Preserving Conceptual Constraints During XML Updates

ERIC PARDEDE, J. WENNY RAHAYU
Department of Computer Science and Computer Engineering, La Trobe University, Melbourne, Australia
Email: {ekpardede,wenny}@cs.latrobe.edu.au

DAVID TANIAR
School of Business Systems, Monash University, Melbourne, Australia
Email: David.Taniar@infotech.monash.edu.au

Received: December 19 2004; accepted: April 24 2005

***Abstract*— Despite the increasing demand for an effective XML document repository, many are still reluctant to store XML documents in their natural tree form. One main reason is the inadequacy of XML query languages to update the tree-form XML documents. Even though some of the languages have supported minimum update facilities, they do not concern on preserving the documents constraints. The results are updated documents with very low database integrity. In this paper, we propose a methodology to accommodate XML Update without violating the conceptual constraints of the documents. The method takes form as a set of functions that perform checking mechanisms before update operations. In this paper we discuss the conceptual constraints embedded in three different relationship structures: association, aggregation and inheritance relationship. We highlight four constraints related with association relationship (nuber of participants, referential integrity, cardinality, and adhesion), five constraints related with aggregation relationship (cardinality, adhesion, ordering, homogeneity and share-ability) and two constraints related to inheritance relationship (disjoint and number of super-class). In addition, a specific constraint, which is collection type of children, will also be discussed. The proposed method can be implemented in different ways, for example in this paper we use XQuery language. Since the XML update requires schema, in this paper we also propose the mapping of the these constraints in the conceptual level to the XML Schema. We use XML Schema for structure validation, even though the algorithm can be used by any schema languages.**

***Index Terms*— XML updates, conceptual model, XML constraints, XML schema, XQuery**

I. INTRODUCTION

In the last few years the interest in storing XML Documents in the native XML Databases (NXD) has emerged rapidly [1, 14]. The main idea is to store the documents in their natural tree form. However, it is well-known that many users still prefer to use the RDBMS for their document storage. One reason for not using the NXD is the incompleteness of its query language. Many proprietary XML query languages and even W3C-standardized languages still have limitations compared to the Relational Model SQL. One of the important limitations is the lack of support for update operation [14].

Different NXD apply different strategies for XML updates. Very frequently after updates, an XML document loses the conceptual level semantic. Some of the examples include the document that contains many dangling references, loses the key attribute, has unnecessary duplications, etc. To the best of our knowledge, there is no XML query language that has considered the semantic problems emerging from the updates. We suggest this as an important issue to raise and to investigate further.

In this paper we propose a new methodology providing checking mechanisms, which work like triggers in any DBMS. The methods take form as a set of functions that check, for example, deletion of a refereed key node, insertion of duplicated attribute, etc. The functions ensure that the operations do not violate the conceptual constraints of the XML document. The constraints that will be discussed are classified into three main structural relationships: *association*, *aggregation* and *inheritance relationships* [11]

1) **Association** relationship is a 'reference' relationship between one object with another object in a system. For an XML document, the object is a node or a document. Semantically, this relationship type can be distinguished by some constraints such as *number of participant type*, *referential integrity*, *cardinality* and *adhesion.*
2) **Agregation** relationship is a relationship type in which a composite object ("whole") consists of component objects ("parts"). We distinguish several semantic constraints in this relationship type: *cardinality*, *adhesion*, *ordering*, *homogeneity*, and *share-ability*.
3) **Inheritance** relationship is a relationship type in which an object/class inherits the properties from another object/class. The former is called the sub-class and the latter is the super-class. There are two constraints involved in this relationship type: the *exclusive disjunction* and the *number of ancestors*. In addition, we will also cover a complex relationship between a parent node to a collection of children. In this relationship, the relationship not only exists between the parent and each of the children

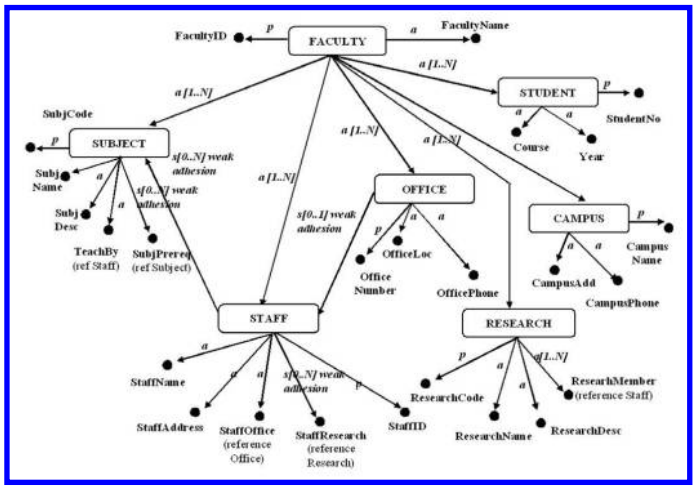

Fig. 1. Association Types in XML Document tree.

separately, but also exists between the children in the collection. The collection can be differentiated based on the ordering and duplication semantics into *set*, *list* and *multiset* [11].

Our proposed update methodology will perform the checking by validating the XML document instance toward a schema. Therefore, we will also need a schema that can capture the conceptual semantic constraints of the XML document. In this paper, we propose the use of XML Schema [10, 16]. It is important to mention that it should not limit the usage of the proposed method. The transformation can be conducted in any standardized or proprietary schema language.

