High-level Operations for Changing Temporal Schema, Conventional Schema and Annotations, in the τXSchema Framework

Zouhaier Brahmia, Fabio Grandi, Barbara Oliboni, Rafik Bouaziz

January 15, 2014

TR-96

A TIMECENTER Technical Report

Title	High-level Operations for Changing Temporal Schema, Conventional Schema and Annotations, in the τXSchema Framework
Author(s)	Zouhaier Brahmia, Fabio Grandi, Barbara Oliboni, Rafik Bouaziz
Publication History	January 2014. A TIMECENTER Technical Report.

TIMECENTER Participants

Michael H. Böhlen, University of Zurich, Switzerland; Curtis E. Dyreson, Utah State University, USA; Fabio Grandi, University of Bologna, Italy; Christian S. Jensen (codirector), Aarhus University, Denmark; Vijay Khatri, Indiana University, USA; Gerhard Knolmayer, University of Berne, Switzerland; Carme Martín, Technical University of Catalonia, Spain; Thomas Myrach, University of Berne, Switzerland; Mario A. Nascimento, University of Alberta, Canada; Sudha Ram, University of Arizona, USA; John F. Roddick, Flinders University, Australia; Keun H. Ryu, Chungbuk National University, Korea; Simonas Šaltenis, Aalborg University, Denmark; Dennis Shasha, New York University, USA; Richard T. Snodgrass (codirector), University of Arizona, USA; Paolo Terenziani, University of Piemonte Orientale "Amedeo Avogadro," Alessandria, Italy; Stephen W. Thomas, Queen's University, Canada; Kristian Torp, Aalborg University, Denmark; Vassilis Tsotras, University of California, Riverside, USA; Fusheng Wang, Emory University, USA; Jef Wijsen, University of Mons-Hainaut, Belgium; and Carlo Zaniolo, University of California, Los Angeles, USA

For additional information, see The TIMECENTER Homepage:
URL: <http://www.cs.aau.dk/TimeCenter>

The TIMECENTER icon on the cover combines two "arrows." These "arrows" are letters in the so-called *Rune* alphabet used one millennium ago by the Vikings, as well as by their precedessors and successors. The Rune alphabet (second phase) has 16 letters, all of which have angular shapes and lack horizontal lines because the primary storage medium was wood. Runes may also be found on jewelry, tools, and weapons and were perceived by many as having magic, hidden powers.

The two Rune arrows in the icon denote "T" and "C," respectively.

High-level Operations for Changing Temporal Schema, Conventional Schema and Annotations, in the τXSchema Framework

Zouhaier Brahmia
University of Sfax, Tunisia − email: zouhaier.brahmia@fsegs.rnu.tn
Fabio Grandi
University of Bologna, Italy − email: fabio.grandi@unibo.it
Barbara Oliboni
University of Verona, Italy − email: barbara.oliboni@univr.it
Rafik Bouaziz
University of Sfax, Tunisia − email: raf.bouaziz@fsegs.rnu.tn

Abstract

τXSchema [1] is a framework for constructing and validating time-varying XML documents through the use of a temporal schema. This latter ties together a conventional schema (i.e., a standard XML Schema document) and its corresponding logical and physical annotations, which are stored in an annotation document. Conventional schema and logical and physical annotations undergo changes to respond to changes in user requirements or in the real-world. Consequently, the corresponding temporal schema is also evolving over time. In this report, we study operations which help designers for changing such schema components. Indeed, we propose three sets of high-level operations for changing temporal schema, conventional schema, and annotations. These high-level operations are based on the low-level operations proposed in [2,3], [4,5] and [18]. They are also consistency preserving and more user-friendly than the low-level operations. Besides, we have divided the proposed operations into basic high-level operations (i.e., high-level operations that cannot be defined by using other basic high-level operations) and complex ones.

Keywords: τXSchema, Schema versioning, XML, XML Schema, Temporal database

1. Introduction

τXSchema [1] is a framework (i.e., a language, a repository of XML documents and XSD files, and a suite of tools) in which an XML schema designer could create and validate temporal XML documents thanks to the use of a temporal schema. This latter is also used when manipulating or querying temporal XML data that are stored in temporal documents. The temporal schema associates a conventional schema (i.e., a standard XML Schema document) to an annotation document which contains logical and physical annotations. The annotations allow the XML schema designer to specify which portion(s) of an XML document can vary over time, how the document can change, and where timestamps should be placed.

systems must allow designers to change these schema components, for example, through an easy-to-use tool which provides user-friendly high-level schema change operations.

In our previous works [2,3], [4,5], and [18], we proposed three complete and sound sets of low-level operations, for changing conventional schema, logical and physical annotations, and temporal schema, respectively. Obviously, these operations are not designated to be used directly by τXSchema designers, since they are too primitive and not very user-friendly. They allow us to define high-level and more user-friendly

operations. A high-level operation is a valid sequence of low-level operations, that correspond to frequent schema evolution needs and allows expressing complex changes in a more compact way [14].

To help τXSchema designers and to make our approach more useful, we propose in this report two sets of high-level operations for changing conventional schema and annotations. Besides, since the temporal schema must be updated after changing the conventional schema and/or the corresponding annotations, and in order to complete the picture, we propose also a set of high-level operations that provides an interface to designers for consistently updating temporal schema.

τXSchema designers could use these high-level operations to make any change on τXSchema schema, by composing them into valid sequences and collectively executing them on the considered component (the temporal schema, the conventional schema, the logical annotations or the physical annotations) within the same transaction. Here, a transaction consists of a sequence of valid schema change operations that would be carried out on the τXSchema framework and that would be either all successfully completed or all cancelled.

The proposed high-level operations are based on the low-level operations (i.e., primitives) already defined in [2,3], [4,5], and [18]. Since each low-level operation is consistency preserving (i.e., each operation applied to a consistent τXSchema schema component produces a consistent τXSchema schema component) and each high-level operation is defined using a sequence of low-level operations, the proposed high-level operations are consequently consistency preserving. On the other hand, the proposed operations are more user-friendly since they consider complex components and sub-documents rather than single elements.

We have classified the new operations into two categories: basic high-level operations (i.e., high-level operations that cannot be defined by using other basic high-level operations) and complex high-level operations (i.e., high-level operations that are defined by using other basic and/or complex high-level operations).

The rest of this report is organized as follows. Section 2 describes the proposed set of high-level operations for changing the temporal schema in the τXSchema framework. Section 3 presents the proposed set of high-level operations for changing conventional schema. Section 4 gives the proposed set of high-level operations for changing logical and physical annotations. In Section 5, we discuss related work and show the contributions of our present work. Section 6 concludes the report with an outline of future steps in our on-going work dealing with versioning aspects in the τXSchema framework.

2. High-level Operations for Changing the Temporal Schema

In [18], we defined four change primitives that act on the temporal schema: CreateTemporalSchema, RemoveTemporalSchema, AddSlice, and RemoveSlice. Based on these primitives, we proposed four high-level operations for changing temporal schema in the τXSchema framework. These operations are as follows:

$Op^{TS}01$: DefineTemporalSchema(TS.xml,
sourceFirstVersionCS, targetFirstVersionCS,
sourceFirstVersionAD, targetFirstVersionAD)

It creates a new temporal schema document (TS.xml) that includes a first version of a conventional schema (located at targetFirstVersionCS and whose contents are obtained from the sourceFirstVersionCS) and a first version of the corresponding annotation document (located at targetFirstVersionAD and whose contents are obtained from the sourceFirstVersionAD).

Here, the **sourceFirstVersionCS** (**sourceFirstVersionAD**, respectively) parameter could be:

1) The keyword **empty**; in this case the resource pointed by **targetFirstVersionCS** (**targetFirstVersionAD**, respectively) is initialized to an empty **conventional schema** (**annotation document**, respectively).

2) The keyword **current**; in this case the resource pointed by **targetFirstVersionCS** (**targetFirstVersionAD**, respectively) is initialized with a copy of the current **conventional schema** (**annotation document**, respectively) resource, whose location is found in the **TS.xml** temporal schema file by choosing the `<slice/>` subelement with the maximum value of **begin** in the `<sliceSequence/>` element of the `<conventionalSchema/>` (`<annotationSet/>`, respectively) container. Notice that this is the normal case after the creation of the *first* temporal schema version (obviously with a first conventional schema version and a first annotation document version).

3) A specified file name (URL): in this case, a copy of the specified resource is renamed as **targetFirstVersionCS** (**targetFirstVersionAD**, respectively) and used as the new location (e.g., this case is used to create a new conventional schema (annotation document, respectively) version from an already existing XML schema (XML document, respectively) file, which could be quite common when creating the first version but can be used also later for reuse purpose and/or integrating independently developed XML schemata (XML documents, respectively) into a τXSchema framework).

The **targetFirstVersionCS** (**targetFirstVersionAD**, respectively) parameter corresponds to the location of the new **conventional schema** (**annotation document**, respectively) version; it must not correspond to the URL of any already existing XML schema (XML document, respectively) file/resource.

This operation is mapped onto the following sequence of primitives:

(i) CreateTemporalSchema(TS.xml)

(ii) AddSlice(TS.xml, conventionalSchema, sourceFirstVersionCS, targetFirstVersionCS)

(iii) AddSlice(TS.xml, annotationSet, sourceFirstVersionAD, targetFirstVersionAD)

Example: Suppose that the designer would like to define a new temporal schema for vehicles of a company, based on an XML Schema file "Vehicle.xsd" and an XML document "VehicleAnnotations.xml" that includes temporal and physical annotations associated to "Vehicle.xsd". To do this, he/she calls the DefineTemporalSchema operation as follows:

DefineTemporalSchema("VehicleTemporalSchema.xml", "Vehicle.xsd", "Vehicle_V1.xsd",
"VehicleAnnotations.xml", "VehicleAnnotations_V1.xml")

$Op^{TS}02$: UpdateTemporalSchema(TS.xml, sourceNewVersionCS, targetNewVersionCS, sourceNewVersionAD, targetNewVersionAD)

It updates a temporal schema by including a new conventional schema version, sourceNewVersionCS, or a new annotation document version, sourceNewVersionAD (only one of these two parameters can be omitted).

This operation is mapped onto the following list of primitives:

(i) If (sourceNewVersionCS is not null) then
AddSlice(TS.xml, conventionalSchema, sourceNewVersionCS, targetNewVersionCS)
(ii) If (sourceNewVersionAD is not null) then
AddSlice(TS.xml, annotationSet, sourceNewVersionAD, targetNewVersionAD)

$Op^{TS}03$: DropTemporalSchema(TS.xml)

It allows the designer to drop a temporal schema, if necessary.

This operation is mapped onto the following list of operations:

```
for each sourceLocation := <slice/> element
        in <conventionalSchema/> container of "TS.xml" do:
    RemoveSlice(TS.xml, conventionalSchema, sourceLocation)
for each sourceLocation := <slice/> element
        in <annotationSet/> container of "TS.xml" do:
    RemoveSlice(TS.xml, annotationSet, sourceLocation)
RemoveTemporalSchema(TS.xml)
```

Otherwise, this operation could also be mapped onto the following list of operations:

```
for sourceLocation in doc("TS.xml")//conventionalSchema/slice
    RemoveSlice(TS.xml, conventionalSchema, sourceLocation)
for sourceLocation in doc("TS.xml")//annotationSet/slice
    RemoveSlice(TS.xml, annotationSet, sourceLocation)
RemoveTemporalSchema(TS.xml)
```

Notice that:

1) The **RemoveSlice(TS.xml, conventionalSchema, sourceLocation)** primitive removes from the `<conventionalSchema/>` container the `<slice/>` element having its attribute **location** set to the value "sourceLocation".
2) The **RemoveSlice(TS.xml, annotationSet, sourceLocation)** primitive removes from the `<annotationSet/>` container the `<slice/>` element having its attribute **location** set to the value "sourceLocation".
3) The **RemoveTemporalSchema(TS.xml)** primitive removes, from the disk, the empty "TS.xml" file.

$Op^{TS}04$: RenameTemporalSchema(TS.xml, newName)

It changes the name of an existing temporal schema ("TS.xml") to "newName".

3. High-level Operations for Changing Conventional Schema

We have defined thirty-nine basic high-level operations. They deal with XML Schema elements, attributes, and constraints. We have also defined ten complex high-level operations which deal with entire conventional schema and portions of conventional schema (or subschema).

3.1. Basic High-Level Operations

3.1.1. Basic High-Level Operations dealing with Elements

$Op^{CS}01$: MoveElement(CS.xsd, elementPath, position, targetElementPath)

It moves an existing element (located at elementPath) to a new position (i.e., after or before) with regard to another element (located at targetElementPath), in the same conventional schema "CS.xsd". This operation must update:

- all other components of this conventional schema that are using (or referring to) this element (e.g., <key>, <unique>, and <keyref> components);
- all <item> and <stamp> components, in the current version of the annotation document corresponding to this conventional schema, that are referring to this element.

Example: Suppose that the designer decides to move an element <manufacturer> from an element <vehicle> to another element <vehicle-model> (see Figures 1 and 2), since all vehicles having the same model are built by the same manufacturer. To do this, he/she calls the MoveElement operation as follows:

MoveElement(CS.xsd, " //xsd:element[@name='manufacturer']", before, "//xsd:element[@name='model-name']")

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
<xsd:element name = "vehicles">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element ref = "vehicle" maxOccurs = "unbounded"/>
    </xsd:sequence>
   </xsd:complexType>
 </xsd:element>
 <xsd:element name = "vehicle">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element name = "registration-number" type = "xsd:string"/>
     <xsd:element name = "model" type = "vehicle-model"/>
     <xsd:element name = "color" type = "xsd:string"/>
     <xsd:element name = "manufacturer" type = "xsd:string"/>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
 <xsd:element name = "vehicle-model">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element name = "model-name" type = "xsd:string"/>
     <xsd:element name = "year" type = "xsd:gYear"/>
     <xsd:element name = "price" type = "xsd:double"/>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Figure 1: Conventional schema for vehicles before change.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
 <xsd:element name = "vehicles">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element ref = "vehicle" maxOccurs = "unbounded"/>
    </xsd:sequence>
   </xsd:complexType>
```

```
 </xsd:element>
 <xsd:element name = "vehicle">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element name = "registration-number" type = "xsd:string"/>
     <xsd:element name = "model" type = "vehicle-model"/>
     <xsd:element name = "color" type = "xsd:string"/>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
 <xsd:element name = "vehicle-model">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element name = "manufacturer" type = "xsd:string"/>
     <xsd:element name = "model-name" type = "xsd:string"/>
     <xsd:element name = "year" type = "xsd:gYear"/>
     <xsd:element name = "price" type = "xsd:double"/>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Figure 2: Conventional schema for vehicles after change.

Op^{CS}02: CopyElement(CS.xsd, elementPath, position, targetElementPath)

It copies an existing element (located at elementPath) into a new position (i.e., after or before) with regard to another element (located at targetElementPath), in the same conventional schema "CS.xsd".

Example: Let us resume the example of Figure 2. Suppose that the designer would like to copy the element `<price>` (which is a sub-element of the `<vehicle-model>` element) in the element `<vehicle>`, after the `<color>` element. He/She calls the CopyElement operation as follows:

```
CopyElement(CS.xsd, "//xsd:element[@name='price']", after, "//xsd:element[@name='color']")
```

The element `<vehicle>` becomes as follows:

```
  <xsd:element name = "vehicle">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="registration-number" type ="xsd:string"/>
     <xsd:element name="model" type="vehicle-model"/>
     <xsd:element name="color" type="xsd:string"/>
     <xsd:element name="price" type="xsd:double"/>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
```

Op^{CS}03: RenameElement(CS.xsd, elementPath, newNameElement)

It changes the name of an existing element (located at elementPath) with a new name (newNameElement), in the conventional schema "CS.xsd". This operation must update:

- all other components of this conventional schema that are using (or referring to) this element

(e.g., <element> with ref attribute, <key>, <unique>, and <keyref> components) by replacing the old name with the new one;
- all <item> and <stamp> components, in the current version of the annotation document corresponding to this conventional schema, that are referring to this element.

