceptual Modeling (ER'2001)*, *Yokohama*, *Japan*, pp. 149-163.

McBeath, D., Farrell, J. and Hinkelman, S. (2004), "XML Schema Design Guidelines – Version 1.3", MedBiquitous Consortium, 25 October 2004, available at: https://medbiq.org/std_specs/techguidelines/xmldesignguidelines.pdf (accessed 26 June 2019)

Nečaský, M., Klímek, J., Malý, J. and Mlýnková, I. (2012), "Evolution and change management of XML-based systems", *Journal of Systems and Software*, Vol. 85 No. 3, pp. 683-707.

Range Commanders Council (RCC). (2015), "XML Style Guide", Standard RCC 125-15, July 2015, available at: https://apps.dtic.mil/dtic/tr/fulltext/u2/a627623.pdf (accessed 26 June 2019)

Snodgrass, R. T., Dyreson, C. E., Currim, F., Currim, S. and Joshi, S. (2008), "Validating Quicksand: Temporal Schema Versioning in τXSchema", *Data & Knowledge Engineering*, Vol. 65 No. 2, pp. 223-242.

Van der Vlist, E. (2011), *XML Schema: The W3C's Object-Oriented Descriptions for XML*, O'Reilly Media, Sebastopol.

W3C. (2004), "XML Schema Part 0: Primer Second Edition", W3C Recommendation, 28 October 2004, available at: http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/ (accessed 26 June 2019)

W3C. (2008), "Extensible Markup Language (XML) 1.0 (Fifth Edition) ", W3C Recommendation, 26 November 2008, available at: https://www.w3.org/TR/2008/REC-xml-20081126/ (accessed 26 June 2019)

W3C. (2014), "XQuery 3.0: An XML Query Language", W3C Recommendation, 2 May 2014, available at: https://www.w3.org/TR/2014/REC-xquery-30-20140408/ (accessed 26 June 2019)

Wang, Y., Albani, A. and Barjis, J. (2011), "Transformation of DEMO Metamodel into XML Schema", *Proceedings of the 1st Enterprise Engineering Working Conference (EEWC 2011)*, *Antwerp*, *Belgium*, pp. 46-60.

xFront. (2018), "Global versus Local: A Collectively Developed Set of Schema Design Guidelines", xFront document, available at: http://www.xfront.com/GlobalVersusLocal.html (accessed 26 June 2019)

Yan, L., Ma, Z. M. and Liu, J. (2009), "Fuzzy data modeling based on XML schema", *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC 2009)*, *Honolulu*, *Hawaii*, *USA*, pp. 1563-1567.

Appendices

Appendix A: The BO2GE general normalization procedure

This appendix provides the XQuery code of a general BO2GE procedure that can be used to normalize, putting it into "Garden of Eden" design style, any kind of XSD file. In fact, the procedure in Figure 14 supports the management of all XML Schema constructs except redefine/override and key/keyref-related ones. The base structure of the procedure is the same as for the simplified version proposed in Section 3.2.4, extended with additional code and helping functions designed to deal with additionally considered constructs, including annotations, include/import statements, choice/all/group/attributeGroup constructs and complex type redefinitions involving restrictions and extensions. In particular, for group/attributeGroup structures (which are supposed to be declared as global to be reused and, thus, usually have a name but can also be locally declared without a name in nested environments), missing names are generated by helping functions using two initials of the nested elements-attributes followed by the suffix "Group". For instance, for the group:

```
<xs:group>
   <xs:sequence>
      <xs:element name="customer" type="xs:string"/>
      <xs:element name="billto" type="xs:string"/>
      <xs:element name="shipto" type="xs:string"/>
   </xs:sequence>
</xs:group>
```

the generated name is "cubishGroup".

Notice that this procedure can also be used, as it is, to normalize "Russian Doll", "Salami Slice", or "Venetian Blind" general schemas. Moreover, all the other XX2GE and GE2YY procedures presented in this paper could be extended in a similar vein to work with any kind of XML Schema general constructs.

Appendix B: Queries for testing conformance to design styles of XML schemas

In this appendix, we present the four XQuery test queries, named test_RD.xq (cf. Figure 15), test_SS.xq (cf. Figure 16), test_VB.xq (cf. Figure 17), and test_GE.xq (cf. Figure 18), for testing conformance to the "Russian Doll" style, the "Salami Slice" style, the "Venetian Blind" style, and the "Garden of Eden" style, respectively.

As assumed at the beginning of Section 2, also these test queries are supposed to work with an XSD file storing the schema of any XML file with a tree structure, with elements as inner nodes and values of elements or attributes having a predefined XMLSchema type or a simpleType as leaves, that is XML files usually considered in data management (i.e., like our "employee" samples provided in Sec. 2). Also the functioning of all these test queries has been tested with the Altova XMLSpy 2019 tool.