The rest of the paper is structured as follows. Section 2 briefly introduces the conceptual constraints in XML Documents. Section 3 provides the related work on XML Document update including the updates using XQuery language. In section 4 we provide the transformation/mapping method of the XML Document constraints into logical model XML Schema. Section 5 discusses the proposed update methodology utilizing the proposed XML Schema. As an implementation example, we apply the methodology by using W3C-standardized language XQuery. Finally, the paper will be concluded in section 6.

II. Background: Constraints classification in XML documents

In this section we introduce three structural relationships and the collection structure in XML documents. Each of them will have certain semantic constraints that should be preserved after document update. We will also show simple cases that illustrate these constraints. For the modeling we will use Semantic Network Diagram [2] and if necessary UML [12].

A. Association relationships in XML document

Association relationship is a structural relationship, where associated objects are peers, meaning that they are conceptually in the same level. The constraints exist in this relationship type, which are number of participant type, referential integrity, cardinality, and adhesion, and can be shown by using Semantic Network Diagram [2]. We will show a running example describing different association relationship constraints (see Fig. 1).

The number of participants can be distinguished into unary, binary and higher-arity relationships. In the former, an object refers to another object of the same type. For example in Fig. 1, there is one unary relationship in *Subject*. The node *SubjPrereq* in a *Subject* document refers to another subject. Note that the figure has been simplified by not showing the cyclic edge. In the binary relationship, the participants come from different types. In our example, there are three binary relationships: *Subject:Staff*, *Staff:Office*, and *Staff:Research*.

The higher-arity relationship such as ternary relationship is not as common as the unary and binary relationship. An example of such a relationship is *Student:Subject:Campus*. Unfortunately, we cannot use tree-based diagram such as semantic network model to represent the non-binary relationship. Therefore, we still use OO-based modelling diagram, Unified Modeling Language (UML) (see Fig.2).

The referential integrity constraint ensures that the association between the participants are valid and will remain valid after manipulation. For example, a staff will have associated office. The particular office must exist before a staff can be allocated (referred) in it. Therefore, in a staff instance, we need a reference element/attribute to an office instance The next constraint is cardinality. It identifies the number of objects of the same type to which a single object can relate. For example, a staff can teach no subject at all to many subjects [0..N]. In the diagram the default cardinality is [1..1].

Finally, association adhesion identifies whether the associated objects must or must not coexist and adhere to each other. In the diagram the default adhesion is a strong adhesion. For example, *Staff* has weak adhesion association to *Subject*, which means that the former does not require the latter to exist.

B. Aggregation Relationship in XML Document

By nature, XML documents are structured as a set of aggregation relationship. In this relationship, a “whole” component has one or more “part” components related to it. A ”part” component on the other side can only have one “whole” component related to it. In the following figure we illustrate different semantics constraints in aggregation relationship, i.e. cardinality, adhesion, ordering, homogeneity and share-ability. We still use the Faculty XML document as in the previous sub section, with some additional or omission nodes. Except the share-ability, the constraints can be identified in XML data model such as in semantic network diagram (see Fig. 3).

Aggregation cardinality identifies the number of instances of a particular “part” component that a single instance of a “whole” component can relate to. For example in Fig. 3, a faculty has exactly one or more *staff* [1..N] and a *staff* can have none to many *publication* [0..N]. The cardinality default is [1..1].

Aggregation adhesion identifies whether “whole” and “part” components must or must not coexist and adhere to each other. For example, *Faculty* has weak adhesion aggregation

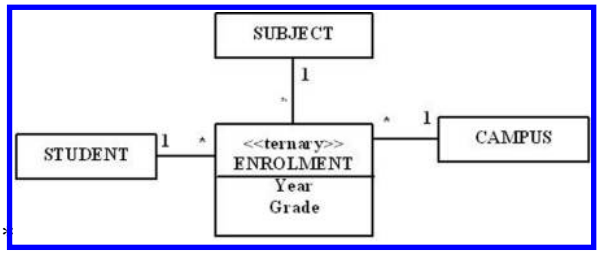

Fig. 2. Ternary Relationship Example in UML.

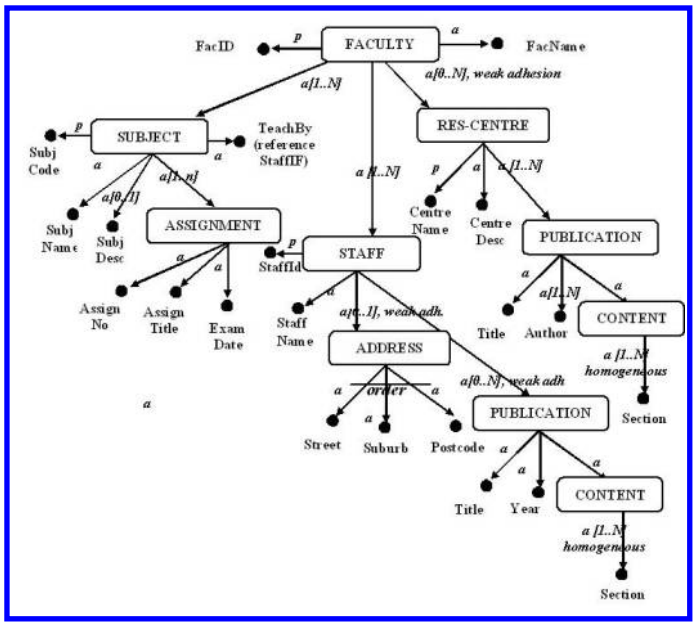

Fig. 3. Aggregation Types in XML Document Tree.

with component *research centre*, which means the existence of the latter does not totally depend on the former. Another weak adhesion example is between a *staff* and an *address*. The default for the diagram is strong adhesion.