Example: Let us resume the example of Figure 2. Suppose that the designer would like to rename the element <price> (which is a sub-element of the <vehicle> element) to <vehicle-price>. He/She calls the RenameElement operation as follows:

```
RenameElement(CS.xsd, "//xsd:element[@name='price']", "vehicle-price")
```

Op^{CS}04: ReplaceElementWithNewElement(CS.xsd, elementPath, newElementId, newElementName, newElementType, newElementDefault, newElementFixed, newElementAbstract, newElementFinal, newElementRef, newElementMinOccurs, newElementMaxOccurs, newElementBlock, newElementForm, newElementNillable, newElementSubstitutionGroup)

It replaces an existing simple element (located at elementPath) with a new simple element (having the following properties: newElement**Id**, newElement**Name**, newElement**Type**, newElement**Default**, newElement**Fixed**, newElement**Abstract**, newElement**Final**, newElement**Ref**, newElement**MinOccurs**, newElement**MaxOccurs**, newElement**Block**, newElement**Form**, newElement**Nillable**, and newElement**SubstitutionGroup**), in the conventional schema "CS.xsd". This operation must update:

- all other components of this conventional schema that are using (or referring to) the replaced element (e.g., <element> with ref attribute, <key>, <unique>, and <keyref> components);
- all <item> and <stamp> components, in the current version of the annotation document corresponding to this conventional schema, that are referring to the replaced element.

Example: Let us resume the example of Figure 2. Suppose that the designer would like to replace the element <color> (which is a sub-element of the <vehicle> element) with a new element <vehicle-power>. He/She calls the ReplaceElementWithNewElement operation as follows:

```
ReplaceElementWithNewElement(CS.xsd, "//xsd:element[@name='color']", , "vehicle-power",
        "xsd:double", , , , , , 1, , , , , )
```

Op^{CS}05: TransformSubElementToAttribute(CS.xsd, subElementPath, newAttributeId, newAttributeType, newAttributeDefault, newAttributeFixed, newAttributeUse, newAttributeForm, newAttributeRef)

It transforms a simple sub-element (located at subElementPath) into an attribute (with possibly one or more of the following properties: newAttribute**Id**, newAttribute**Type**, newAttribute**Default**, newAttribute**Fixed**, newAttribute**Use**, newAttribute**Form**, or newAttribute**Ref**) *of its parent element*, in the conventional schema "CS.xsd". This operation could not be performed if the sub-element to be transformed does not have a simple type. This operation must update:

- all other components of this conventional schema that are using (or referring to) the transformed element (e.g., <element> with ref attribute, <key>, <unique>, and <keyref>

components);

- all <item> and <stamp> components, in the current version of the annotation document corresponding to this conventional schema, that are referring to the transformed element.

Example: Let us resume the example of Figure 2. Suppose that the designer would like to transform the sub-element <registration-number> of the complex element <vehicle> into an attribute "registration-number" of the same element. He/She calls the TransformSubElementToAttribute operation as follows:

```
TransformSubElementToAttribute(CS.xsd, "//xsd:element[@name='registration-number']", ,
        "xsd:string", , , "required", , )
```

$Op^{CS}06$: TransformElementToAttribute(CS.xsd, sourceElementPath, targetElementPath, newAttributeId, newAttributeType, newAttributeDefault, newAttributeFixed, newAttributeUse, newAttributeForm, newAttributeRef)

It transforms a simple element (located at sourceElementPath) into an attribute (with possibly one or more of the following properties: newAttribute**Id**, newAttribute**Type**, newAttribute**Default**, newAttribute**Fixed**, newAttribute**Use**, newAttribute**Form**, or newAttribute**Ref**) *of another element* (located at targetElementPath), in the conventional schema "CS.xsd". This operation could not be performed if the element to be transformed does not have a simple type and if the element that will include the new attribute is not already a complex element. This operation must update:

- all other components of this conventional schema that are using (or referring to) the transformed element (e.g., <element> with ref attribute, <key>, <unique>, and <keyref> components);
- all <item> and <stamp> components, in the current version of the annotation document corresponding to this conventional schema, that are referring to the transformed element.

$Op^{CS}07$: ExchangeElements(CS.xsd, element1Path, element2Path)

It exchanges the element located at element1Path with the element located at element2Path, in the conventional schema "CS.xsd". Notice here that the two elements are not necessarily sub-elements of the same complex element.

$Op^{CS}08$: SplitElementIntoElements(CS.xsd, elementPath)

It splits a non-empty complex element (located at elementPath) that contains only simple sub-elements, into so many elements as sub-elements. In other words, we could say that this operation replaces this complex element by its sub-elements.

Notice. A complex element contains other elements and/or attributes. There are four kinds of a complex element: (i) empty element, (ii) element that contains only other elements, (iii) element that contains only text, and (iv) element that contains both text and other elements. In this operation, we deal only with the second kind.

Moreover, this operation must update:

- all other components of this conventional schema that are using (or referring to) the split element (e.g., <key>, <unique>, and <keyref> components);

- all <stamp> components, in the current version of the annotation document corresponding to this conventional schema, that are referring to the split element.

Example: Suppose that the designer would like to split a complex element <AuthorContact> which contains five simple sub-elements (<Address>, <Phone>, <Cell>, <Fax>, and <Email>) into five new elements (as shown in Figure 3). He/She calls the SplitElementIntoElements operation as follows:

```
SplitElementIntoElements(CS.xsd, "//xsd:element[@name='Author-contact']")
```

Before applying *SplitElementIntoElements* operation	**After applying *SplitElementIntoElements* operation**
<xsd:element name="AuthorContact"> <xsd:complexType> <xsd:sequence> <xsd:element name="Address" type="xsd:string"/> <xsd:element name="Phone" type="xsd:string"/> <xsd:element name="Cell" type="xsd:string"/> <xsd:element name="Fax" type="xsd:string"/> <xsd:element name="Email" type="xsd:string"/> </xsd:sequence> </xsd:complexType> </xsd:element>	<xsd:element name="Address" type="xsd:string"/> <xsd:element name="Phone" type="xsd:string"/> <xsd:element name="Cell" type="xsd:string"/> <xsd:element name="Fax" type="xsd:string"/> <xsd:element name="Email" type="xsd:string"/>

Figure 3: Effect of the SplitElementIntoElements operation.

Op^{CS}09: AddSubElement(CS.xsd, parentElementPath, precedingElementPath, newSubElementId, newSubElementName, newSubElementType, newSubElementDefault, newSubElementFixed, newSubElementAbstract, newSubElementFinal, newSubElementRef, newSubElementMinOccurs, newSubElementMaxOccurs, newSubElementBlock, newSubElementForm, newSubElementNillable, newSubElementSubstitutionGroup)

It adds a new simple sub-element (having the following properties: newSubElement**Id**, newSubElement**Name**, newSubElement**Type**, newSubElement**Default**, newSubElement**Fixed**, newSubElement**Abstract**, newSubElement**Final**, newSubElement**Ref**, newSubElement**MinOccurs**, newSubElement**MaxOccurs**, newSubElement**Block**, newSubElement**Form**, newSubElement**Nillable**, and newSubElement**SubstitutionGroup**) to an existing element (located at parentElementPath), possible after the element located at precedingElementPath (if this parameter does not contain any value, the new added sub-element is considered as the first one).

This operation can be effected by means of an AddElement primitive [2,3] only if the target element has already a "sequence" structure that, otherwise, must be created before the AddElement primitive can be used. For example, the AddSubElement operation applied to:

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

must transform it, for instance, into:

```
<xsd:element name="model-name">
    <xsd:complexType mixed="true">
        <xsd:sequence/>
    </xsd:complexType>
</xsd:element>
```

before the AddElement primitive can be used to add the new sub-element to the "sequence".

This operation is mapped onto the following list of primitives:

If (parentElementPath refers to an element having a "sequence", "choice", or "all" structure) then

(1) AddElement(CS.xsd, structure, parentComponentPath, precedingComponentPath, id, name, type, default, fixed, abstract, final, ref, minOccurs, maxOccurs, block, form, nillable, substitutionGroup)

Else

(1) DeleteAttribute(CS.xsd, element, 'parentElementPath/@type')

(2) AddComplexType(CS.xsd, element, parentComponentPath, precedingComponentPath, , , , , , "true",)

(3) AddSequence(CS.xsd, complexType, parentComponentPath, precedingComponentPath, , ,)

(4) AddElement(CS.xsd, sequence, parentElementPath, precedingElementPath, id, name, type, default, fixed, abstract, final, ref, minOccurs, maxOccurs, block, form, nillable, substitutionGroup)

End If

$Op^{CS}10$: DropElement(CS.xsd, elementPath)

It removes an element located at "elementPath" from the conventional schema "CS.xsd". Such an operation must fail with an error message when the element to be removed is used (or referred) by other components in the conventional schema "CS.xsd". Otherwise, this operation must remove all `<item>` and `<stamp>` components, in the annotation document corresponding to this conventional schema, that are referring to the removed element.

$Op^{CS}11$: CollapseSubElements(CS.xsd, firstSubElementPath, lastSubElementPath, orderIndicator)

It collapses successive subelements (such as the first one is located at "firstSubElementPath" and the last one is located at "lastSubElementPath") belonging to a same parent element, under an order indicator (choice, sequence, or all), in the conventional schema "CS.xsd".

This operation must update:

- all other components of this conventional schema that are using (or referring to) the collapsed elements (e.g., `<key>`, `<unique>`, and `<keyref>` components);
- all `<stamp>` components, in the annotation document corresponding to this conventional schema, that are referring to the collapsed elements.

Example: Suppose that the designer would like to collapse two subelements, `e2` and `e3`, under a `choice` indicator, as shown in Figure 4. To do this, he/she calls the CollapseSubElements operation as

follows:

```
CollapseSubElements(CS.xsd, "//xsd:element[@name='e2']", "//xsd:element[@name='e3']", choice)
```

As regards this operation, we were inspired by the high-level primitive “collapse_substruct” presented in [14].

Before applying *CollapseSubElements* operation	**After applying *CollapseSubElements* operation**
<xsd:sequence> <xsd:element name=”e1”/> <xsd:element name=”e2”/> <xsd:element name=”e3”/> </xsd:sequence>	<xsd:sequence> <xsd:element name=”e1”/> **<xsd:choice>** **<xsd:element name=”e2”/>** **<xsd:element name=”e3”/>** **</xsd:choice>** </xsd:sequence>

Figure 4: Effect of the CollapseSubElements operation.

3.1.2. Basic High-Level Operations dealing with Attributes of Elements

$Op^{CS}12$: AddAttribute(CS.xsd, targetElementPath, newAttributeId, newAttributeName, newAttributeType, newAttributeDefault, newAttributeFixed, newAttributeUse, newAttributeForm, newAttributeRef)

It adds an attribute (having the following properties: newAttribute**Id**, newAttribute**Name** (mandatory), newAttribute**Type** (mandatory), newAttribute**Default**, newAttribute**Fixed**, newAttribute**Use**, newAttribute**Form**, and newAttribute**Ref**) to an existing element (located at targetElementPath) that could have (or not) a complex type. In case the target element has not a complex type, this operation has to change the definition of this element before adding the new attribute.

Such an operation must fail with an error message when the target element has a complex type and already has got an attribute with the same name of the new attribute.

Example: Let us resume the example of Figure 2. Suppose that the designer should add a new attribute “country” (having a string type) to the simple element <manufacturer/> defined as follows:

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

To do this, he/she calls the AddAttribute operation as follows:

```
AddAttribute(CS.xsd, "//xsd:element[@name='manufacturer']", ,"country","xsd:string", , , , , )
```

In order to add the new attribute, this operation has to change the definition of this element to

```
<complexType name="manufacturer">
   <simpleContent>
       <extension base="xsd:string">
            <attribute name="country" type="xsd:string"/>
       </extension>
    </simpleContent>
</complexType>
```

Notice that since high-level operations should rather make reference to the XML document structure

defined by the Schema rather than to punctual definitions in the XML Schema, the AddAttribute operation should be applicable, for instance, to an element:

```
<manufacturer>...</manufacturer>
```

regardless if it has been defined in the Schema as:

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

or as:

```
<xsd:simpleType name="manufacturer-type">
    <xsd:restriction base="xsd:string"/>
</xsd:simpleType>
<xsd:element name="manufacturer" type="manufacturer-type"/>
```

or as:

```
<xsd:element name="manufacturer" >
    <xsd:simpleType name="manufacturer-type">
        <xsd:restriction base="xsd:string"/>
    </xsd:simpleType>
</xsd:element>
```

or other ways.

$Op^{CS}13$: MoveAttribute(CS.xsd, attributePath, targetElementPath)

It moves an existing attribute (located at attributePath) from an element to another element (located at targetElementPath) that could have (or not) a complex type. In case the target element has not a complex type, this operation has to change the definition of the target element before moving the new attribute.

Such an operation must fail with an error message when the target element has a complex type and already has got an attribute with the same name of the new attribute.

Furthermore, this operation must update:

- all other components of this conventional schema that are using (or referring to) this attribute (e.g., <key>, <unique>, and <keyref> components);
- all <stamp> components, in the current version of the annotation document corresponding to this conventional schema, that are referring to this attribute.

Example: Let us resume the example of Figure 2. Suppose that the designer would like to move an attribute "MY" (belonging to the element <vehicle> and having a string type) to the simple element <model-name>. To do this, he/she calls the MoveAttribute operation as follows:

```
MoveAttribute(CS.xsd, "//xsd:element[@name='vehicle']/xsd:attribute[@name='MY']",
                      "//xsd:element[@name='model-name']")
```

Thus, the definition of the element <model-name>:

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

has to be changed to:

```
<complexType name="model-name">
   <simpleContent>
       <extension base="xsd:string">
            <attribute name="MY" type="xsd:string"/>
       </extension>
    </simpleContent>
</complexType>
```

$Op^{CS}14$: CopyAttribute(CS.xsd, attributePath, targetElementPath)

It copies an existing attribute (located at attributePath) of an element into another element (located at targetElementPath) that could have (or not) a complex type. In case the target element has not a complex type, this operation has to change the definition of the target element before copying the new attribute (see examples of $Op^{CS}12$ and $Op^{CS}13$).

Such an operation must fail with an error message when the target element has a complex type and already has got an attribute with the same name of the new attribute.

Example: Suppose that the designer would like to copy an attribute "birthdate" of an element `<student>` into another element `<university-degree>`. To do this, he/she calls the CopyAttribute operation as follows:

```
CopyAttribute(CS.xsd, "//xsd:element[@name='student']/xsd:attribute[@name='birthdate']",
            "//xsd:element[@name='university-degree']")
```

$Op^{CS}15$: RenameAttribute(CS.xsd, attributePath, newAttributeName)

It changes the name of an attribute (located at attributePath) of an element, in the conventional schema "CS.xsd". This operation must update:

- all other components of this conventional schema that are using (or referring to) this attribute (e.g., `<key>`, `<unique>`, and `<keyref>` components);
- all `<stamp>` components, in the current version of the annotation document corresponding to this conventional schema, that are referring to this attribute.

Example: Suppose that the designer would like to rename an attribute "customer-number" of an element `<customer>` to "customer-code". To do this, he/she calls the RenameAttribute operation as follows:

```
RenameAttribute(CS.xsd, "//xsd:element[@name='customer']/xsd:attribute[@name='customer-
            number']", "customer-code")
```

$Op^{CS}16$: TransformAttributeToSubElement(CS.xsd, attributeName, complexElementPath, position, neighborSubElementPath, newSubElementId, newSubElementType, newSubElementDefault, newSubElementFixed, newSubElementAbstract, newSubElementFinal, newSubElementRef, newSubElementMinOccurs, newSubElementMaxOccurs,

newSubElementBlock, newSubElementForm, newSubElementNillable, newSubElementSubstitutionGroup)

It transforms an attribute (having the name "attributeName") of a complex element (located at complexElementPath). In case the complex element has a simple content or a complex content, this operation has to change the definition of this element, so it becomes a complex element with a complex content, including a "sequence" structure which contains only one sub-element corresponding to the transformed attribute. However, in case the complex element has already a "sequence" structure containing some sub-elements, this operation creates a simple element from the corresponding attribute and adds it at a specified position (i.e., after or before) with regard to another neighbor (i.e., predecessor or successor) sub-element (located at neighborSubElementPath) *of the same parent element* (located at complexElementPath), in the same conventional schema "CS.xsd"

Whatever is the case, the new sub-element will have possibly one or more of the following properties: newSubElement**Id**, newSubElement**Type**, newSubElement**Default**, newSubElement**Fixed**, newSubElement**Abstract**, newSubElement**Final**, newSubElement**Ref**, newSubElement**MinOccurs**, newSubElement**MaxOccurs**, newSubElement**Block**, newSubElement**Form**, newSubElement**Nillable**, and newSubElement**SubstitutionGroup**).

This operation must update:

- all other components of this conventional schema that are using (or referring to) the transformed attribute (e.g., `<key>`, `<unique>`, and `<keyref>` components);
- all `<stamp>` components, in the current version of the annotation document corresponding to this conventional schema, that are referring to the transformed attribute.