In the following, we provide their semantics with reference to the truth values of the $test* variables used in the XQuery code.

The semantics of the test_RD.xq testing query (shown in Figure 15) is as follows:

- $test1 is true iff there is only one global (outer) definition, which is an element definition (root element) and there are no definitions with a ref attribute;
- $test2 is true iff all local (inner) element definitions have a name attribute and either have a predefined type or contain a simpleType or complexType definition;
- $test3 is true iff all local (inner) attribute definitions have a name attribute and either have a predefined type or contain a simpleType definition.

The semantics of the test_SS.xq testing query (shown in Figure 16) is as follows:

- $test1 is true iff all global (outer) definitions are either element definitions (with a name attribute and either having a predefined type or containing a simpleType or complexType definition) or attribute definitions (with a name attribute and either having a predefined type or containing a simpleType definition);
- $test2 is true iff only one global element definition (root element) is not referenced in any other element definition;
- $test3 is true iff all local (inner) element definitions do not have a name attribute and have a ref attribute equal to the name of a globally defined element;
- $test4 is true iff all local (inner) attribute definitions do not have a name attribute and have a ref attribute equal to the name of a globally defined attribute.

The semantics of the test_VB.xq testing query (shown in Figure 17) is as follows:

- $test1 is true iff there is only one global (outer) element definition (root element), all the other global definitions are complexType or simpleType definitions only and there are no definitions with a ref attribute anywhere;
- $test2 is true iff all local (inner) element definitions have a name attribute and have a type attribute equal to a predefined type or equal the name of a globally defined complexType or simpleType and all local (inner) attribute definitions have a name attribute and have a type attribute equal to a predefined type or equal the name of a globally defined simpleType;
- $test3 is true iff all the complexType definitions are global and have a name which is referenced at type of an element and all the simpleType definitions are global and have a name which is referenced at type of an element or an attribute.

The semantics of the test_GE.xq testing query (shown in Figure 18) is as follows:

- $test1 is true iff there global (outer) definitions include element definitions with a name and a type (which can be either a predefined type or the name of a global complexType or simpleType), attribute definitions with a name and a type (which can be either a predefined type or the name of a global simpleType), a complexType definition (with a name that must be referenced as type of an element) or simpleType definition (with a name that must be referenced as type of an element or an attribute);
- $test2 is true iff only one global element definition (root element) is not referenced in any other element definition;
- $test3 is true iff all local (inner) element definitions do not have a name but a ref attribute referencing the name of globally defined element;
- $test4 is true iff all local (inner) attribute definitions do not have a name but a ref attribute referencing the name of globally defined attribute;
- $test5 is true if all complexType definitions are global and have a name attribute whose value is referenced as type in an element definition and all simpleType definitions are global and have a name attribute whose value is referenced as type in an element or attribute definition.

Notice that these XQuery test queries can also be combined together in a style-detect query, named test_style.xq and producing an answer RD, SS, VB, GE or BO for any input XSD file, as shown in Figure 19.

Appendix C: Query for resolving naming conflicts

In this appendix, we present an XQuery program, named solveNameConflicts.xq (cf. Figure 20), for solving the conflict of duplicate names. We just provide the listing of a demonstrative implementation, without the optimizations suggested in Sec. 3.1.

The functioning of solveNameConflicts.xq is based on three passes over the input XML schema. In the first pass, homonym elements and homonym attributes are detected and a supporting data structure is built and stored in the $renameList variable via the setRenameList() function. Its structure is as follows:

```
<renameList>
  <renameItem>
    <newName> ...new name with suffix _n added to
              renamed element/attribute... </newName>
    <path> ...full path of the renamed element/attribute
           in the source XML schema... </path>
  </renameItem>
  <renameItem>
    <newName> ...new name with suffix _n added to
              renamed element/attribute... </newName>
    <path> ...full path of the renamed element/attribute
           in the source XML schema... </path>
  </renameItem>
  ...
</renameList>
```

In the second pass, the input XML schema is basically rewritten by the recursive function change(): every component of the input schema is simply copied to the output schema, but when an element or attribute is found such as its path equals the `<path/>` value stored in a `<renameItem/>` of the `<renameList/>` data structure, its name is changed to the corresponding `<newName/>` value during the copy.
In the third pass, comments with provenance links are finally generated for each renamed element or attribute.

Notice that, although it is claimed to be fully compliant with XQuery 3.1, Altova XMLSpy 2019 does not support the `fn:path()` function and the full syntax of the `typeswitch` construct. A slightly different version of the solveNameConflicts.xq program, compatible with Altova XMLSpy 2019, can be found in the GitHub repository.