Ordered aggregation identifies whether the "part" components must compose the "whole" component in a particular order. The opposite is unordered aggregation, which is usually not explicitly mentioned. In the example, *address* has ordered aggregation.

Aggregation homogeneity identifies whether "whole" component is made by either homogeneous or heterogeneous "part" component type. For example, in the *publication* node the *content* has homogeneous aggregation of *section* document.

Aggregation share-ability identifies whether instance(s) of a "part" component can be shared by more than one instances of one or more "whole" components. If they can be shared, we call it a shareable aggregation. This aggregation type cannot be depicted in any XML conceptual model including the semantic network diagram. Therefore, we use UML model to illustrate this case (see Fig. 4). In this example we assume that the *publication* component is shareable between *staff* and *research centre*.

C. Inheritance relationship in XML document

Inheritance relationship is a relationship that defines an entity/class in terms of another. It organises the classes in taxonomies based on their similarities and differences. The ancestor or super-class holds common information, while the descendants or sub-classes can inherit this information or specify additional contents. Semantically, this relationship can be distinguished by the exclusive disjoint constraint and the number of ancestor.

Exclusive disjoint constraint identifies that in the relationship a super-class can only inherit the common contents to one and only one sub-class type. It means that with this constraint, once there is an instance of a sub-class type, we cannot create another instance from another sub-class type inheriting the super-class content.

This constraint differentiates the inheritance relationship into two types: union inheritance and mutual-exclusion inheritance. The former is the inheritance without the exclusive-disjoint constraint. In this case a super-class can inherit the common content to many instances from different sub-class type. We emphasize that it does not mean the super-class instance is the union of all its sub-class instances. On the other side, the mutual-exclusion inheritance is the inheritance type with exclusive disjoint constraint.

Exclusive disjoint constraint can be easily identified in XML Data Model such as in Semantic Network Diagram (see Fig. 5). In this example a *staff* can only inherit the properties into instances of type *academic* or *management* because there is an exclusive disjoint constraint.

The number of ancestors distinguishes the inheritance relationship into *single inheritance* and *multiple inheritance*. The former indicates that a class can get the properties from only single ancestor, while in the latter, a class can get the properties from more than one ancestors. In Fig. 5, the inheritance between *staff*, *academic* and *management* is a single inheritance.

An example of multiple inheritance is shown below (see Fig. 6). A tutor can inherit the properties from a *staff* and a *student*. Note that we have to use UML to mdel this case, since semantic network diagram does not allow multiple parent or super-class in a XML document.

D. Collection in XML document

Collection in XML document can be mistakenly identified as a simple aggregation. The difference is that in collection there is a constraint between the "part" components. Say now,

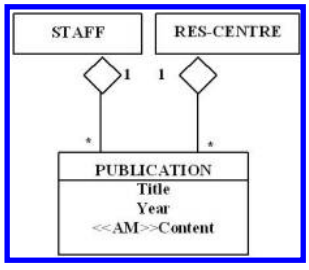

Fig. 4. Non-shareable aggregation.

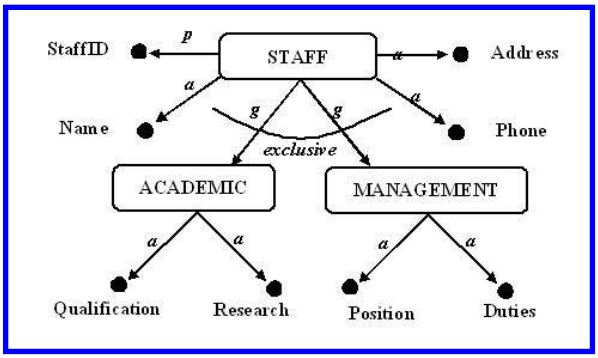

Fig. 5. Inheritance Relationship in XML Document.

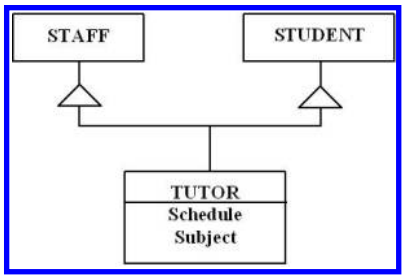

Fig. 6. Multiple Inheritance Relationship.

one group of "part" components has to be a collection of a "whole" component. Therefore, we can identify this structure as a specific type of aggregation relationship.

Based on Object Database Management Group definition, collection existence in XML Document can be differentiated into set, list and multiset. Set is an unordered collection that does not allow duplication. List is an ordered collection that can have duplicated content. Multiset is unordered collection that can have duplicated content.

For example, we use a case study of Auction document. In this database we store the information of some auction details (see Fig. 7.) including the auction date, the location and the auctioneer. In addition, the *Auctions* element also has *AuctionItems* as the child element. A part of *AuctionItems* (*LotNo, LotName, LotPrice, LotContent*) are actually collection element. A specific *LotNo* value must be associated with a certain *LotName*, *LotPrice* and *LotContent* and vice versa. Another collection example is the elements under the *LotContent*. *ItemCondition* for example is associated with a certain *ItemName*, *ItemDesc* and *ItemDimension*.