Example: Let us resume the example of Figure 2 and take into account the example that illustrates the use of the operation $Op^{CS}12$. Suppose that the designer would like to transform the attribute "country" (added to the element `<manufacturer>` in the example of $Op^{CS}12$) to an element `<country>` within the same parent element (as shown in Figure 5). He/She calls the TransformAttributeToSubElement operation as follows:

```
TransformAttributeToSubElement(CS.xsd, "country", "//xsd:element[@name='manufacturer']", , , ,
            "xsd:string", , , , , , , , , , )
```

Before the change	After the change

`<complexType name="manufacturer">`
`<simpleContent>`
`<extension base="xsd:string">`
`<attribute name="country"`
`type="string"/>`
`</extension>`
`</simpleContent>`
`</complexType>`
	`<complexType name="manufacturer">`
	`<sequence>`
	`<element name="country"`
	`type="string"/>`
	`</sequence>`
	`</complexType>`

Figure 5: Effect of the TransformAttributeToSubElement operation.

$Op^{CS}17$: TransformAttributeToElement(CS.xsd, attributeName, complexElementPath, position, neighborElementPath, newElementId, newElementType, newElementDefault, newElementFixed, newElementAbstract, newElementFinal, newElementRef, newElementMinOccurs,

newElementMaxOccurs, newElementBlock, newElementForm, newElementNillable, newElementSubstitutionGroup)

It transforms an attribute (having the name "attributeName") of a complex element (located at complexElementPath) into a new simple element (with possibly one or more of the following properties: newSubElement**Id**, newSubElement**Type**, newSubElement**Default**, newSubElement**Fixed**, newSubElement**Abstract**, newSubElement**Final**, newSubElement**Ref**, newSubElement**MinOccurs**, newSubElement**MaxOccurs**, newSubElement**Block**, newSubElement**Form**, newSubElement**Nillable**, and newSubElement**SubstitutionGroup**) located at a specified position (i.e., after or before) with regard to another neighbor (i.e., predecessor or successor) element (located at neighborElementPath) *that does not belong to the old parent element (located at complexElementPath) of this attribute*. This operation must update:

- all other components of this conventional schema that are using (or referring to) the transformed attribute (e.g., `<key>`, `<unique>`, and `<keyref>` components);
- all `<stamp>` components, in the current version of the annotation document corresponding to this conventional schema, that are referring to the transformed attribute.

Op^{CS}18: ReplaceAttributeWithNewAttribute(CS.xsd, attributeName, targetElementPath, newAttributeId, newAttributeName, newAttributeType, newAttributeDefault, newAttributeFixed, newAttributeUse, newAttributeForm, newAttributeRef)

It replaces an existing attribute (having the name "attributeName") of a complex element (located at targetElementPath) with a new attribute (having the following properties: newAttribute**Id**, newAttribute**Name** (mandatory), newAttribute**Type** (mandatory), newAttribute**Default**, newAttribute**Fixed**, newAttribute**Use**, newAttribute**Form**, and newAttribute**Ref**), in the conventional schema "CS.xsd". Such an operation must fail with an error message when the target element already has got an attribute with the same name of the new attribute. Furthermore, this operation must update:

- all other components of this conventional schema that are using (or referring to) the replaced attribute (e.g., `<key>`, `<unique>`, and `<keyref>` components);
- all `<stamp>` components, in the current version of the annotation document corresponding to this conventional schema, that are referring to the replaced attribute.

Example: Suppose that the designer would like to replace an attribute "birthdate" of an element `<club-member>` with a new attribute "MonthAndYearOfBirth" (having an XSD "gYearMonth" type). To do this, he/she calls the ReplaceAttributeWithNewAttribute operation as follows:

```
ReplaceAttributeWithNewAttribute(CS.xsd, "birthdate", "//xsd:element[@name='club-member']", ,
        "MonthAndYearOfBirth", "xsd:gYearMonth", , , , , )
```

Op^{CS}19: SplitAttributeIntoAttributes(CS.xsd, elementPath, attributeName, newAttributeNameTypes)

It splits, in the conventional schema "CS.xsd", an attribute (having the name "attributeName") of an element (located at elementPath) into two or more attributes (related to the same element), according to the number of pairs (newAttributeName, newAttributeType) that are provided by the designer in the fourth parameter "newAttributeNameTypes" (which is of string type) and are separated by semicolons.

Moreover, this operation must update:

- all other components of this conventional schema that are using (or referring to) the split attribute (e.g., <key>, <unique>, and <keyref> components);
- all <stamp> components, in the current version of the annotation document corresponding to this conventional schema, that are referring to the split attribute.

Example: Suppose that a designer would like to split an attribute "address" of an element <customer> into five attributes: "number", "street", "city", "zip code", and "country" (as shown in Figure 6). To do this, he/she calls the SplitAttributeIntoAttributes operation as follows:

```
SplitAttributeIntoAttributes(CS.xsd, "//xsd:element[@name='customer']", "address", "(number,
        xsd:positiveInteger);(street, xsd:string);(city, xsd:string);(zip, xsd:positiveInteger);(country,
        xsd:string)")
```

Before the change

```
<xsd:complexType name="customer">
 <xsd:sequence>
  <xsd:element name="name"
        type="xsd:string"/>
  <xsd:element name="turnover"
        type="xsd:double"/>
 </xsd:sequence>
 <xsd:attribute name="address"
        type="xsd:string"/>
</xsd:complexType>
```

After the change

```
<xsd:complexType name="customer">
 <xsd:sequence>
  <xsd:element name="name"
     type="xsd:string"/>
  <xsd:element name="turnover"
     type="xsd:double"/>
  <xsd:element name="number"
     type="xsd:positiveInteger"/>
  <xsd:element name="street"
     type="xsd:string"/>
  <xsd:element name="zip"
     type="xsd:positiveInteger"/>
  <xsd:element name="country"
     type="xsd:string"/>
 </xsd:sequence>
</xsd:complexType>
```

Figure 6: Effects of the SplitAttributeIntoAttributes operation.

$Op^{CS}20$: RemoveAttribute(CS.xsd, elementPath, attributeName)

It removes an attribute (having the name "attributeName") from an element (located at "elementPath"), in the conventional schema "CS.xsd".

Such an operation must fail with an error message when the attribute to be removed is used (or referred) by other components in the same conventional schema "CS.xsd". Otherwise, this operation must update remove all <stamp> components, in the annotation document corresponding to this conventional schema, that are referring to the removed attribute.

3.1.3. Basic High-Level Operations dealing with Conventional Schema Constraints

XML Schema provides four types of constraints:

1. Datatype restrictions;

2. Identity constraints;
3. Referential Integrity constraints;
4. Cardinality constraints.

In the following, we propose high-level operations for changing each type of conventional schema constraints.

3.1.3.1. Basic High-Level Operations dealing with Datatype Restrictions

Datatype restrictions allow defining constraints on the structure and content of elements, or on the content of attributes. A datatype restriction is defined using the XML Schema `simpleType` component in order to derive a new simple type from an existing one (built-in or derived)

$Op^{CS}21$: DerivingNewSimpleTypeByRestriction(CS.xsd, targetElementPath, position, newSimpleTypeName, baseType, nameFirstFacet, valueFirstFacet, fixedFirstFacet, nameSecondFacet, valueSecondFacet, fixedSecondFacet)

It creates a new simple type (having the name "simpleTypeName") by restricting an existing simple type (having the name "baseType") through the use of a first facet (having the following properties: nameFirstFacet, valueFirstFacet, and fixedFirstFacet (optional) which takes "true" or "false") and possibly a second one (having the following properties: nameSecondFacet, valueSecondFacet, and fixedSecondFacet), that constrain the existing simple type's range of values. This operation adds the new simple type at a specified position (i.e., after or before) with regard to another target element (located at targetElementPath).

Example: Suppose that the designer would like to specify a restriction on the type "float", in order to define a new data type "examination-mark-type" whose range of values is between 0 and 20 (inclusive), as shown in Figure 7. To do this, he/she calls the DerivingNewSimpleTypeByRestriction operation as follows:

```
DerivingNewSimpleTypeByRestriction(CS.xsd, "//xsd:complexType[@name='student-type']", after,
        "examination-mark-type", "xsd:float", "minInclusive", "0", , "maxInclusive", "20", )
```

```
<xsd:complexType name="student-type">
…
</xsd:complexType>
<xsd:simpleType name="examination-mark-type">
  <xsd:restriction base="xsd:float">
    <xsd:minInclusive value="0"/>
    <xsd:maxInclusive value="20"/>
  </xsd:restriction>
</xsd:simpleType>
```

Figure 7: Effects of the DerivingNewSimpleTypeByRestriction operation.

$Op^{CS}22$: DefineNewSimpleTypeThroughEnumeration(CS.xsd, targetElementPath, position, newSimpleTypeName, baseType, distinctValueSet)

It creates a new simple type (having the name "newSimpleTypeName") by limiting the values of an existing simple type (having the name "baseType") to a set of distinct values that are provided by the designer in the fourth parameter "distinctValueSet" (which is of string type) and are separated by

semicolons, through the use of the facet “enumeration”. This operation adds the new simple type at a specified position (i.e., after or before) with regard to another target element (located at targetElementPath).

Example: Suppose that the designer would like to specify a restriction on the type “xsd:string”, in order to define a new data type “customer-type” which contains the following list of values: “regular” and “temporary”, as shown in Figure 8. To do this, he/she calls the DefineNewSimpleTypeThroughEnumeration operation as follows:

```
DefineNewSimpleTypeThroughEnumeration(CS.xsd, "//xsd:complexType[@name='product-type']",
        after, "customer-type", "xsd:string", "regular;temporary")
```

```
<xsd:complexType name="product-type">
…
</xsd:complexType>
<xsd:simpleType name="customer-type">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="regular"/>
    <xsd:enumeration value="temporary"/>
  </xsd:restriction>
</xsd:simpleType>
```

Figure 8: Effects of the DefineNewSimpleTypeThroughEnumeration operation.

$Op^{CS}23$: DropDataTypeRestriction(CS.xsd, dataTypeRestrictionPath)

It drops a restriction that was specified on a data type, in the conventional schema “CS.xsd”.

Such an operation could be performed if and only if there is no element, in this conventional schema, that has this “data type restriction” as a type.

Example: Suppose that the designer would like to drop a data type restriction titled “bicycle-type” (since the company has decided not to sell any more bicycles). To do this, he/she calls the DropDataTypeRestriction operation as follows:

```
DropDataTypeRestriction(CS.xsd, "//xsd:simpleType[@name='bicycle-type']")
```

$Op^{CS}24$: RenameDataTypeRestriction(CS.xsd, dataTypeRestrictionPath, newName)

It changes the name of a datatype restriction (located at “datatypeRestrictionPath”) with “newName”, in the conventional schema “CS.xsd”. This operation must update all other components of this schema that are using (or referring to) this datatype restriction by replacing the old name with the new one.

Example: Suppose that the designer would like to change the name of the datatype restriction “customer-type” (presented above) to “customer-category”. To do this, he/she calls the RenameDataTypeRestriction operation as follows:

```
RenameDataTypeRestriction(CS.xsd, "//xsd:simpleType[@name='customer-type']", "customer-
        category")
```

3.1.3.2. Basic High-Level Operations dealing with Identity Constraints

An identity constraint [1] is a key or a unique constraint defined by a conventional schema designer on an element or on an attribute.

An identity constraint is defined using one of the two XML Schema components: <xsd:key> and <xsd:unique>. Each one of these two components is a container and it should contain an <xsd:selector> component and one or more <xsd:field> components. Both <xsd:selector> and <xsd:field> components contain an XPath expression.

In the following, we first present high-level operations for changing uniqueness constraints and then we give high-level operations for changing key constraints.

A) Uniqueness Constraints

Op^{CS}25: DefineUniquenessConstraint(CS.xsd, targetElementPath, position, uniquenessConstraintName, selectorXpath, firstFieldXpath, secondFieldXpath)

It specifies a new uniqueness constraint (with the name "uniquenessConstraintName"), in the conventional schema "CS.xsd", to express that the value of an element or an attribute must be unique within a certain scope. To do this, this operation first selects a set of parent elements (through "selectorXpath") and then indicates the attribute(s) and/or the child element(s) "filed(s)" (through "firstFieldXpath" and possibly "secondFieldXpath") related to each selected element, that has (or have) to be unique within the scope of the set of selected elements. This operation adds the new constraint at a specified position (i.e., after or before) with regard to another target element (located at targetElementPath).

Example: Suppose that the designer would like to specify a uniqueness constraint on the attribute <SSN> of an element <employee>, as shown in Figure 9. To do this, he/she calls the DefineUniquenessConstraint operation as follows:

```
DefineUniquenessConstraint(CS.xsd, "//xsd:element[@name='employees']/complexType", after,
                    "uniqueSSN", "employee", "@SSN", )
```

```
<xsd:element name="employees">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element name="employee" maxOccurs="unbounded">
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      …
     </xsd:sequence>
     <xsd:attribute name="SSN" type="xsd:string"/>
    </xsd:complexType>
   </xsd:element>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:unique name="uniqueSSN">
  <xsd:selector xpath="employee"/>
  <xsd:field xpath="@SSN"/>
 </xsd:unique>
</xsd:element>
```

Figure 9: Effects of the DefineUniquenessConstraint operation.

$Op^{CS}26$: DropUniquenessConstraint(CS.xsd, uniquenessConstraintPath)

It drops a uniqueness constraint (located at "uniquenessConstraintPath") from the conventional schema "CS.xsd".

Such an operation could be performed if and only if there is no referential integrity constraint (defined using a `keyref` component), in this conventional schema, that refers to the uniqueness constraint to be dropped.

Example: Suppose that the designer would like to drop a uniqueness constraint that was defined on a sub-element `<phone>` of a complex element `<employee>`. To do this, he/she calls the DropUniquenessConstraint operation as follows:

```
DropUniquenessConstraint(CS.xsd, "//xsd:unique[@name='uniquePhone']")
```

$Op^{CS}27$: RenameUniquenessConstraint(CS.xsd, uniquenessConstraintPath, newName)

It changes the name of a uniqueness constraint (located at "uniquenessConstraintPath") to "newName", in the conventional schema "CS.xsd". This operation must update all other components of this schema that are using (or referring to) this uniqueness constraint by replacing the old name with the new one.

B) Key Constraints

$Op^{CS}28$: DefineKeyConstraint(CS.xsd, targetElementPath, position, keyConstraintName, selectorXpath, firstFieldXpath, secondFieldXpath)

It specifies a new key constraint (with the name "keyConstraintName"), in the conventional schema "CS.xsd", to express that the value of an element or an attribute must be unique within a certain scope and must not be set to nil. To do this, this operation first selects a set of parent elements (through "selectorXpath") and then indicates the attribute(s) and/or the child element(s) "filed(s)" (through "firstFieldXpath" and possibly "secondFieldXpath") related to each selected element, that has (or have) to be unique within the scope of the set of selected elements and also has (or have) not to be nillable. This operation adds the new constraint at a specified position (i.e., after or before) with regard to another target element (located at targetElementPath).

Example: Suppose that the designer would like to specify a key constraint on the attribute "customer-code" of an element `<customer>`. To do this, he/she calls the DefineKeyConstraint operation as follows:

```
DefineKeyConstraint(CS.xsd, "//xsd:element[@name='customers']/complexType", after, "key-customer-
                    code", "customer", "@customer-code", )
```

$Op^{CS}29$: DropKeyConstraint(CS.xsd, keyConstraintPath)

It drops a key constraint (located at "keyConstraintPath") from the conventional schema "CS.xsd".

Such an operation could be performed if and only if there is no referential integrity constraint (defined using a `keyref` component), in this conventional schema, that refers to the key constraint to be

dropped.

Op^{CS}30: RenameKeyConstraint(CS.xsd, keyConstraintPath, newName)

It changes the name of a key constraint (located at "keyConstraintPath") to "newName", in the conventional schema "CS.xsd". This operation must update all other components of this schema that are using (or referring to) this key constraint by replacing the old name with the new one.

3.1.3.3. Basic High-Level Operations dealing with Referential Integrity Constraints

A referential integrity constraint [1] in the XML setting is similar to the corresponding constraint in the relational setting. It refers to a key or a unique constraint already defined by a conventional schema designer on an element or on an attribute.

A referential integrity constraint is defined using the XML Schema component `<xsd:keyref>`. This component is a container and it should contain an `<xsd:selector>` component and one or more `<xsd:field>` components. Both `<xsd:selector>` and `<xsd:field>` components contain an XPath expression. Moreover, notice that `keyref` uses a key via its attribute `refer` and that the list, the order and the types of all fields in a `keyref` must be identical to those of the corresponding key (i.e., named by `refer`).