The first collection in this example is a set collection since the lot detail is unique for each lot number and we do not require any ordering. The second collection on the other side can be a list or a multiset. The item details can be repeated. For example, in the same lot we can have two dinner plates with the same condition, description and dimension. If we assume the collection has to be ordered based on the condition (say A for pristine condition to E for poor condition) we have a list collection. Otherwise, the collection is a multiset.

III. Related work: XML document update

There are several strategies for updating XML documents in NXDs [1, 14]. At the time of writing, there are three main

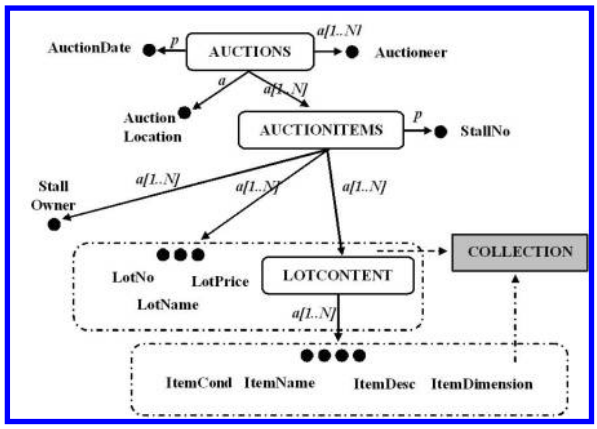

Fig. 7. Collection of Children in XML Document.

strategies. It is important to mention that none of these are concerned with the semantic constraint of the XML document that is being updated.

1) **Proprietary Update Language**. The system can use proprietary update languages that will allow updating within the server. Usually the system has versioning capabilities that enable users to get different versions of the documents. Some of the systems that use this strategy are Ipedo [10] and SODA [13]
2) **XUpdate**. Some systems use XUpdate, the standard proposed by XML DB initiative for updating a distinct part of a document [19]. Some open source NXDs such as Exist and Xindice use this option [5]
3) **XML API**. The other strategy that is followed by most NXD products is by retrieving the XML document and then updated using XML API. After update, the document is returned to the database. One of the systems using this strategy is TIMBER [4].

Different strategies have limited the database interchangeability. To unite these different strategies, [15] has tried to propose the update processes for XML Documents into an XML language. These processes are embedded into XQuery and thus can be used for any NXD that has supported this language.

XQuery is a W3C-standardized language designed for processing XML data. This query language is based on XQuery 1.0 and XPath 2.0 Data Model, which represents the XML document as an ordered, labeled tree in which nodes have identity and may be associated with simple or complex types [18].

To locate the nodes, this language uses path expressions. It has rich expressions to combine and restructure the nodes. The most important expression is the FLWOR expressions. It consists of five clauses. FOR and LET clauses associate variables to expressions, WHERE clause filters the binding results generated by FOR and LET clauses using certain predicates, ORDER BY clause sorts the binding results in a binding stream, and RETURN clause builds the results (see Fig. 8a).

When it was first introduced, XQuery was only used for document query or retrieval. [15] introduces some primitive operations embedded in XQuery for XML document update.

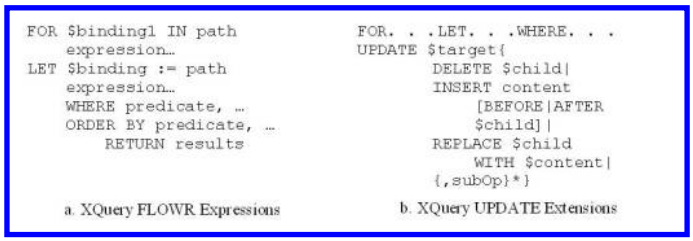

Fig. 8. FLOWR in XQuery Language.

XML UPDATE	Operation			Standard	Preserve Semantics
	Delete	Insert	Replace		
Proprietary language	✓	✓	✓	✗	✗
XML Update	✓	✓	✗	✓	✗
XML API	✓	✓	✗	✗	✗
Proposed method	✓	✓	✓	✓	✓

Fig. 9. Comparison of the Related Works with the Proposed Methodology.

The update is applicable for ordered and unordered XML Documents and also for single or multiple level of nodes.

Fig. 8b depicts the XQuery update extensions. The UPDATE clause specifies the node targets that will be updated. Inside the UPDATE clause, we determine the operation types. Notice that we can have nested update operations.

Nonetheless, even with this proposal there is a basic question to answer. We do not know how the update operations can affect the semantic correctness of the updated XML Documents. This fact has motivated us to take the topic a step further. The following figure shows the comparison of the existing strategy with our methodology. The first column shows the operation types provided by a different strategy. Our proposed method can accommodate three different operations. The second column shows whether the strategy utilizes any W3C-standardized features. In this case, our proposed method can be applied using XQuery standard language. The third column shows whether the strategies are concerned with preserving the semantics of the documents. In this case our proposed method is the only one that has preserved the semantics.

IV. Proposed Transformation Methodology

During XML update checking, we will require a schema to validate the document structure. Among different schema languages (XML-Data, DDML, DCD, SOX, RDF, DTD) [6], we select to use *XML Schema*. It has high precision in defining the data type, as well as an effective ability to define the semantic constraints of a document.

A. Mapping association relationship to XML schema

As it is mentioned before, we can identify four constraints related to association relationship. We will discuss the transformation methodology to capture each constraint using bullet points below.

1) ***Cardinality****:* constraint preservation is very straightforward. In XML Schema, we can determine the ”minoccurs” and ”maxoccurs” after an element declaration. It is a general practice that these constraints are not used for attribute. A particular attribute will appear, at most, once in an object/instance.

Adhesion	Cardinality	What to Use
Strong	1:1	Attribute (use = “required”)
	1:N	Element (minoccurs=“1” maxoccurs=“unbounded”)
	N:N	Element (minoccurs=“<2„ >” maxoccurs=“unbounded”)
Weak	0:1	Attribute (use = “optional”)
	0:N	Element (minoccurs=“0” maxoccurs=“unbounded”)

Fig. 10. Adhesion and Cardinality Constraints in XML Schema.

For example, faculty has to have one and only one name. If we use an element, we can determine the cardinality constraints as is shown below. Faculty also has to have one and only one ID. For an attribute, we do not use the same cardinality constraints. Instead, we just employ ”use” constraint to ”required”.

```
<xs:element name = "faculty">
  <xs:complexType>
    <xs:element name="FacultyName" type="xs:string"
        minoccurs="1" maxoccurs="1">
  ...
  <xs:attribute name="FacultyID" type="xs:string"
      use="required"/>
</xs:complexType>
</xs:element>
```

2) ***Adhesion****:* constraint can easily be indicated by the “use” constraint. A strong adhesion has use “required” and a weak adhesion has use “optional”. However, in practice the “use” constraint can only be attached in attribute. We will find a problem if, for example, the adhesion constraint is applied to an element.

Fortunately, we can utilize the cardinality constraint to define the adhesion constraint as well. Cardinality [0..1] or [0..N] indicates a weak adhesion, while [1..1] or [1..N] indicates a strong adhesion. For the example above, both name and ID have strong cardinality to a *Faculty* element.

For a consensus we provide a table combining the two constraints that can be very useful for the next section.

3) ***Referential integrity****:* in XML Schema can be provided by two options ID/IDREF and key/keyref. We have chosen to use the latter for several reasons: (i) key has no restriction on the lexical space [16], (ii) key cannot be duplicated and thus facilitate the unique semantic of a key node, and (iii) key/keyref can be defined in a specific element and, thereby, helping the implementer to maintain the reference.

For our design, we propose to create a specific element under the root element to store all keys and keyrefs. It is required to track the path of the associated elements once an update is performed. We cannot just declare the key/keyref inside the element where the key/keyref attributes or elements appeared. We will experience problems in tracking an associated element since keyref can only trace the key declared in the same element or in its direct predecessor. For example, if we declare the keyref *TeachBy* inside *Subject* element and key *StaffID* inside *Staff* element, the keyref cannot trace the key and thus, if we update a staff, there is no checking done to ensure the subject refers to the particular staff. Below is the example of an element name “GroupKey” that stores the keys and keyrefs of all nodes under the root node faculty.

```
<xs:element name = "faculty">
```

```
    ...
    <xs:element name="GroupKey" type="xs:string"
        minoccurs="1" maxoccurs="1">

    <key name="SubjCode">
      <xs:selector xpath="Faculty/Subject"/>
      <xs:field xpath="@SubjCode"/>
    </key>
    <keyref name="SubjPrereq" refer="SubjCode">
      <xs:selector xpath="Faculty/Subject"/>
      <xs:field xpath="@SubjPrereq"/>
    </keyref>
    ...
  </xs:element>...
</xs:element>
```

*4) **Number of Participants Type**:* will determine the location of the keyref attribute/element. In a unary relationship, the key and keyref attribute/element is located under one element. For the example (see Fig. 1), the *SubjCode* and *SubjPrereq* are located under *Subject* element. It will be very easy (and also convenient for the system) to also declare the key/keyref under the *Subject* element and not under the "GroupKey" element. However, for the sake of uniformity the declaration will still be under the "GroupKey" element. By doing this, we can use the same functions for all association types in section 5. In addition, there is a possibility of another keyref in the document that might refer to key *SubjCode*. XML Schema below shows the unary relationship under *Subject* element. Note that the previous code for key/keyref declaration is also required.

```
<xs:element name = "faculty">
    ...
    <xs:element name="Subject" minoccurs="1"
        maxoccurs="unbounded">
      <xs:complexType>
        <xs:element name="SubjName" type="xs:string">
        <xs:element name="SubjDesc" type="xs:string">
        <xs:element name="SubjPrereq" type="xs:string"
            minoccurs="0" maxoccurs="unbounded">
        <xs:element name="TeachBy" type="xs:string"
            minoccurs="0" maxoccurs="1">">
        <xs:attribute name="SubjCode" type="xs:string"
            use="required"/>
    </xs:complexType>
  </xs:element name>...
</xs:element>
```