Op^{CS}31: DefineReferentialIntegrityConstraint(CS.xsd, targetElementPath, position, referentialIntegrityConstraintName, identityConstraintPath)

It specifies, in the conventional schema "CS.xsd", a new referential integrity constraint (with the name "referentialIntegrityConstraintName") that refers to the identity (i.e., uniqueness or key) constraint located at "identityConstraintPath". This operation adds the new constraint at a specified position (i.e., after or before) with regard to another target element (located at "targetElementPath").

Such an operation could not be performed if the key or the unique constraint, to which the referential integrity constraint refers, does not exist.

Example: Suppose that the designer would like to specify a referential integrity constraint within an element `<invoices>` that refers to the key constraint titled "key-customer-code" that was defined on the attribute "customer-code" of the element `<customer>` (see example of Op^{CS}28). To do this, he/she calls the DefineReferentialIntegrityConstraint operation as follows:

```
DefineReferentialIntegrityConstraint(CS.xsd, "//xsd:element[@name='invoices']/complexType", after,
        "refer-customer", "//xsd:key[@name='key-customer-code']")
```

Op^{CS}32: DropReferentialIntegrityConstraint(CS.xsd, referentialIntegrityConstraintPath)

It drops a referential integrity constraint (located at "referentialIntegrityConstraintPath"), in the conventional schema "CS.xsd".

Op^{CS}33: RenameReferentialIntegrityConstraint(CS.xsd, referentialIntegrityConstraintPath, newName)

It changes the name of a referential integrity constraint (located at "referentialIntegrityConstraintPath")

to “newName”, in the conventional schema “CS.xsd”. This operation must update all other components of this schema that are using (or referring to) this referential integrity constraint by replacing the old name with the new one.

3.1.3.4. Basic High-Level Operations dealing with Cardinality Constraints

Cardinality constraints are occurrence constraints. They could be defined on elements or on attributes. For elements, they are used to define how often an element can occur (through `minOccurs` and `maxOccurs` attributes of the XML Schema `<xsd:element>` component). For attributes, cardinality constraints are used to define whether the attribute must appear or not in the corresponding element (through the value `required` or `optional` that should be assigned to the `use` attribute of the XML Schema `<xsd:attribute>` component).

A) Basic High-Level Operations dealing with Cardinality Constraints on Elements

The three following operations act on the `minOccurs` and `maxOccurs` attributes of the XML Schema `<xsd:element>` component.

$Op^{CS}34$: SpecifyCardinalityConstraintOnElement(CS.xsd, elementPath, valueMinOccurs, valueMaxOccurs)

It specifies a cardinality constraint on an element (by adding the attribute `minOccurs` and/or the attribute `maxOccurs` to the corresponding `<xsd:element>` definition and by assigning a value to each one of them), in the conventional schema “CS.xsd”.

Example: Suppose that the designer would like to specify a cardinality constraint on a subelement `<ISBN>` of a complex element `<Book>`, in order to express that a book could not have an ISBN. To do this, he/she calls the SpecifyCardinalityConstraintOnElement operation as follows:

```
SpecifyCardinalityConstraintOnElement(CS.xsd, "//xsd:element[@name='ISBN']", "0", )
```

$Op^{CS}35$: DropCardinalityConstraintOnElement(CS.xsd, elementPath, MinOccurs, MaxOccurs)

It drops a cardinality constraint that was specified on an element (by removing the attribute `minOccurs` and/or the attribute `maxOccurs` from the corresponding `<xsd:element>` definition), in the conventional schema “CS.xsd”.

$Op^{CS}36$: ChangeCardinalityConstraintOnElement(CS.xsd, elementPath, newValueMinOccurs, newValueMaxOccurs)

It changes a cardinality constraint that was specified on an element (by assigning a new value to the attribute `minOccurs` and/or a new value to the attribute `maxOccurs` in the corresponding `<xsd:element>` definition), in the conventional schema “CS.xsd”.

B) Basic High-Level Operations dealing with Cardinality (or Optionality) Constraints on Attributes

The three following operations act on the `use` attribute of the XML Schema `<xsd:attribute>`

component.

$Op^{CS}37$: SpecifyOptionalityConstraintOnAttribute(CS.xsd, attributePath, valueUse)

It specifies a cardinality constraint on an attribute of an element (by adding the attribute `use` to the corresponding `<xsd:attribute>` definition and by assigning a value (`required` or `optional`) to it), in the conventional schema "CS.xsd".

Example: Suppose that the designer would like to specify an optionality constraint on the attribute "SSN" of the element `<employee>` (see Figure 9), in order to express that each employee should have a social security number. To do this, he/she calls the SpecifyOptionalityConstraintOnAttribute operation as follows:

```
SpecifyOptionalityConstraintOnAttribute(CS.xsd, "//xsd:attribute[@name='SSN']", "required")
```

$Op^{CS}38$: DropOptionalityConstraintOnAttribute(CS.xsd, attributePath)

It drops a cardinality constraint that was specified on an attribute of an element (by removing the attribute `use` from the corresponding `<xsd:attribute>` definition), in the conventional schema "CS.xsd".

$Op^{CS}39$: ChangeOptionalityConstraintOnAttribute(CS.xsd, attributePath, newValueUse)

It changes a cardinality constraint that was specified on an attribute of an element (by assigning a new value to the attribute `use` in the corresponding `<xsd:attribute>` definition), in the conventional schema "CS.xsd".

3.2. Complex High-Level Operations

In this subsection, we study complex high-level schema change operations (i.e., high-level operations that are defined by using other basic and/or complex high-level operations). More precisely, we propose ten schema change operations: five operations acting on whole conventional schema in the subsection 3.2.1 and five operations acting on portions of conventional schema (i.e., subschema) in the subsection 3.2.2.

3.2.1. Complex High-level Operations dealing with whole Conventional Schema

$Op^{CS}40$: CreateConventionalSchemaByExtraction(sourceCS, selectionBeginningPath, selectionEndPath, targetCS, option)

It extracts some XSD code (which starts at "selectionBeginningPath" and ends at "selectionEndPath") from a source conventional schema ("sourceCS") and saves it as a new target conventional schema ("targetCS"), with specified option (leave, delete, or link). The option parameter values are detailed as follows:

1) **leave**: the sourceCS conventional schema is left unchanged after the extraction (i.e. the operation simply saves a copy of the selected subschema into the new targetCS);

2) **delete**: the selected subschema is deleted from the sourceCS conventional schema after the extraction (this would potentially require propagation of heavy modifications to the XML data instances);

3) **link**: the selected subschema is substituted in the sourceCS conventional schema by: `<xsd:include schemaLocation="targetCS"/>` after the extraction (unless the use of "xsd:include" makes some troubles in the tauXSchema framework, this option leaves the conventional schema globally (virtually) unchanged even if it has been split into two parts with the extraction).

Furthermore, we should notice that the extracted subschema has to be completed with the required headers (which may include the outermost `<xsd:schema/>` element) in order to be saved as a valid independent conventional schema.

Example: Suppose that the designer would like to create a conventional schema for customers which are organizations by extracting its XSD code from an existing conventional schema for customers (see Figure 10), with option "link". To do this, he/she calls the CreateConventionalSchemaByExtraction operation as follows:

```
CreateConventionalSchemaByExtraction(customers.xsd,
            "//xsd:element[@name="customer-organizations"]",
            "//xsd:element[@name="customer-organizations"]",
            customerOrganizations.xsd, link).
```

After the execution of this operation, the desired conventional schema will be created (see Figure 11); the conventional schema for customers will be changed and becomes as in Figure 12.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="customer-persons">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="customer-person" maxOccurs="unbounded">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element name="name" type="xsd:string"/>
       <xsd:element name="address" type="xsd:string"/>
       <xsd:element name="phone" type="xsd:string"/>
       <xsd:element name="turnover" type="xsd:double"/>
       <xsd:element name="birthdate" type="xsd:date"/>
      </xsd:sequence>
     </xsd:complexType>
    </xsd:element>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="customer-organizations">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="customer-organization"
                 maxOccurs="unbounded">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element name="name" type="xsd:string"/>
       <xsd:element name="address" type="xsd:string"/>
       <xsd:element name="phone" type="xsd:string"/>
       <xsd:element name="turnover" type="xsd:double"/>
       <xsd:element name="activityDomain" type="xsd:string"/>
      </xsd:sequence>
     </xsd:complexType>
```

```
    </xsd:element>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Figure 10: Conventional schema for customers (initial customers.xsd).

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="customer-organizations">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="customer-organization"
                 maxOccurs="unbounded">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element name="name" type="xsd:string"/>
       <xsd:element name="address" type="xsd:string"/>
       <xsd:element name="phone" type="xsd:string"/>
       <xsd:element name="turnover" type="xsd:double"/>
       <xsd:element name="activityDomain" type="xsd:string"/>
      </xsd:sequence>
     </xsd:complexType>
    </xsd:element>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Figure 11: Conventional schema for customers which are organizations (customerOrganizations.xsd).

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="customer-persons">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="customer-person" maxOccurs="unbounded">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element name="name" type="xsd:string"/>
       <xsd:element name="address" type="xsd:string"/>
       <xsd:element name="phone" type="xsd:string"/>
       <xsd:element name="turnover" type="xsd:double"/>
       <xsd:element name="birthdate" type="xsd:date"/>
      </xsd:sequence>
     </xsd:complexType>
    </xsd:element>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:include schemaLocation="customerOrganizations.xsd">
</xsd:schema>
```

Figure 12: Changed conventional schema for customers (final customers.xsd).

$Op^{CS}41$: MergeConventionalSchema(CS.xsd, sourceCS, targetElementPath, position)

It inserts all the XSD code of a source conventional schema ("sourceCS") into another target conventional schema ("CS.xsd") at a specified position (i.e., before or after) with regard to a target element (located at "targetElementPath") in the target schema.

Example: Suppose that a designer in an enterprise has a requirement which consists in merging into a conventional schema that describes (permanent) employees "employees.xsd" (see Figure 13) a conventional schema that describes temporary employees "temporary-employees.xsd" (see Figure 14). To do this, he/she calls the MergeConventionalSchema operation as follows:

```
MergeConventionalSchema(employees.xsd, temporary-employees.xsd, "/", after)
```

The results could be as in Figure 15.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="employees">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="employee" maxOccurs="unbounded">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element name="firstName" type="xsd:string"/>
       <xsd:element name="lastName" type="xsd:string"/>
       <xsd:element name="hireDate" type="xsd:date"/>
       <xsd:element name="salary" type="xsd:float"/>
      </xsd:sequence>
     </xsd:complexType>
    </xsd:element>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Figure 13: Conventional schema for permanent employees (initial employees.xsd).

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="temporaryEmployees">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="temporaryEmployee" maxOccurs="unbounded">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element name="firstName" type="xsd:string"/>
       <xsd:element name="lastName" type="xsd:string"/>
       <xsd:element name="hireDate" type="xsd:date"/>
       <xsd:element name="period" type="xsd:positiveInteger"/>
       <xsd:element name="hourlyRate" type="xsd:float"/>
      </xsd:sequence>
     </xsd:complexType>
    </xsd:element>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
```

```
</xsd:schema>
```

Figure 14: Conventional schema for temporary employees (temporary-employees.xsd).

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="employees">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:choice maxOccurs="unbounded">
     <xsd:element name="permanentEmployee">
      <xsd:complexType>
       <xsd:sequence>
        <xsd:element name="firstName" type="xsd:string"/>
        <xsd:element name="lastName" type="xsd:string"/>
        <xsd:element name="hireDate" type="xsd:date"/>
        <xsd:element name="salary" type="xsd:float"/>
       </xsd:sequence>
      </xsd:complexType>
     </xsd:element>
     <xsd:element name="temporaryEmployee">
      <xsd:complexType>
       <xsd:sequence>
        <xsd:element name="firstName" type="xsd:string"/>
        <xsd:element name="lastName" type="xsd:string"/>
        <xsd:element name="hireDate" type="xsd:date"/>
        <xsd:element name="period" type="xsd:positiveInteger"/>
        <xsd:element name="hourlyRate" type="xsd:float"/>
       </xsd:sequence>
      </xsd:complexType>
     </xsd:element>
    </xsd:choice>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Figure 15: Merged conventional schema for employees (final employees.xsd).

For the best of our knowledge, merging of schema was studied in classical databases and more precisely within the topic of database schema integration [6,7,8,9,10,11,12] or database schema coercion [13], but it has not been studied as a schema versioning issue (neither under relational/object/object-oriented databases nor under XML databases). We first propose to use merging of schemata as a high-level change operation in the context of schema versioning.

$Op^{CS}42$: ReplaceConventionalSchema(CS.xsd, newCS)

It replaces an existing conventional schema ("CS.xsd") with a new one ("newCS"); this latter could be provided by the designer either as a string explicitly containing the new schema text or as a file name corresponding to a source conventional schema to be used for replacement.

Such an operation must fail with an error message in case the operation finds other conventional schema (in the database) which are referring to the conventional schema that should be replaced. Otherwise, this operation must also drop the annotation document corresponding to the replaced conventional schema.

$Op^{CS}43$: DropConventionalSchema(CS.xsd)

It removes a conventional schema ("CS.xsd") from the database.

Such an operation must fail with an error message in case the operation finds other conventional schema (in the database) which are referring to the conventional schema that should be dropped. Otherwise, this operation must also drop the annotation document corresponding to the removed conventional schema.

$Op^{CS}44$: RenameConventionalSchema(CS.xsd, newName)

It changes the name of a conventional schema ("CS.xsd") to "newName".

This operation must update all other components of the database that are referring to this conventional schema, by replacing the old name with new one.

3.2.2. Complex High-level Operations dealing with Portions of Conventional Schema

$Op^{CS}45$: InsertSubSchema(CS.xsd, targetElementPath, position, subSchema)

It inserts a new subschema ("subSchema") at a specified position (i.e., before or after) with regard to a target element (located at "targetElementPath") in the conventional schema "CS.xsd".

The new subschema to be inserted can be provided by the designer either as a string explicitly containing the subschema text or as a file name corresponding to a source conventional schema to be used for insertion (after removal of headers).

Example: Suppose that the designer would like to add a new subschema that describes foreign students at the end (i.e., after the complex element <students/>) of the current conventional schema which describes only local students "students_V1.xsd" (see Figure 16). To do this, he/she calls the InsertSubSchema operation as follows:

```
InsertSubSchema(students_V1.xsd, "//xsd:element[@name='students']", after,
    "<xsd:element name="foreign-students">
      <xsd:complexType>
       <xsd:sequence>
        <xsd:element name="foreign-student" maxOccurs="unbounded">
         <xsd:complexType>
          <xsd:sequence>
           <xsd:element name="name" type="xsd:string"/>
           <xsd:element name="address" type="xsd:string"/>
           <xsd:element name="phone" type="xsd:string"/>
           <xsd:element name="country" type="xsd:string"/>
          </xsd:sequence>
         </xsd:complexType>
        </xsd:element>
       </xsd:sequence>
      </xsd:complexType>
    </xsd:element>")
```

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="students">
  <xsd:complexType>
```

```
    <xsd:sequence>
     <xsd:element name="student" maxOccurs="unbounded">
      <xsd:complexType>
       <xsd:sequence>
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="address" type="xsd:string"/>
        <xsd:element name="phone" type="xsd:string"/>
       </xsd:sequence>
      </xsd:complexType>
     </xsd:element>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Figure 16: Conventional schema for only local students (students_V1.xsd).

Figure 17 shows the new version "students_V2.xsd" of the updated conventional schema.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="students">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="student" maxOccurs="unbounded">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element name="name" type="xsd:string"/>
       <xsd:element name="address" type="xsd:string"/>
       <xsd:element name="phone" type="xsd:string"/>
      </xsd:sequence>
     </xsd:complexType>
    </xsd:element>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="foreign-students">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="foreign-student" maxOccurs="unbounded">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element name="name" type="xsd:string"/>
       <xsd:element name="address" type="xsd:string"/>
       <xsd:element name="phone" type="xsd:string"/>
       <xsd:element name="country" type="xsd:string"/>
      </xsd:sequence>
     </xsd:complexType>
    </xsd:element>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Figure 17: New conventional schema for local and foreign students (students_V2.xsd).

$Op^{CS}46$: RemoveSubSchema(CS.xsd, subSchemaBeginning, subSchemaEnd)

It removes a subschema which starts at a specified position ("subSchemaBeginning") and ends at another specified position ("subschemaEnd"), in the conventional schema "CS.xsd".