In a binary relationship the adhesion constraint is used to determine the location of the keyref attribute/element. Locate the keyref attribute/element under the associated element that has strong adhesion towards the other in, for example (see Fig. 1), and note the association relationship between a subject and a staff. The staff does not strongly adhere to the subject, but subject does to a staff. In another words, a subject must have associated staff as the teacher/lecturer. Therefore, the reference, *TeachBy*, should be located under subject element. Note that the keyref declaration itself is still defined under "GroupKey" element.

```
<xs:element name = "faculty">

    <xs:element name="Subject" minoccurs="1"
        maxoccurs="unbounded">
      ...
        <xs:attribute name="TeachBy" type="xs:string"
            use="required"/>
      ...
    </xs:element name>
    <xs:element name="Staff" minoccurs="1"
        maxoccurs="unbounded">
      ...
        <xs:attribute name="staffID" type="xs:string"
            use="required"/>
    ...
  </xs:element>...
</xs:element>
```

In the case where both associated elements are in strong adhesion, the keyref attribute/element will be added in both elements. Note that the keyref declarations are still located under "GroupKey" element. An example (see Fig.1) is the association between staff and research. We add reference *ResearchMember* inside research element that refers to *StaffID*, and reference *StaffResearch* inside research element that refers to *ResearchCode*.

```
<xs:element name = "faculty">
    ...
    <xs:element name="Staff" minoccurs="1"
        maxoccurs="unbounded">
      <xs:complexType>
        ...
          <xs:element name="StaffResearch" type="xs:string"
              minoccurs="0" maxoccurs="unbounded">
          <xs:attribute name="staffID" type="xs:string"
              use="required"/>
        ...
  </xs:element>
  <xs:element name="Research" minoccurs="1"
      maxoccurs="unbounded">
    <xs:complexType>
      ...
      <xs:element name="ResearchMember" type="xs:string"
          minoccurs="1" maxoccurs="unbounded">
      <xs:attribute name="ResearchCode" type="xs:string"
          use="required"/>
      ...
  </xs:element>...
</xs:element>
```

The last mapping is for ternary relationship (see Fig. 2). Following the example, a ternary relationship of *Student:Subject:Campus* cannot be changed with the relationships *Student:Subject*, *Student:Campus* and *Subject:Campus*. A ternary relationship is not necessarily equal to three binary relationships.

Our proposal is by creating another child element under the root node to store the ternary relationship. All participating elements will become the reference elements under this new element. Note that we use attribute to ensure that the relationship element has all necessary reference from the participating elements.

```
<xs:element name="Faculty">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Subject" minOccurs="0"
          maxOccurs="unbounded"/>
      <xs:element ref="Student"/>
      <xs:element ref="Campus"/>
      <xs:element name="Enrolment" type="EnrolmentType"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:complexType name="EnrolmentType">
  <xs:sequence>
    <xs:element name="Year" type="xs:gYear"/>
    <xs:element name="Grade" type="xs:string"/>
  </xs:sequence>
  <xs:attribute name="SubjectRef" type="xs:string"
      use="required"/>
  <xs:attribute name="StudentRef" type="xs:string"
      use="required"/>
  <xs:attribute name="CampusRef" type="xs:string"
      use="required"/>
</xs:complexType>

<xs:element name="Enrolment" type="EnrolmentType">
```

To implement the referential integrity we use key/keyref instead of ID/IDREF.

```
<xs:element name="GroupKey">
  <xs:key name="SubjName">
    <xs:selector xpath="Faculty/Subject"/>
    <xs:field xpath="SubjName"/>
  </xs:key>
  <xs:key name="CampusName">
    <xs:selector xpath="Faculty/Campus"/>
    <xs:field xpath="CampusName"/>
  </xs:key>
  <xs:key name="StudentNo">
    <xs:selector xpath="Faculty/Student"/>
    <xs:field xpath="StudentNo"/>
  </xs:key>
  <xs:keyref name="CampusRef" refer="CampusName">
    <xs:selector xpath="Faculty/Enrolment"/>
    <xs:field xpath="CampusRef"/>
  </xs:keyref>
  <xs:keyref name="StudentRef" refer="StudentNo">
    <xs:selector xpath="Faculty/Enrolment"/>
    <xs:field xpath="StudentRef"/>
  </xs:keyref>
  <xs:keyref name="SubjRef" refer="SubjName">
    <xs:selector xpath="Faculty/Enrolment"/>
    <xs:field xpath="SubjRef"/>
  </xs:keyref>
</xs:element>
```

B. Mapping aggregation relationship to XML schema

For aggregation relationship, we have identified five semantic constraints and we will discuss the transformation methodology in this section. However, the first two, which is cardinality and adhesion, can also be found in association relationship. Therefore, we will not repeat the mapping for these two constraints.

1) ***Ordering****:* constraint in XML Schema is defined using "sequence". The ordering is only applicable for the element and not the attribute. In general practice, the attribute is located after all the elements of the same "whole" component. For example, the elements *street*, *suburb*, and *postcode* in an *address* are in a specific order.

```
<xs:element name="address" minoccurs="0"
    maxoccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="street" type="xs:string">
      <xs:element name="suburb" type="xs:string">
      <xs:element name="postcode" type="xs:number">
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

2) ***Homogeneity****:* constraint is not treated by using a unique specification in XML Schema. In fact, if we only declare a single element under a "whole" component, it has shown the homogenous aggregation. For example, the component *section* under a *content* is a homogenous aggregation.

```
 <xs:element name="content">
  <xs:complexType>
      <xs:element name="section" type="xs:string"
          minoccur="1" maxoccurs="unbounded">
  </xs:complexType>
</xs:element>
```

3) ***Share-ability****:* constraint in XML Schema can be accommodated by having the "part" components separately and then linking them with the "whole" components. For example (see Fig. 4), we can assume that the *publication* component is shareable between *staff* and *research centre*. Then, it will be treated as an ordinary association relationship.

```
<xs:element name="Staff" type="StaffType"/>
<xs:complexType name="StaffType">
  <xs:sequence>
  ...
     <xs:element name="Publication" type="PublicationType"
         minOccurs="0" maxOccurs="unbounded"/>
  ...
</xs:complexType>

<xs:element name="ResearchCentre"
    type="ResearchCentreType"/>

<xs:complexType name="ResearchCentreType">
  <xs:sequence>
  ...
    <xs:element name="Publication" maxOccurs="unbounded">
  ...
</xs:complexType>

<xs:complexType name="PublicationType">
  <xs:sequence>
    <xs:element name="Title" type="xs:string"/>
    <xs:element name="Year" type="xs:gYear"/>
    <xs:element name="Content" type="ContentType"/>
  </xs:sequence>
</xs:complexType>
```

C. Mapping Inheritance Relationship to XML Schema

In XML Schema the inheritance can be represented by using "extension". Firstly, we have to define the super-class. Secondly, inside a sub-class element, we determine the extension base, which is the super-class. For example (see Fig. 5), the sub-classes academic and management are both extended from the staff super-class.

```
<xs:complexType name = "staff-TYPE">
  <xs:element name="Address" type="xs:string">
  <xs:attribute name="StaffID" type="xs:string"
      use="required"/>
</xs:complexTtype>

<xs:complexType name = "academic-TYPE">
  <xs:complexContent>
    <xs:extension base="staff-TYPE">
      <xs:element name="Qualification" type="xs:string">
      <xs:element name="Research" type="xs:string">
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name = "management-TYPE">
  <xs:complexContent>
    <xs:extension base="staff-TYPE">
      <xs:element name="Position" type="xs:string">
      <xs:element name="Duties" type="xs:string">
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Nevertheless, this mapping does not allow for the constraints associated with an inheritance relationship. Therefore, we propose another mapping below.

1) ***Exclusive Disjoint****:* constraint in XML Schema can be enforced by using "choice" inside a complex type. Under the "choice" constraint, we can specify what elements that a complex type will have and specify whether it is a union or a mutual-exclusion inheritance.

For the purpose, we create another type that will have elements of the sub-classes type. For example in the following XML Schema we create a new complex type *new-staff-TYPE* with elements academic and management. In this new complex type, we also require the key/ID of the *Staff-TYPE*. It is important to accommodate the update process checking in section 5.

```
union inheritance
<xs:complexType name = "new-staff-TYPE">
  <xs:element name="Academic" type="academic-TYPE">
  <xs:element name="Management" type="management-TYPE">
  <xs:attribute name="StaffID" type="xs:string"
   use="required"/>
</xs:complexTtype>

mutual-exclusion inheritance
<xs:complexType name = "new-staff-TYPE">
  <xs:choice minoccurs="0">
    <xs:element name="Academic" type="academic-TYPE">
    <xs:element name="Management" type="management-TYPE">
  <xs:choice>
  <xs:attribute name="StaffID" type="xs:string"
   use="required"/>
</xs:complexTtype>
```

Every time we want to create a sub-class instance we should create it through the element *New-Staff*, which has *new-staff-TYPE* instance. Note that we also need to declare the key using key/keyref mechanism.

*2) **Number of ancestor**:* constraint differentiates inheritance into single and multiple. A single inheritance is the instance of any new type (in our example *new-staff-TYPE*). The schema below shows the single inheritance instance

```
Single inheritance instance
<xs:element name="New_Staff" type="new-staff-TYPE">
</xs:element>
<key name="StaffID">
  <xs:selector xpath="Faculty/New-Staff"/>
  <xs:field xpath="@StaffID"/>
</key>
```

A problem arises if we want to map a multiple inheritance, for example (see Fig. 6), if we want to declare a complex type tutor under the complex types staff and student. At the time of writing a complex type can only have a single extension base.

We propose another approach to handle the limitation. A sub-class will not have extension bases. Instead we will implement the super-classes as the sub-class's child element. Following the example (see Fig. 6), the sub-class Tutor will have child elements Staff and Student.

```
Multiple inheritance instance
<xs:element name = "Staff" type="staff-TYPE">
<xs:element name = "Student" type="student-TYPE">
<xs:element name = "tutor">
  <xs:complexType>
    <xs:choice minoccurs="0">
      <xs:element ref="Staff"/>
      <xs:element ref="Student"/>
    <xs:choice>
    <xs:element name="Schedule" type="xs:string">
    <xs:element name="Subject" type="xs:string">
  <xs:complexType>
</xs:element>
```

D. Mapping collection of children to XML schema

As mentioned before, there are three types of collection that can be identified in an XML document. The mapping into schema will be different depending on the type of collection. The summary is shown in the following table. Note that we can also use group container if we only have single collection.

*1) **Set Collection**:* can actually be implemented as a new complex element with a unique key to disable the duplication. For the example (see Fig. 7), we can have a complex element *Lot* under the element *AuctionItems*. The current element (*LotName, LotPrice, LotContent*) can become the children of this new complex element.