Such an operation must fail with an error message in case the operation finds other components of the same conventional schema, which are using, directly (e.g., as a type for elements) or indirectly (e.g., as base type of an element's type), some components of the removed subschema. Otherwise, this operation must also (i) remove, from the corresponding conventional schema, all constraints (uniqueness, key, referential integrity, and datatype constraints) that are defined on items belonging completely to the removed subschema, and (ii) update (or remove), in the annotation document corresponding to the conventional schema which includes the subschema to be removed:

- all `<item>` and `<stamp>` components that are referring to `<element>` components in the removed subschema;
- all `<stamp>` components that are referring to `<attribute>` components in the removed subschema.

Notice here that the constraints (uniqueness, key, referential integrity, and datatype constraints) defined on items belonging completely to the removed subschema, do not forbid the removing of the subschema since their remove does not have any effect on the consistency and on the validity of the new schema.

$Op^{CS}47$: ReplaceSubSchema(CS.xsd, subSchemaBeginning, subSchemaEnd, newSubSchema)

It replaces a subschema (which starts at "subSchemaBeginning" and ends at "subSchemaEnd") by a new subschema ("newSubSchema"), in the same conventional schema "CS.xsd".

The new subschema to be inserted can be provided by the designer either as a string explicitly containing the subschema text or as a file name corresponding to a source conventional schema to be used for replacement (after removal of headers).

Such an operation must fail with an error message in case the operation finds other components of the same conventional schema, which are using, directly (e.g., as a type for elements) or indirectly (e.g., as base type of some element types), some components of the replaced subschema. Otherwise, this operation must also (i) remove, from the corresponding conventional schema, all constraints (uniqueness, key, referential integrity, and datatype constraints) that are defined on items belonging completely to the replaced subschema, and (ii) update (or remove), in the annotation document corresponding to the conventional schema which includes the subschema to be replaced:

- all `<item>` and `<stamp>` components that are referring to `<element>` components in the replaced subschema;
- all `<stamp>` components that are referring to `<attribute>` components in the replaced subschema.

$Op^{CS}48$: MoveSubSchema(CS.xsd, subSchemaBeginning, subSchemaEnd, targetElementPath, position)

It moves a subschema (which starts at "subSchemaBeginning" and ends at "subSchemaEnd") to a new position (i.e., before or after) with regard to a target element (located at "targetElementPath"), in the same conventional schema "CS.xsd".

Such an operation must fail with an error message in case the operation finds other components of the same conventional schema, which are using, directly (e.g., as a type for elements) or indirectly (e.g., as base type of some element types), some components of the replaced subschema.

This operation must update (i) in the other parts of the corresponding conventional schema, all components (constraints, complex elements, …) that are using (or referring to) components belonging to the moved subschema, and (ii) in the annotation document corresponding to the conventional schema which includes the subschema to be moved:

- all <item> and <stamp> components that are referring to <element> components in the moved subschema;
- all <stamp> components that are referring to <attribute> components in the moved subschema.

$Op^{CS}49$: CopySubSchema(CS.xsd, subSchemaBeginning, subSchemaEnd, targetElementPath, position)

It copies a subschema (which starts at "subSchemaBeginning" and ends at "subSchemaEnd") into a specified position (i.e., before or after) with regard to a target element (located at "targetElementPath"), in the same conventional schema "CS.xsd".

4. High-level Operations for Changing Logical and Physical Annotations

We have defined fifty-five high-level schema change operations which act on the annotation document. We organize them into three categories: (i) operations that are common to the logical and to the physical annotations, presented in the subsection 4.1, (ii) operations that are specific to the logical annotations, described in the subsection 4.2, and (iii) operations that are specific to the physical annotations, presented in the subsection 4.3.

To illustrate the proposed operations, we will use, as an example, annotations that are associated to a given conventional schema (see Figure 18) which describes a commercial enterprise that sells equipments to cutomers; invoices of these latters are also modelled.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="enterprise">
  <xsd:complexType mixed="true">
   <xsd:sequence>
    <xsd:element ref="equipment" minOccurs="1"
                 maxOccurs="unbounded"/>
    <xsd:element ref="customer" minOccurs="1"
                 maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
  <xsd:key name="equipmentKey">
   <xsd:selector xpath="enterprise/equipment"/>
   <xsd:field xpath="@equipmentNo"/>
  </xsd:key>
 </xsd:element>
 <xsd:element name="equipment">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="equipmentName" type="xsd:string"
                 minOccurs="1"/>
```

```
     <xsd:element name="qtyInStock" type="xsd:positiveInteger"
                  minOccurs="1" maxOccurs="1"/>
     <xsd:element name="price" type="xsd:float" minOccurs="1"
                  maxOccurs="1"/>
    </xsd:sequence>
    <xsd:attribute name="equipmentNo" type="xsd:positiveInteger"
                  use="required"/>
    <xsd:attribute name="equipmentCategory" type="xsd:string"
                  use="required"/>
   </xsd:complexType>
  </xsd:element>
  <xsd:element name="customer">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="customerName" type="xsd:string"
                  minOccurs="1"/>
     <xsd:element name="customerAddress" type="xsd:string"
                  minOccurs="0" maxOccurs="unbounded"/>
     <xsd:element name="invoice" minOccurs="0"
                  maxOccurs="unbounded">
      <xsd:complexType>
       <xsd:sequence>
        <xsd:element name="invoiceNo" type="xsd:positiveInteger"
                     minOccurs="1"/>
        <xsd:element name="invEquipmentNo" minOccurs="1"
                     maxOccurs="unbounded"/>
        <xsd:element name="invEquipmentQty"
                     type="xsd:positiveInteger" minOccurs="1"
                     maxOccurs="1"/>
       </xsd:sequence>
      </xsd:complexType>
     </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="customerNo" type="xsd:positiveInteger"
                  use="required"/>
   </xsd:complexType>
  </xsd:element>
  <xsd:unique name="uniqueCustomerNo">
   <xsd:selector xpath="customer">
   <xsd:field    xpath="@customerNo">
  </xsd:unique>
</xsd:schema>
```

Figure 18: Conventional Schema for a commercial enterprise (enterprise_V1.xsd).

4.1. High-level Operations Common to Logical and Physical Annotations

We have defined seven high-level change operations that are common to the logical and to the physical annotations. They are as follows:

$Op^{AD}01$: IncludeAnnotationFiles(AD.xml, logicalAnnotationFileLocation,

physicalAnnotationFileLocation)

It includes, in the annotation document "AD.xml", an XML document that contains logical annotations, located at "logicalAnnotationFileLocation", or an XML document that contains physical annotations, located at "physicalAnnotationFileLocation" (only one of these two parameters can be omitted).

This operation is mapped onto the following list of primitives:

```
(1) If (logicalAnnotationFileLocation is not null) then
        AddInclude(AD.xml, logical, logicalAnnotationFileLocation)
(2) If (physicalAnnotationFileLocation is not null) then
        AddInclude(AD.xml, physical, physicalAnnotationFileLocation)
```

Example: Suppose that the designer would like to include in an annotation document "EnterpriseAnnotations.xml" an XML document that contains logical annotations, located at "http://www.enterprise.com/Annotations/enterpriseLogicalAnnotations.xml", and an XML document that contains physical annotations, located at "http://www.enterprise.com/Annotations/enterprisePhysicalAnnotations.xml". To do this, he/she calls the IncludeAnnotationFiles operation as follows:

```
IncludeAnnotationFiles(EnterpriseAnnotations.xml,
        "http://www.enterprise.com/Annotations/enterpriseLogicalAnnotations.xml",
        "http://www.enterprise.com/Annotations/enterprisePhysicalAnnotations.xml")
```

$Op^{AD}02$: SpecifyDefaultTimeFormatUsedInAnnotationDocument(AD.xml, annotationType, pluginUsed, granularityOfTimeFormat, calendricSystemUsed, dateFormatProperties, valueSchemaUsedForDate)

It specifies the default time format used in the annotation document "AD.xml" for logical or for physical annotations (according to the **annotationType** parameter that should have the value **logical** or **physical**). This default time format has the following properties: pluginUsed, granularityOfTimeFormat, calendricSystemUsed, dateFormatProperties, and valueSchemaUsedForDate.

This operation is mapped onto the following list of primitives:

```
if (annotationType = logical) then
AddDefaultTimeFormat(AD.xml, logical, pluginUsed, granularityOfTimeFormat, calendricSystemUsed,
        dateFormatProperties, valueSchemaUsedForDate)
else
AddDefaultTimeFormat(AD.xml, physical, pluginUsed, granularityOfTimeFormat,
        calendricSystemUsed, dateFormatProperties, valueSchemaUsedForDate)
```

$Op^{AD}03$: RemoveAnnotationDocument(AD.xml)

It removes an annotation document "AD.xml" from the database.

$Op^{AD}04$: RenameAnnotationDocument(AD.xml, newName)

It changes the name of an existing annotation document ("AD.xml") with a new name ("newName").

$Op^{AD}05$: ReplaceAnnotationDocument(AD.xml, newAD)

It replaces an existing annotation document ("AD.xml") with a new one ("newAD"); this latter could be provided by the designer either as a string explicitly containing the new annotation document text or as a file name corresponding to a source annotation document to be used for replacement.

$Op^{AD}06$: MergeAnnotationDocuments(AD.xml, sourceAD)

It inserts all the XML code of a source annotation document ("sourceAD") into another target annotation document ("AD.xml") as follows: the set of logical annotations of "sourceAD" are inserted at the end of the logical annotations of "AD.xml" (within the `<logical/>` container) and the set of physical annotations of "sourceAD" are inserted at the end of the physical annotation of "AD.xml" (within the `<physical/>` container).

Notice here that the set of logical (physical, respectively) annotations of "sourceAD" could also be inserted at the beginning of the logical (physical, respectively) annotations of "AD.xml" within the `<logical/>` (`<physical/>`, respectively) container, since the ordering of `<item/>` elements (`<stamp/>` elements, respectively) in the `<logical/>` (`<physical/>`, respectively) container is unimportant.

$Op^{AD}07$: SplitAnnotationDocument(AD.xml, logicalAnnotationsFirstNewAD, physicalAnnotationsFirstNewAD, logicalAnnotationsSecondNewAD, physicalAnnotationsSecondNewAD)

It decomposes an existing annotation document ("AD.xml") into two new annotation documents as follows: the first one contains "logicalAnnotationsFirstNewAD" as a set of logical annotations and "physicalAnnotationsFirstNewAD" as a set of physical annotations; the second one stores "logicalAnnotationsSecondNewAD" and "physicalAnnotationsSecondNewAD".

. Each one of the last four parameters is an XML code provided by the designer as a string which must be a set of annotations extracted from "AD.xml" as follows: the contents of the "logicalAnnotationsFirstNewAD" or the "logicalAnnotationsSecondNewAD" parameter is a set of `<item/>` elements extracted from "AD.xml"; the contents of the "physicalAnnotationsFirstNewAD" or the "physicalAnnotationsSecondNewAD" parameter is a set `<stamp/>` elements extracted from "AD.xml".

Furthermore, the set of extracted logical and physical annotations has to be completed with the required headers, in order to be saved in valid independent annotation documents. Indeed, first, the contents of the "logicalAnnotationsFirstNewAD" or the "logicalAnnotationsSecondNewAD" parameter has to be completed with the `<logical/>` element and the contents of the "physicalAnnotationsFirstNewAD" or the "physicalAnnotationsSecondNewAD" parameter has to be completed with the `<physical/>` element. Then, the contents of each one of the two couples ("logicalAnnotationsFirstNewAD", "physicalAnnotationsFirstNewAD") and ("logicalAnnotationsSecondNewAD", "physicalAnnotationsSecondNewAD") has to be completed by the outermost `<annotationSet/>` element.

4.2. High-level Operations for Changing Logical Annotations

We have defined thirty-eight high-level operations for changing logical annotations. We organize them into three categories: (i) operations that act on the whole logical annotation set, presented in the subsection 4.2.1, (ii) operations that act on a portion of a logical annotation set, described in the subsection 4.2.2, and (iii) operations that act on a time-varying item, presented in the subsection 4.2.3.

4.2.1. High-Level Operations dealing with the whole Logical Annotation Set

$Op^{AD}8$: InsertLogicalAnnotationSet(AD.xml, logicalAnnotationSet)

It adds, in an annotation document "AD.xml", a set of logical annotations ("logicalAnnotationSet": a data of string type). Since the ordering of <item/> elements in the <logical/> container is unimportant, this operation adds the new set of logical annotations (i.e., the new set of <item/> elements) at the end of the existing set of logical annotations, within the <logical/> container. If this latter does not exist in "AD.xml", this operation first creates it and then inserts the new set of logical annotations.

Example: Suppose that the designer would like to add a set of logical annotations in the annotation document "EnterpriseAnnotations.xml". To do this, he/she calls the InsertLogicalAnnotationSet operation as follows:

```
InsertLogicalAnnotationSet(EnterpriseAnnotations.xml,
  "<logical>
    <item target="/enterprise">
     <transactionTime/>
     <itemIdentifier name="enterpriseId1"
                    timeDimension="transactionTime">
      <field path="//text"/>
     </itemIdentifier>
    </item>
    <item target="/enterprise/customer">
     <validTime kind="state" content="varying"
                          existence="varyingWithGaps"/>
     <transactionTime/>
     <itemIdentifier name="customerId1" timeDimension="bitemporal">
      <field path="@customerNo"/>
     </itemIdentifier>
    </item>
    <item target="/enterprise/equipment/price">
     <validTime kind="state" content="varying"/>
     <transactionTime/>
     <itemIdentifier name="priceId1" timeDimension="bitemporal">
      <field path="."/>
     </itemIdentifier>
    </item>
   </logical>")
```

$Op^{AD}9$: RemoveLogicalAnnotationSet(AD.xml)

It removes the set of logical annotations (i.e., all sub-elements of the <logical/> element) from an annotation document "AD.xml".

$Op^{AD}10$: ReplaceLogicalAnnotationSet(AD.xml, newLogicalAnnotationSet)

It replaces, in an annotation document "AD.xml", the existing set of logical annotations with a new set of logical annotations ("newLogicalAnnotationSet") stored in an XML document.

4.2.2. High-Level Operations dealing with a Portion of Logical Annotations

$Op^{AD}11$: InsertLogicalAnnotationSubSet(AD.xml, logicalAnnotationSubSet)

It adds, in an annotation document "AD.xml", a subset of logical annotations ("logicalAnnotationSubSet": a data of string type). Since the ordering of `<item/>` elements, in the `<logical/>` container, is unimportant, this operation adds the new subset of logical annotations (i.e., the new subset of `<item/>` elements) at the end of the existing set of logical annotations, within the `<logical/>` container. If this latter does not exist in "AD.xml", this operation first creates it and then inserts the subset of logical annotations.

Example: Suppose that the designer would like to add a subset of logical annotations in the annotation document "EnterpriseAnnotations.xml". To do this, he/she calls the InsertLogicalAnnotationSubSet operation as follows:

```
InsertLogicalAnnotationSubSet(EnterpriseAnnotations.xml,
    "<item target="/enterprise/customer/customerAddress">
      <validTime kind="state" existence="varyingWithGaps"/>
      <itemIdentifier name="addressId1" timeDimension="validTime">
       <field path="."/>
      </itemIdentifier>
     </item>
     <item target="/enterprise/customer/invoice">
      <validTime kind="event"/>
      <transactionTime/>
      <itemIdentifier name="invoiceId1" timeDimension="bitemporal">
       <field path="invoiceNo"/>
      </itemIdentifier>
     </item>
     <item target="/enterprise/equipment/qtyInStock">
      <validTime kind="state" content="varying"/>
      <transactionTime/>
      <itemIdentifier name="qtyId1" timeDimension="bitemporal">
       <field path="."/>
      </itemIdentifier>
     </item>")
```

$Op^{AD}12$: RemoveLogicalAnnotationSubSet(AD.xml, beginningItemPath, endingItemPath)

It removes a subset of logical annotations, delimited by a beginning item (located at "beginningItemPath") and an ending item (located at "endingItemPath"), from the `<logical/>` container of an annotation document "AD.xml".

$Op^{AD}13$: ReplaceLogicalAnnotationSubSet(AD.xml, beginningItemPath, endingItemPath, newLogicalAnnotationSubSet)

It replaces, in an annotation document "AD.xml", a subset of the logical annotations, delimited by a beginning item (located at "beginningItemPath") and an ending item (located at "endingItemPath"), with a new subset of logical annotations ("newLogicalAnnotationSubSet") stored in an XML document.

4.2.3. High-Level Operations dealing with Time-Varying Items

Since a single logical annotation is described by a time-varying item (i.e., an <item/> element in the <logical/> container), we think that high-level operations for changing logical annotations should include operations acting on such a time-varying item. But the <item/> element is a complex one and includes several sub-elements. In fact, it has one attribute (target) and eleven sub-elements (see [1], pages 221-223): <validTime/>, <transactionTime/>, <itemIdentifier/>, <attribute/>, <nonSeqUnique/>, <nonSeqKey/>, <uniqueNullRestricted/>, <nonSeqKeyref/>, <cardConstraint/>, and <transitionConstraint/>. Notice that each one of these sub-elements has also sub-elements (possible with attributes). Moreover, the maximal occurrence of each one of the last seven sub-elements (i.e., <attribute/>, <nonSeqUnique/>, <nonSeqKey/>, <uniqueNullRestricted/>, <nonSeqKeyref/>, <cardConstraint/>, and <transitionConstraint/>) is set to unbounded. Therefore, while taking into account all the information presented above, we propose the following list of high-level operations for defining, removing and changing time-varying items.

$Op^{AD}14$: DefineTimeVaryingItem(AD.xml, itemTarget,
validTimeKind, validTimeContent, validTimeExistence,
validTimeContentVaryingApplicabilityBegin,
validTimeContentVaryingApplicabilityEnd,
validTimeMaximalExistenceBegin, validTimeMaximalExistenceEnd,
validTimeFrequency,
transactionTimeFrequency,
itemIdentifierName, itemIdentifierTimeDimension,
itemIdentifierKeyRefName, itemIdentifierKeyRefType,
itemIdentifierPathField)

It defines, in an annotation document "AD.xml", a new time-varying item for an element (located at "itemTarget" in the conventional schema corresponding to "AD.xml") and having the following properties: validTimeKind, validTimeContent, validTimeExistence, validTimeContentVaryingApplicabilityBegin, validTimeContentVaryingApplicabilityEnd, validTimeMaximalExistenceBegin, validTimeMaximalExistenceEnd, validTimeFrequency, transactionTimeFrequency, itemIdentifierName, itemIdentifierTimeDimension, itemIdentifierKeyRefName, itemIdentifierKeyRefType, and itemIdentifierPathField. This operation inserts a non-empty new <item/> element in the <logical/> container of "AD.xml". It is mapped onto the following list of primitives:

(i) AddItem(AD.xml, itemTarget)

(ii) AddValidTimeToItem(AD.xml, itemTarget, validTimeKind, validTimeContent, validTimeExistence)

(iii) AddContentVaryingApplicabilityToValidTimeInItem(AD.xml, itemTarget, contentVaryingApplicabilityBegin, contentVaryingApplicabilityEnd)

(iv) AddMaximalExistenceToValidTimeInItem(AD.xml, itemTarget, maximalExistenceBegin, maximalExistenceEnd)

(v) AddFrequencyToValidTimeInItem(AD.xml, itemTarget, validTimeFrequency)

(vi) AddTransactionTimeToItem(AD.xml, itemTarget, transactionTimeKind, transactionTimeContent, transactionTimeExistence)

(vii) AddFrequencyToTransactionTimeInItem(AD.xml, itemTarget, transactionTimeFrequency)

(viii) AddItemIdentifierToItem(AD.xml, itemTarget, itemIdentifierName, itemIdentifierTimeDimension)

(ix) AddKeyRefToItemIdentifier(AD.xml, itemTarget, itemIdentifierName, keyRefName, keyRefType)

(x) AddFieldToItemIdentifier(AD.xml, itemTarget, itemIdentifierName, pathField)

Example: Suppose that the designer would like to annotate the element <equipment> through a new logical annotation (i.e., through a new <item> element) in the annotation document "EnterpriseAnnotations.xml". To do this, he/she calls the DefineTimeVaryingItem operation as follows:

```
DefineTimeVaryingItem(EnterpriseAnnotations.xml, "/enterprise/equipment",
                      "state", "varying", "varyingWithGaps", , , , , , ,
                      "equipmentId1", "bitemporal", "equipmentKey", "snapshot", )
```

The new <item> element that will be added to the annotation document "EnterpriseAnnotations.xml" is as follows:

```
<item target="/enterprise/equipment">
 <validTime kind="state" content="varying"
existence="varyingWithGaps"/>
 <transactionTime/>
 <itemIdentifier name="equipmentId1" timeDimension="bitemporal">
  <keyref refName="equipmentKey" refType="snapshot"/>
 </itemIdentifier>
</item>
```

$Op^{AD}15$: RemoveTimeVaryingItem(AD.xml, itemTarget)

It removes, from the annotation document "AD.xml", an existing time-varying item associated to an element located at "itemTarget" in the conventional schema corresponding to "AD.xml". This operation deletes the corresponding <item/> element and all its sub-elements from the <logical/> container. This operation is mapped onto the following list of primitives:

(i) for each <attribute/> sub-element in <item/> element
RemoveAttributeFromItem(AD.xml, itemTarget, attributeName)
(ii) for each <nonSeqUnique/> sub-element in <item/> element
RemoveNonSeqUniqueFromItem(AD.xml, itemTarget, constraintName)
(iii) for each <nonSeqKey/> sub-element in <item/> element
DeleteNonSeqKeyFromItem(AD.xml, itemTarget, constraintName)
(iv) for each <uniqueNullRestricted/> sub-element in <item/> element
RemoveUniqueNullRestrictedFromItem(AD.xml, itemTarget, constraintName)
(v) for each <nonSeqKeyref/> sub-element in <item/> element
DeleteNonSeqKeyRefFromItem(AD.xml, itemTarget, constraintName)
(vi) for each <cardConstraint/> sub-element in <item/> element
DeleteCardConstraintFromItem(AD.xml, itemTarget, constraintName)
(vii) for each <transitionConstraint/> sub-element in <item/> element
DeleteTransitionConstraintFromItem(AD.xml, itemTarget, constraintName)
(viii) DeleteItemIdentifierFromItem(AD.xml, itemTarget, constraintName)

(ix) DeleteTransactionTimeFromItem(AD.xml, itemTarget, constraintName)

(x) DeleteValidTimeFromItem(AD.xml, itemTarget, constraintName)

(xi) DeleteItem(AD.xml, itemTarget, constraintName)

Op^{AD}16: ChangeTimeVaryingItem(AD.xml, itemTarget,
newValidTimeKind, newValidTimeContent, newValidTimeExistence,
newValidTimeContentVaryingApplicabilityBegin,
newValidTimeContentVaryingApplicabilityEnd,
newValidTimeMaximalExistenceBegin, newValidTimeMaximalExistenceEnd,
newValidTimeFrequency,
newTransactionTimeFrequency,
newItemIdentifierName, newItemIdentifierTimeDimension,
newItemIdentifierKeyRefName, newItemIdentifierKeyRefType,
newItemIdentifierPathField)

It changes, in the annotation document "AD.xml", the value of one or more of the properties (i.e., validTimeKind, validTimeContent, validTimeExistence, validTimeContentVaryingApplicabilityBegin, validTimeContentVaryingApplicabilityEnd, validTimeMaximalExistenceBegin, validTimeMaximalExistenceEnd, validTimeFrequency, transactionTimeFrequency, itemIdentifierName, itemIdentifierTimeDimension, itemIdentifierKeyRefName, itemIdentifierKeyRefType, and itemIdentifierPathField) of a time-varying item associated to an element located at "itemTarget" in the conventional schema corresponding to "AD.xml".

Op^{AD}17: ReplaceTimeVaryingItem(AD.xml, itemTarget, newItemTarget
newItemValidTimeKind, newItemValidTimeContent,
newItemValidTimeExistence,
newItemValidTimeContentVaryingApplicabilityBegin,
newItemValidTimeContentVaryingApplicabilityEnd,
newItemValidTimeMaximalExistenceBegin,
newItemItemValidTimeMaximalExistenceEnd,
newItemValidTimeFrequency,
newItemTransactionTimeFrequency,
newItemIdentifierName, newItemIdentifierTimeDimension,
newItemIdentifierKeyRefName, newItemIdentifierKeyRefType,
newItemIdentifierPathField)

It replaces, in the annotation document "AD.xml", a time-varying item (having as a target the element located at "itemTarget" in the conventional schema corresponding to "AD.xml") with a new time-varying item (having the following properties: newItemTarget, newItemValidTimeKind, newItemValidTimeContent, newItemValidTimeExistence, newItemValidTimeContentVaryingApplicabilityBegin, newItemValidTimeContentVaryingApplicabilityEnd, newItemValidTimeMaximalExistenceBegin, newItemValidTimeMaximalExistenceEnd, newItemValidTimeFrequency, newItemTransactionTimeFrequency, newItemIdentifierName, newItemIdentifierTimeDimension, newItemIdentifierKeyRefName, newItemIdentifierKeyRefType, and newItemIdentifierPathField).

Notice that after defining a new time-varying item (through the DefineTimeVaryingItem operation presented above), the designer could add, to this new item, an attribute that is being annotated or a non-sequenced constraint [1,19]. High-level operations for managing attributes and non-sequenced constraints within time-varying items are presented in the subsections 4.2.3.1 and 4.2.3.2, respectively.

4.2.3.1. High-Level Operations dealing with Attributes in Time-varying Items

In conventional schema, attributes of elements (i.e., defined through `<attribute>` components) can

vary over time. However, they cannot be specified as items in an annotation document; an attribute's enclosing data element can be part of an item.

The designer could use the four following high-level operations for managing an <attribute> sub-element in an <item> element.

Op^{AD}18: AddTimeVaryingAttributeToItem(AD.xml, itemTarget, attributeName, validTimeKind, validTimeContent, validTimeContentVaryingApplicabilityBegin, validTimeContentVaryingApplicabilityEnd, validTimeFrequency, transactionTimeFrequency)

It adds a time-varying attribute (having the following properties: attributeName, validTimeKind, validTimeContent, validTimeContentVaryingApplicabilityBegin, validTimeContentVaryingApplicabilityEnd, validTimeFrequency, and transactionTimeFrequency) to a time-varying item (associated to an element located at "itemTarget" in the conventional schema corresponding to the annotation document "AD.xml"). This operation adds an <attribute/> sub-element to an <item/> element.

Example: Suppose that the designer would like to express that the attribute "equipmentCategory" of the element <equipment> is time-varying; thus, he/she should annotate this attribute through a new <attribute> subelement of the <item> element corresponding to the <equipment> element, in the annotation document "EnterpriseAnnotations.xml". To do this, he/she calls the AddTimeVaryingAttributeToItem operation as follows:

```
AddTimeVaryingAttributeToItem(EnterpriseAnnotations.xml, "/enterprise/equipment",
                              "equipmentCategory", "state", "varying", , , , )
```

The <attribute> element that will be added to the annotation document "EnterpriseAnnotations.xml" is as follows:

```
<attribute name="equipmentCategory">
 <validTime kind="state" content="varying"/>
 <transactionTime/>
</attribute>
```

Furthermore, in the annotation document "EnterpriseAnnotations.xml", the <item> element corresponding to the <equipment> element will become as follows:

```
<item target="/enterprise/equipment">
 <validTime kind="state" content="varying"
existence="varyingWithGaps"/>
 <transactionTime/>
 <itemIdentifier name="equipmentId1" timeDimension="bitemporal">
  <keyref refName="equipmentKey" refType="snapshot"/>
 </itemIdentifier>
 <attribute name="equipmentCategory">
  <validTime kind="state" content="varying"/>
  <transactionTime/>
 </attribute>
</item>
```

Op^{AD}19: RemoveTimeVaryingAttributeFromItem(AD.xml, itemTarget, attributeName)

It removes a time-varying attribute (having the name "attributeName") from the time-varying item associated to an element located at "itemTarget" in the conventional schema corresponding to the annotation document "AD.xml". This operation removes an `<attribute/>` sub-element from an `<item/>` element.

Op^{AD}20: ChangeTimeVaryingAttributeInItem(AD.xml, itemTarget, attributeName, validTimeKind, validTimeContent, validTimeContentVaryingApplicabilityBegin, validTimeContentVaryingApplicabilityEnd, validTimeFrequency, transactionTimeFrequency)

It changes the value of one or more of the properties (i.e., attributeName, validTimeKind, validTimeContent, validTimeContentVaryingApplicabilityBegin, validTimeContentVaryingApplicabilityEnd, validTimeFrequency, and transactionTimeFrequency) of a time-varying attribute (having the name "attributeName") in a time-varying item (associated to an element located at "itemTarget" in the conventional schema corresponding to the annotation document "AD.xml").

Op^{AD}21: ReplaceTimeVaryingAttributeInItem(AD.xml, itemTarget, attributeName, newAttributeName, newAttributeValidTimeKind, newAttributeValidTimeContent, newAttributeValidTimeContentVaryingApplicabilityBegin, newAttributeValidTimeContentVaryingApplicabilityEnd, newAttributeValidTimeFrequency, newAttributeTransactionTimeFrequency)

It replaces a time-varying attribute (having the name "attributeName") with a new time-varying attribute (having the following properties: newAttributeName, newAttributeValidTimeKind, newAttributeValidTimeContent, newAttributeValidTimeContentVaryingApplicabilityBegin, newAttributeValidTimeContentVaryingApplicabilityEnd, newAttributeValidTimeFrequency, and newAttributeTransactionTimeFrequency), in a time-varying item (associated to an element located at "itemTarget" in the conventional schema corresponding to the annotation document "AD.xml").

4.2.3.2. High-Level Operations dealing with Non-sequenced Constraints in Time-varying Items

A non-sequenced constraint is a temporal constraint specified on an element or an attribute, as a logical annotation within an item. It is evaluated over a time interval (i.e., over some part or the whole of the applicability bound) rather than at each point in time separately. For example, a non-sequenced unique (or key) constraint requires that the constrained element (or attribute) is unique over time (not just at a point in time).

Each non-sequenced constraint is specified in the logical annotations through a subelement of the `<item>` element.

Furthermore, a non-sequenced constraint could be one of the following six types:

- a non-sequenced unique constraint (specified through the `<nonSeqUnique>` subelement);
- a non-sequenced key constraint (specified through the `<nonSeqKey>` subelement);
- a non-sequenced unique constraint with null value restrictions (specified through the `<uniqueNullRestricted>` subelement);

- a non-sequenced referential integrity constraint (specified through the <nonSeqKeyref> subelement);
- a non-sequenced cardinality constraint (specified through the <cardConstraint> subelement);
- a non-sequenced transition constraint (specified through the <transitionConstraint> subelement).

In each subsection of the following six subsections, we present the proposed high-level operations acting on each type of non-sequenced constraints.

A) High-Level Operations acting on Non-sequenced Unique Constraints

$Op^{AD}22$: AddNonSeqUniqueConstraintToItem(AD.xml, itemTarget, constraintName, conventionalIdentifier, dimension, evaluationWindow, slideSize, applicabilityBegin, applicabilityEnd, selector, field)

It adds a non-sequenced unique constraint (having the following properties: constraintName, conventionalIdentifier, dimension, evaluationWindow, slideSize, applicabilityBegin, applicabilityEnd, selector, and field) to a time-varying item (associated to an element located at "itemTarget" in the conventional schema corresponding to the annotation document "AD.xml").

Example: Let us resume our example of Figure 18. We have a conventional unique constraint defined on the attribute "customerNo" of the element <customer>. Since a conventional unique constraint does not imply non-sequenced uniqueness, the same "customerNo" could be re-used for another customer or changed between snapshots (for the same customer, as long as it remains unique).

Suppose that the designer would like to specify that the value of the attribute "customerNo" is unique across a temporal document (with snapshots coalesced across the window of evaluation). To do this, he/she defines a non-sequenced unique constraint on the attribute "customerNo" by calling the AddNonSeqUniqueConstraintToItem operation as follows:

```
AddNonSeqUniqueConstraintToItem(EnterpriseAnnotations.xml, "/enterprise/customer",
                "nonSeqUniqCustNo", , , "year", "day", "2013-01-01", , ".", "@customerNo")
```

The <nonSeqUnique> subelement that will be added to the annotation document "EnterpriseAnnotations.xml" is as follows:

```
<nonSeqUnique name="nonSeqUniqCustNo" evaluationWindow="year"
              slideSize="day" >
 <applicability begin="2013-01-01" />
 <selector xpath="." />
 <field xpath="@customerNo" />
</nonSeqUnique>
```

Furthermore, in the annotation document "EnterpriseAnnotations.xml", the <item> element corresponding to the <customer> element will become as follows:

```
<item target="/enterprise/customer">
 <validTime kind="state" content="varying"
existence="varyingWithGaps"/>
 <transactionTime/>
 <itemIdentifier name="customerId1" timeDimension="bitemporal">
  <field path="@customerNo"/>
 </itemIdentifier>
```

```
 <nonSeqUnique name="nonSeqUniqCustNo" evaluationWindow="year"
               slideSize="day">
  <applicability begin="2013-01-01" />
  <selector xpath="." />
  <field xpath="@customerNo" />
 </nonSeqUnique>
</item>
```

$Op^{AD}23$: RemoveNonSeqUniqueConstraintFromItem(AD.xml, itemTarget, constraintName)

It removes a non-sequenced unique constraint (having the name "constraintName") from the time-varying item associated to an element located at "itemTarget" in the conventional schema corresponding to the annotation document "AD.xml".

$Op^{AD}24$: ChangeNonSeqUniqueConstraintInItem(AD.xml, itemTarget, constraintName, newName, newConventionalIdentifier, newDimension, newEvaluationWindow, newSlideSize, newApplicabilityBegin, newApplicabilityEnd, newSelector, newField)

It changes the value of one or more of the properties (i.e., constraintName, conventionalIdentifier, dimension, evaluationWindow, slideSize, applicabilityBegin, applicabilityEnd, selector, and field) of a non-sequenced unique constraint (having the name "constraintName") in a time-varying item (associated to an element located at "itemTarget" in the conventional schema corresponding to the annotation document "AD.xml").

$Op^{AD}25$: ReplaceNonSeqUniqueConstraintInItem(AD.xml, itemTarget, constraintName, newConstraintName, newConstraintConventionalIdentifier, newConstraintDimension, newConstraintEvaluationWindow, newConstraintSlideSize, newConstraintApplicabilityBegin, newConstraintApplicabilityEnd, newConstraintSelector, newConstraintField)

It replaces a non-sequenced unique constraint (having the name "constraintName") with a new non-sequenced unique constraint (having the following properties: newConstraintName, newConstraintConventionalIdentifier, newConstraintDimension, newConstraintEvaluationWindow, newConstraintSlideSize, newConstraintApplicabilityBegin, newConstraintApplicabilityEnd, newConstraintSelector, and newConstraintField), in a time-varying item (associated to an element located at "itemTarget" in the conventional schema corresponding to the annotation document "AD.xml").

B) High-Level Operations acting on Non-sequenced Key Constraints

$Op^{AD}26$: AddNonSeqKeyConstraintToItem(AD.xml, itemTarget, constraintName, conventionalIdentifier, dimension, evaluationWindow, slideSize, applicabilityBegin, applicabilityEnd, selector, field)

It adds a non-sequenced key constraint (having the following properties: constraintName, conventionalIdentifier, dimension, evaluationWindow, slideSize, applicabilityBegin, applicabilityEnd, selector, and field) to a time-varying item (associated to an element located at "itemTarget" in the

conventional schema corresponding to the annotation document "AD.xml").

Op^{AD}27: RemoveNonSeqKeyConstraintFromItem(AD.xml, itemTarget, constraintName)

It removes a non-sequenced key constraint (having the name "constraintName") from the time-varying item associated to an element located at "itemTarget" in the conventional schema corresponding to the annotation document "AD.xml".

Op^{AD}28: ChangeNonSeqKeyConstraintInItem(AD.xml, itemTarget, constraintName, newName, newConventionalIdentifier, newDimension, newEvaluationWindow, newSlideSize, newApplicabilityBegin, newApplicabilityEnd, newSelector, newField)

It changes the value of one or more of the properties (i.e., constraintName, conventionalIdentifier, dimension, evaluationWindow, slideSize, applicabilityBegin, applicabilityEnd, selector, and field) of a non-sequenced key constraint (having the name "constraintName") in a time-varying item (associated to an element located at "itemTarget" in the conventional schema corresponding to the annotation document "AD.xml").

Op^{AD}29: ReplaceNonSeqKeyConstraintInItem(AD.xml, itemTarget, constraintName, newConstraintName, newConstraintConventionalIdentifier, newConstraintDimension, newConstraintEvaluationWindow, newConstraintSlideSize, newConstraintApplicabilityBegin, newConstraintApplicabilityEnd, newConstraintSelector, newConstraintField)

It replaces a non-sequenced key constraint (having the name "constraintName") with a new non-sequenced key constraint (having the following properties: newConstraintName, newConstraintConventionalIdentifier, newConstraintDimension, newConstraintEvaluationWindow, newConstraintSlideSize, newConstraintApplicabilityBegin, newConstraintApplicabilityEnd, newConstraintSelector, and newConstraintField), in a time-varying item (associated to an element located at "itemTarget" in the conventional schema corresponding to the annotation document "AD.xml").

C) High-Level Operations acting on Non-sequenced Unique Constraints with Null Value Restrictions

Op^{AD}30: AddNonSeqUniqueNullRestrictedConstraintToItem(AD.xml, itemTarget, constraintName, conventionalIdentifier, nullCountMin, nullCountMax, dimension, evaluationWindow, slideSize, applicabilityBegin, applicabilityEnd, selector, field)

It adds a non-sequenced unique constraint with null value restrictions (having the following properties: constraintName, conventionalIdentifier, nullCountMin, nullCountMax, dimension, evaluationWindow, slideSize, applicabilityBegin, applicabilityEnd, selector, and field) to a time-varying item (associated to an element located at "itemTarget" in the conventional schema corresponding to the annotation document "AD.xml").

Op^{AD}31: RemoveNonSeqUniqueNullRestrictedConstraintFromItem(AD.xml, itemTarget,

constraintName)

It removes a non-sequenced unique constraint with null value restrictions (having the name "constraintName") from the time-varying item associated to an element located at "itemTarget" in the conventional schema corresponding to the annotation document "AD.xml".

$Op^{AD}32$: ChangeNonSeqUniqueNullRestrictedConstraintInItem(AD.xml, itemTarget, constraintName, newName, newConventionalIdentifier, newNullCountMin, newNullCountMax, newDimension, newEvaluationWindow, newSlideSize, newApplicabilityBegin, newApplicabilityEnd, newSelector, newField)

It changes the value of one or more of the properties (i.e., constraintName, conventionalIdentifier, nullCountMin, nullCountMax, dimension, evaluationWindow, slideSize, applicabilityBegin, applicabilityEnd, selector, and field) of a non-sequenced unique constraint with null value restrictions (having the name "constraintName") in a time-varying item (associated to an element located at "itemTarget" in the conventional schema corresponding to the annotation document "AD.xml").

$Op^{AD}33$: ReplaceNonSeqUniqueNullRestrictedConstraintInItem(AD.xml, itemTarget, constraintName, newConstraintName, newConstraintConventionalIdentifier, newConstraintNullCountMin, newConstraintNullCountMax, newConstraintDimension, newConstraintEvaluationWindow, newConstraintSlideSize, newConstraintApplicabilityBegin, newConstraintApplicabilityEnd, newConstraintSelector, newConstraintField)

It replaces a non-sequenced unique constraint with null value restrictions (having the name "constraintName") with a new non-sequenced unique constraint with null value restrictions (having the following properties: newConstraintName, newConstraintConventionalIdentifier, newNullCountMin, newNullCountMax, newConstraintDimension, newConstraintEvaluationWindow, newConstraintSlideSize, newConstraintApplicabilityBegin, newConstraintApplicabilityEnd, newConstraintSelector, and newConstraintField), in a time-varying item (associated to an element located at "itemTarget" in the conventional schema corresponding to the annotation document "AD.xml").

D) High-Level Operations acting on Non-sequenced Referential Integrity Constraints

$Op^{AD}34$: AddNonSeqKeyRefConstraintToItem(AD.xml, itemTarget, constraintName, refer, applicabilityBegin, applicabilityEnd, selector, field)

It adds a non-sequenced referential integrity constraint (having the following properties: constraintName, refer, applicabilityBegin, applicabilityEnd, selector, and field) to a time-varying item (associated to an element located at "itemTarget" in the conventional schema corresponding to the annotation document "AD.xml").

$Op^{AD}35$: RemoveNonSeqKeyRefConstraintFromItem(AD.xml, itemTarget, constraintName)

It removes a non-sequenced referential integrity constraint (having the name "constraintName") from the time-varying item associated to an element located at "itemTarget" in the conventional schema corresponding to the annotation document "AD.xml".

Op^{AD}36: ChangeNonSeqKeyRefConstraintInItem(AD.xml, itemTarget, constraintName, newName, newRefer, newApplicabilityBegin, newApplicabilityEnd, newSelector, newField)

It changes the value of one or more of the properties (i.e., constraintName, refer, applicabilityBegin, applicabilityEnd, selector, and field) of a non-sequenced referential integrity constraint (having the name "constraintName") in a time-varying item (associated to an element located at "itemTarget" in the conventional schema corresponding to the annotation document "AD.xml").

Op^{AD}37: ReplaceNonSeqKeyRefConstraintInItem(AD.xml, itemTarget, constraintName, newConstraintName, newConstraintRefer, newConstraintApplicabilityBegin, newConstraintApplicabilityEnd, newConstraintSelector, newConstraintField)

It replaces a non-sequenced referential integrity constraint (having the name "constraintName") with a new non-sequenced referential integrity constraint (having the following properties: newConstraintName, newConstraintRefer, newConstraintApplicabilityBegin, newConstraintApplicabilityEnd, newConstraintSelector, and newConstraintField), in a time-varying item (associated to an element located at "itemTarget" in the conventional schema corresponding to the annotation document "AD.xml").

E) High-Level Operations acting on Non-sequenced Cardinality Constraints

Op^{AD}38: AddNonSeqCardConstraintToItem(AD.xml, itemTarget, constraintName, restrictionTarget, itemIdentifierRef, dimension, evaluationWindow, slideSize, sequenced, aggregationLevel, minOccurs, maxOccurs, selector, field, applicabilityBegin, applicabilityEnd)

It adds a non-sequenced cardinality constraint (having the following properties: constraintName, restrictionTarget, itemIdentifierRef, dimension, evaluationWindow, slideSize, sequenced, aggregationLevel, minOccurs, maxOccurs, selector, field, applicabilityBegin, and applicabilityEnd) to a time-varying item (associated to an element located at "itemTarget" in the conventional schema corresponding to the annotation document "AD.xml").

Op^{AD}39: RemoveNonSeqCardConstraintFromItem(AD.xml, itemTarget, constraintName)

It removes a non-sequenced cardinality constraint (having the name "constraintName") from the time-varying item associated to an element located at "itemTarget" in the conventional schema corresponding to the annotation document "AD.xml".

Op^{AD}40: ChangeNonSeqCardConstraintInItem(AD.xml, itemTarget, constraintName, newName, newRestrictionTarget, newItemIdentifierRef, newDimension, newEvaluationWindow, newSlideSize, newSequenced, newAggregationLevel, newMinOccurs, newMaxOccurs, newSelector, newField, newApplicabilityBegin, newApplicabilityEnd)

It changes the value of one or more of the properties (i.e., constraintName, restrictionTarget, itemIdentifierRef, dimension, evaluationWindow, slideSize, sequenced, aggregationLevel, minOccurs, maxOccurs, selector, field, applicabilityBegin, and applicabilityEnd) of a non-sequenced cardinality constraint (having the name "constraintName") in a time-varying item (associated to an element located

at “itemTarget” in the conventional schema corresponding to the annotation document “AD.xml”).

Op^{AD}41: ReplaceNonSeqCardConstraintInItem(AD.xml, itemTarget, constraintName newConstraintName, newConstraintRestrictionTarget, newConstraintItemIdentifierRef, newConstraintDimension, newConstraintEvaluationWindow, newConstraintSlideSize, newConstraintSequenced, newConstraintAggregationLevel, newConstraintMinOccurs, newConstraintMaxOccurs, newConstraintSelector, newConstraintField, newConstraintApplicabilityBegin, newConstraintApplicabilityEnd)

It replaces a non-sequenced cardinality constraint (having the name “constraintName”) with a new non-sequenced cardinality constraint (having the following properties: newConstraintName, newConstraintRestrictionTarget, newConstraintIemIdentifierRef, newConstraintDimension, newConstraintEvaluationWindow, newConstraintSlideSize, newConstraintSequenced, newConstraintAggregationLevel, newConstraintMinOccurs, newConstraintMaxOccurs, newConstraintSelector, newConstraintField, newConstraintApplicabilityBegin, and newConstraintApplicabilityEnd), in a time-varying item (associated to an element located at “itemTarget” in the conventional schema corresponding to the annotation document “AD.xml”).

F) High-Level Operations acting on Non-sequenced Transition Constraints

Op^{AD}42: AddNonSeqTransitionConstraintToItem(AD.xml, itemTarget, constraintName, dimension, selector, field, oldValuePair, newValuePair, valueEvolution, applicabilityBegin, applicabilityEnd)

It adds a non-sequenced transition constraint (having the following properties: constraintName, dimension, selector, field, oldValuePair, newValuePair, valueEvolution, applicabilityBegin, and applicabilityEnd) to a time-varying item (associated to an element located at “itemTarget” in the conventional schema corresponding to the annotation document “AD.xml”).

Op^{AD}43: RemoveNonSeqTransitionConstraintFromItem(AD.xml, itemTarget, constraintName)

It removes a non-sequenced transition constraint (having the name “constraintName”) from the time-varying item associated to an element located at “itemTarget” in the conventional schema corresponding to the annotation document “AD.xml”.

Op^{AD}44: ChangeNonSeqTransitionConstraintInItem(AD.xml, itemTarget, constraintName, newName, newDimension, newSelector, newField, oldValuePair, newValuePair, newValueEvolution, newApplicabilityBegin, newApplicabilityEnd)

It changes the value of one or more of the properties (i.e., constraintName, dimension, selector, field, oldValuePair, newValuePair, valueEvolution, applicabilityBegin, and applicabilityEnd) of a non-sequenced transition constraint (having the name “constraintName”) in a time-varying item (associated to an element located at “itemTarget” in the conventional schema corresponding to the annotation document “AD.xml”).

Op^{AD}45: ReplaceNonSeqTransitionConstraintInItem(AD.xml, itemTarget, constraintName,

newConstraintName, newConstraintDimension, newConstraintSelector, newConstraintField, newConstraintOldValuePair, newConstraintNewValuePair, newConstraintValueEvolution, newConstraintApplicabilityBegin, newConstraintApplicabilityEnd)

It replaces a non-sequenced transition constraint (having the name "constraintName") with a new non-sequenced transition constraint (having the following properties: newConstraintName, newConstraintDimension, newConstraintSelector, newConstraintField, newConstraintOldValuePair, newConstraintNewValuePair, newConstraintValueEvolution, newConstraintApplicabilityBegin, and newConstraintApplicabilityEnd), in a time-varying item (associated to an element located at "itemTarget" in the conventional schema corresponding to the annotation document "AD.xml").

4.3. High-level Operations for Changing Physical Annotations

We have defined ten high-level operations for changing physical annotations. We organize them into three categories: (i) operations that act on the whole physical annotation set, presented in the subsection 4.3.1, (ii) operations that act on a portion of a physical annotation set, described in the subsection 4.3.2, and (iii) operations that act on a physical timestamp, presented in the subsection 4.3.3.

4.3.1. High-Level Operations dealing with the whole Physical Annotation Set

$Op^{AD}46$: InsertPhysicalAnnotationSet(AD.xml, physicalAnnotationSet)

It adds, in an annotation document "AD.xml", a set of physical annotations ("physicalAnnotationSet": a data of string type). Since the ordering of `<stamp/>` elements, in the `<physical/>` container, is unimportant, this operation adds the new set of physical annotations (i.e., the new set of `<stamp/>` elements) at the end of the existing set of physical annotations, within the `<physical/>` container. If this latter does not exist in "AD.xml", this operation first creates it and then inserts the new set of physical annotations.

Example: Suppose that the designer would like to add a set of physical annotations in the annotation document "EnterpriseAnnotations.xml". To do this, he/she calls the InsertPhysicalAnnotationSet operation as follows:

```
InsertPhysicalAnnotationSet(EnterpriseAnnotations.xml,
  "<physical>
    <stamp target="//enterprise" dataInclusion="expandedVersion">
     <stampKind timeDimension="transactionTime" stampBounds="step"/>
    </stamp>
    <stamp target="/enterprise/customer"
dataInclusion="expandedVersion">
     <stampKind timeDimension="bitemporal" stampBounds="extent"/>
    </stamp>
    <stamp target="//price" dataInclusion="expandedVersion">
     <stampKind timeDimension="bitemporal" stampBounds="extent"/>
    </stamp>
  </physical>
  ")
```

Op^{AD}47: RemovePhysicalAnnotationSet(AD.xml)

It removes the set of physical annotations (i.e., all sub-elements of the <physical/> element) from an annotation document "AD.xml".

Op^{AD}48: ReplacePhysicalAnnotationSet(AD.xml, newPhysicalAnnotationSet.xml)

It replaces, in an annotation document "AD.xml", the existing set of physical annotations with a new set of physical annotations ("newPhysicalAnnotationSet") stored in an XML document.

4.3.2. High-Level Operations dealing with a Portion of Physical Annotation Set

Op^{AD}49: InsertPhysicalAnnotationSubSet(AD.xml, physicalAnnotationSubSet)

It adds, in an annotation document "AD.xml", a subset of physical annotations ("physicalAnnotationSubSet": a data of string type). Since the ordering of <stamp/> elements, in the <physical/> container, is unimportant, this operation adds the new subset of physical annotations (i.e., the new subset of <stamp/> elements) at the end of the existing set of physical annotations, within the <physical/> container. If this latter does not exist in "AD.xml", this operation first creates it and then inserts the subset of physical annotations.

Example: Suppose that the designer would like to add a subset of physical annotations in the annotation document "EnterpriseAnnotations.xml". To do this, he/she calls the InsertPhysicalAnnotationSubSet operation as follows:

```
InsertPhysicalAnnotationSubSet(EnterpriseAnnotations.xml,
    "<stamp target="//customerAddress" dataInclusion="expandedVersion">
      <stampKind timeDimension="validTime" stampBounds="extent"/>
    </stamp>
    <stamp target="//invoice" dataInclusion="expandedVersion">
      <stampKind timeDimension="bitemporal" stampBounds="extent"/>
    </stamp>
    <stamp target="//qtyInStock" dataInclusion="expandedVersion">
      <stampKind timeDimension="bitemporal" stampBounds="extent"/>
    </stamp>")
```

Op^{AD}50: RemovePhysicalAnnotationSubSet(AD.xml, beginningStampPath, endingStampPath)

It removes a subset of physical annotations, delimited by a beginning stamp (located at "beginningStampPath") and an ending stamp (located at "endingStampPath"), from the <physical/> container of an annotation document "AD.xml".

Op^{AD}51: ReplacePhysicalAnnotationSubSet(AD.xml, beginningStampPath, endingStampPath, newPhysicalAnnotationSubSet)

It replaces, in an annotation document "AD.xml", a subset of the physical annotations, delimited by a beginning stamp (located at "beginningStampPath") and an ending stamp (located at "endingStampPath"), with a new subset of physical annotations ("newPhysicalAnnotationSubSet") stored in an XML document.

4.3.3. High-Level Operations dealing with Physical Timestamps

Since a single physical annotation is described by a physical timestamp (i.e., a <stamp/> element in the <physical/> container), we think that high-level operations for changing physical should include operations acting on such a physical timestamp (i.e., on the <stamp/> element). According to the schema of the annotation document presented in [1], the <stamp/> element is a complex one: it has two attributes (target and dataInclusion) and includes three sub-elements (see [1], page 234-235): <stampKind/>, <defaultTimeFormat>, and <orderBy>. Since the maximal occurrence of each one of the sub-elements of the <stamp/> element is set to 1, we could propose high-level operations for managing such an element without problems.

$Op^{AD}52$: SpecifyPhysicalTimeStamp(AD.xml, stampTarget,
stampDataInclusion,
timeDimensionStampKind, stampBoundsStampKind,
pluginStampKindFormat, granularityStampKindFormat,
calendarStampKindFormat, propertiesStampKindFormat,
valueSchemaStampKindFormat,
targetFieldOrderBy, timeDimensionFieldOrderBy)

It defines, in an annotation document "AD.xml", a new physical timestamp for an element or an attribute (located at "stampTarget" in the conventional schema corresponding to "AD.xml") and having the following properties: stampDataInclusion, timeDimensionStampKind, stampBoundsStampKind, pluginStampKindFormat, granularityStampKindFormat, calendarStampKindFormat, propertiesStampKindFormat, valueSchemaStampKindFormat, targetFieldOrderBy, and timeDimensionFieldOrderBy. This operation inserts a non-empty new <stamp/> element in the <physical/> container of "AD.xml". It is mapped onto the following list of primitives:

(i) AddStamp(AD.xml, stampTarget, stampDataInclusion, timeDimensionStampKind, stampBoundsStampKind)

(ii) SetFormatInStampKind(AD.xml, stampTarget, pluginStampKindFormat, granularityStampKindFormat, calendarStampKindFormat, propertiesStampKindFormat, valueSchemaStampKindFormat)

(iii) AddOrderByFieldToStamp(AD.xml, stampTarget, targetFieldOrderBy, timeDimensionFieldOrderBy)

Example: Suppose that the designer would like to annotate the element <equipment> through a new physical timestamp (i.e., through a new <stamp> element) in the annotation document "EnterpriseAnnotations.xml". To do this, he/she calls the SpecifyPhysicalTimeStamp operation as follows:

```
SpecifyPhysicalTimeStamp(EnterpriseAnnotations.xml, "/enterprise/equipment",
                         "expandedVersion", "bitemporal", "extent", , , , , , , )
```

The new <stamp> element that will be added to the annotation document "EnterpriseAnnotations.xml" is as follows:

```
<stamp target="/enterprise/equipment" dataInclusion="expandedVersion">
  <stampKind timeDimension="bitemporal" stampBounds="extent"/>
</stamp>
```

Op^{AD}53: RemovePhysicalTimeStamp(AD.xml, stampTarget)

It removes, from the annotation document "AD.xml", an existing physical timestamp associated to an element (or an attribute) located at "stampTarget" in the conventional schema corresponding to "AD.xml". This operation deletes the corresponding <stamp/> element and all its sub-elements from the <physical/> container of "AD.xml".

Op^{AD}54: ChangePhysicalTimeStamp(AD.xml, stampTarget,
newStampDataInclusion,
newTimeDimensionStampKind, newStampBoundsStampKind,
newPluginStampKindFormat, newGranularityStampKindFormat,
newCalendarStampKindFormat, newPropertiesStampKindFormat,
newValueSchemaStampKindFormat, newTargetFieldOrderBy,
newTimeDimensionFieldOrderBy)

It changes, in the annotation document "AD.xml", the value of one or more of the properties (i.e., stampDataInclusion, timeDimensionStampKind, stampBoundsStampKind, pluginStampKindFormat, granularityStampKindFormat, calendarStampKindFormat, propertiesStampKindFormat, valueSchemaStampKindFormat, targetFieldOrderBy, and timeDimensionFieldOrderBy) of a physical timestamp associated to an element (or an attribute) located at "stampTarget" in the conventional schema corresponding to "AD.xml".

Op^{AD}55: ReplacePhysicalTimeStamp(AD.xml, stampTarget,
newStampTarget, newStampDataInclusion,
newTimeDimensionStampKind, newStampBoundsStampKind,
newPluginStampKindFormat, newGranularityStampKindFormat,
newCalendarStampKindFormat, newPropertiesStampKindFormat,
newValueSchemaStampKindFormat, newTargetFieldOrderBy,
newTimeDimensionFieldOrderBy)

It replaces, in the annotation document "AD.xml", a physical timestamp (having as a target the element or the attribute located at "stampTarget" in the conventional schema corresponding to "AD.xml") with a new physical timestamp (having the following properties: newStampTarget, newStampDataInclusion, newTimeDimensionStampKind, newStampBoundsStampKind, newPluginStampKindFormat, newGranularityStampKindFormat, newCalendarStampKindFormat, newPropertiesStampKindFormat, newValueSchemaStampKindFormat, newTargetFieldOrderBy, and newTimeDimensionFieldOrderBy).

5. Related Work Discussion

In [14], the authors introduce a set of high-level evolution primitives that could be used by an XML Schema designer to express complex schema changes; each high-level primitive is expressed as a sequence of atomic primitives which are also proposed in [14]. But due to space limitations, they do not give enough details on these high-level primitives (e.g., which is their complete set and operational semantics, which atomic primitives compose each high-level primitive and in which way). Furthermore, the authors do not refer the reader to any technical report or any other work. They present an example in which they use only two high-level primitives: insert_substruct and collapse_substruct. They note that these high-level primitives mainly include primitives for inserting, moving, and changing whole substructures rather than single elements.

In [15], the authors propose a set of three high-level DTD change operations (i.e., SubtreeMoveUp,

SubtreeMoveDown, and RelationshipInverse) and study their effect on XML documents whose DTD is being evolved. The authors note that this set of high-level operations is based on the set of primitive DTD change operations that was proposed in [16].

In [17], the authors propose a set of composite operations for changing XML schema at conceptual levels (i.e., PIM and PSM levels), within an approach based on the principles of Model-Driven Development. Obviously, each composite operation is a sequence of two or more atomic operations. But the authors give only three composite operations and do not refer the reader to a work in which he/she find more details or the full list of possible composite operations. In fact, the authors suppose that the particular set of composite operations depends on the choice of the vendor of a particular system and the requirements of users. Their aim in [17] was to demonstrate that the proposed mechanism can be used in real-world situations.

Similar to all the works presented above (i.e., [14], [15], and [17]), our current work proposes also high-level schema change operations that are validity preserving (i.e., each operation applied to a consistent conventional schema produces a consistent conventional schema).

But our contributions with regards to [14], [15], and [17], are as follows:

1) We have proposed a large set of high-level operations in order to provide more various user-friendly operations that could be directly implemented by creators of tools or systems for XML Schema evolution/versioning. Notice here that our proposed set of high-level operations is not complete: we do not claim to provide all possible high-level operations, since we could not list all requirements of all XML Schema designers; but, if required, an XML Schema designer could build a new customized high-level operation by composing some low-level operations and/or some high-level operations.
2) We have classified the proposed high-level operations in basic operations (i.e., high-level operations that cannot be defined by means of other basic high-level operations) and complex operations.
3) We have not proposed only operations for manipulating portions of schema (i.e., sub-schema or sub-tree), but we have provided also operations dealing with whole conventional schema, XML Schema elements, XML Schema attributes, and XML Schema constraints (datatype restrictions, and key, unique, cardinality, and referential integrity constraints).

6. Conclusion

In this report, we focused on the problem of changing schema in the τXSchema framework: we tried to deal with what kinds of meaningful schema change operations can be supported, and how the schema changes would be actually effected in terms of primitive operations.

Since a τXSchema schema consists of three components (a temporal schema that ties together a conventional schema and an annotation document), we have proposed three sets of high-level schema change operations and described their semantics; each set is devoted to a different schema component. These operations are user-friendly and help designers to perform suitable changes to schema, by allowing them to express complex schema changes in a more compact way.

As a part of our future work, we intend to develop a tool for schema versioning in the τXSchema framework, that supports these operations and provides them to designers. Their implementation will be based on the low-level operations already proposed in [2,3], [4,5], and [18].We will also study the propagation of changes performed by these operations, i.e., their effects on (i) non-temporal data stored in conventional documents which are valid to conventional schema, and also on (ii) temporal data stored in temporal documents and generated, by the Squash tool [1], from non-temporal data and temporal schema. Obviously, this requires studying first of all propagation of changes made by low-level

operations. Currently, we are focusing on this issue.

Furthermore, we also plan to study schema changes affecting the overall XML schema design style, that is involving change of (i) namespaces and their influence on qualified/unqualified local/global definitions, and (ii) design pattern (Russian doll, Salami, Bologna, Venetian blind, and Garden of Eden).

7. References

[1] Currim F., Currim S., Dyreson C. E., Joshi S., Snodgrass R. T., Thomas S. W., Roeder E., "τXSchema: Support for Data- and Schema-Versioned XML Documents", Technical Report TR-91, TimeCenter, 279 pages, September 2009. <http://timecenter.cs.aau.dk/TimeCenterPublications/TR-91.pdf>

[2] Brahmia Z., Bouaziz R., Grandi F., Oliboni B., "A Study of Conventional Schema Versioning in the τXSchema Framework", Technical Report TR-94, TimeCenter, 29 pages, June 2012. <http://timecenter.cs.aau.dk/TimeCenterPublications/TR-94.pdf>

[3] Brahmia Z., Grandi F., Oliboni B., Bouaziz R., "Versioning of Conventional Schema in the τXSchema Framework", *Proceedings of the 8th International Conference on Signal Image Technology & Internet Systems (SITIS'2012)*, Sorrento – Naples, Italy, 25-29 November 2012, pp. 510-518.

[4] Brahmia Z., Bouaziz R., Grandi F., Oliboni B., "Schema Versioning in τXSchema-Based Multitemporal XML Repositories", Technical Report TR-93, TimeCenter, 25 pages, December 2010. <http://timecenter.cs.aau.dk/TimeCenterPublications/TR-93.pdf>

[5] Brahmia Z., Bouaziz R., Grandi F., Oliboni B., "Schema Versioning in τXSchema-Based Multitemporal XML Repositories", *Proceedings of the 5th IEEE International Conference on Research Challenges in Information Science (RCIS 2011)*, Guadeloupe - French West Indies, France, 19-21 May 2011, pp. 1-12.

[6] Kosky A., "Modeling and Merging Database Schema", Technical Report MS-CIS-91-65, University of Pennsylvania, 1991.

[7] Kosky A., "A Formal Model for Databases with Applications to Schema Merging", *Proceedings of the International Workshop on Specifications of Database Systems*, Glasgow, 3–5 July 1991, pp. 154-170.

[8] Buneman P., Davidson S. B., Kosky A., "Theoretical Aspects of Schema Merging", *Proceedings of the 3rd International Conference on Extending Database Technology (EDBT 1992)*, Vienna, Austria, March 23-27, 1992, pp. 152-167.

[9] Buneman P., Davidson S. B., Kosky A., VanInwegen M., "A Basis for Interactive Schema Merging", *Proceedings of the 25th Annual Hawaii International Conference on System Sciences (HICSS'1992)*, 7-10 January 1992, Kauai, Hawaii, volume 2, pp. 311-322.

[10] Quix C., Kensche D., Li X., "Generic Schema Merging", *Proceedings of the 19th International Conference on Advanced Information System Engineering (CAiSE'2007)*, Trondheim, Norway, 11-15 June 2007, pp. 127-141.

[11] Pottinger R., Bernstein P. A., "Schema merging and mapping creation for relational sources", *Proceedings of the 11th International Conference on Extending Database Technology (EDBT 2008)*, Nantes, France, 25-29 March 2008, pp. 73-84.

[12] Li X., Quix C., Kensche D., Geisler S., "Automatic schema merging using mapping constraints among incomplete sources", *Proceedings of the 19th ACM Conference on Information and Knowledge Management (CIKM 2010)*, Toronto, Ontario, Canada, 26-30 October 2010, pp. 299-

308.

[13] Critchlow T., "Schema Coercion: Using Database Meta-information to Facilitate Data Transfer", PhD thesis in computer science, University of Utah, 1997.

[14] Guerrini G., Mesiti M., Rossi D., “Impact of XML Schema Evolution on Valid Documents”, *Proceedings of the 7th ACM International Workshop on Web Information and Data Management (WIDM 2005)*, Bermen, Germany, 4 November 2005, pp. 39-44.

[15] Prashant B. V. N., Kumar P. S., “Managing XML data with Evolving Schema”, *Proceedings of the 13th International Conference on Management of Data (COMAD’2006)*, Delhi, India, 14-16 December 2006, pp. 174-177.

[16] Su H., Kramer D., Chen L., Claypool K. T., Rundensteiner E. A., “XEM: Managing the evolution of XML Documents”, *Proceedings of the 11th International Workshop on Research Issues in Data Engineering: Document Management for Data Intensive Business and Scientific Applications (RIDE 2001)*, Heidelberg, Germany, 1-2 April 2001, pp. 103-110.

[17] Necaský M., Klímek J., Malý J., Mlýnková I., “Evolution and change management of XML-based systems”, *Journal of Systems and Software*, vol. 85, n° 3, 2012, pp. 683-707.

[18] Brahmia Z., Grandi F., Oliboni B., Bouaziz R., “Schema Change Operations for Full Support of Schema Versioning in the τXSchema Framework”, *International Journal of Information Technology and Web Engineering* (in press).

[19] Currim F., Currim S., Dyreson C. E., Snodgrass R. T., Thomas S. W., Zhang R., “Adding Temporal Constraints to XML Schema”, *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, n° 8, 2012, pp. 1361-1377.