Options	Collection Types		
	Set	List	Multiset
New child element	✔ Use key or unique for duplication	✔ Use selected element for ordering semantic only	✔ Does not need key or unique constraint
Union/List data types	✘ No support for no duplication	✘ No support for ordering	✔ List for single type, union for multiple
Group container	Only for single Collection		

Fig. 11. Mapping Options for Collection in Aggregation Relationship.

```
<xs:element name="Auctions" maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Auctioneer" type="xs:string"
          maxOccurs="unbounded"/>
      <xs:element name="AuctionLocation" type="xs:string"/>
      <xs:element name="AuctionItems" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="StallOwner" type="xs:string"
                maxOccurs="unbounded"/>
            <xs:element ref="Lot" maxOccurs="unbounded"/>
          </xs:sequence>
          <xs:attribute name="StallNo" type="xs:integer"
              use="required"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="AuctionDate" type="xs:date"
        use="required"/>
  </xs:complexType>
</xs:element>

<xs:complexType name="LotType">
  <xs:sequence>
  <xs:element name="LotNo" type="xs:integer"/>
    <xs:element name="LotName" type="xs:string"/>
    <xs:element name="LotPrice" type="xs:string"/>
    <xs:element name="LotContent" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="Lot" type="LotType"/>
```

Note that under *LotType* there is a maximum of one *LotNo*, *LotName* and *LotPrice*. By doing this we preserve the collection semantic among these elements. For no duplication feature (set collection), we use key for the *LotType* as shown below.

```
<xs:element name="GroupKey">
...
  <xs:key name="LotNo">
  <xs:selector xpath="AntiqueCentre/Auctions/
                      AuctionItems/Lot"/>
  <xs:field xpath="LotNo"/>
</xs:key>
```

*2) **List Collection**:* can also be implemented using a separate new element. The difference with the set collection is no key required for a list because we need to enable the duplication. A specific element will be selected to determine the order and we usually locate it firstly in the sequence. For the example (see Fig. 7), we can have a complex element *LotItems* under the element *Lot*. The current element (*ItemCondition, ItemName, ItemDesc, ItemDimension*) can become the children of this new complex element with *ItemCondition* become the order-deciding element

```
<xs:complexType name="LotType">
```

```
  <xs:sequence>
    <xs:element name="LotNo" type="xs:integer"/>
    <xs:element name="LotName" type="xs:string"/>
    <xs:element name="LotPrice" type="xs:string"/>
    <xs:element ref="LotContent" type="LotContentType"
        maxOccurs="unbounded">
  </xs:sequence>
</xs:complexType>

<xs:complexType name="LotContentType">
  <xs:sequence>
    <xs:element name="ItemCondition" type="xs:string"
        maxOccurs="unbounded"/>
    <xs:element name="ItemName" type="xs:string"/>
    <xs:element name="ItemDesc" type="xs:string"/>
    <xs:element name="ItemDimension" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="LotContent" type="LotContentType"/>
```

*3) **Multiset Collection**:* is the collection with the least semantic constraint. We can have a separate element to represent this collection as we have shown for the set collection. However, since we want to enable the duplication, we just remove the key or unique constraint for this element.

We can also use the group container as shown in the list collection. The only difference is we do not include the *ItemCondition* in the group because we do not need the ordering feature.

The last alternative is by using *xs:list* or *xs:union* for the collection. The content of the element will be a white space-separated list of values. If the values are of the same data type, we use xs:list. Otherwise, we use xs:union.

```
<xs:complexType name="LotType">
  <xs:sequence>
    <xs:element name="LotNo" type="xs:integer"/>
    <xs:element name="LotName" type="xs:string"/>
    <xs:element name="LotPrice" type="xs:string"/>
    <xs:element name="LotContent" type="ItemList"
        maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="Lot" type="LotType"/>

<xs:simpleType name="ItemList">
  <xs:list itemType="xs:string"/>
</xs:simpleType>
```

The above example assumes that all elements under *LotContent* are of string data type. If for example the last element is of integer data type, we can use union like it is shown below

```
<xs:simpleType name="ItemList">
  <xs:union memberTypes="xs:string xs:string
      xs:string xs:integer"/>
</xs:simpleType>
```

*4) **Group Container**:* is a building block in XML Schema (*xs:group*) that can group the collection elements together to become local or global complex type. It can be a good container for collection type. However, there is a major problem that limits its usage for collection storage. We can only use it for a single collection. This is because, in this instance, the group's elements are actually implemented in the same hierarchy as the group's siblings. Therefore, for multiple collection we might lose track on which group's elements are belonging to a specific element.

On the example below, we create a group container that has all of the collection elements under *LotContent* (see Fig. 7). In the XML Schema, this group is refereed by the *LotContent* element.

```
<xs:complexType name="LotType">
  <xs:sequence>
    <xs:element name="LotNo" type="xs:integer"/>
    <xs:element name="LotName" type="xs:string"/>
    <xs:element name="LotPrice" type="xs:string"/>
    <xs:element name="LotContent"
        maxOccurs="unbounded">
      <xs:complexType>
        <xs:sequence>
          <xs:group ref="LotItems"
              maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>

<xs:element name="Lot" type="LotType"/>

<xs:group name="LotItems">
  <xs:sequence>
    <xs:element name="ItemCondition" type="xs:string""/>
    <xs:element name="ItemName" type="xs:string"
        maxOccurs="unbounded"/>
    <xs:element name="ItemDesc" type="xs:string"
        maxOccurs="unbounded"/>
    <xs:element name="ItemDimension" type="xs:string"
        maxOccurs="unbounded" />
  </xs:sequence>
</xs:group>
```

V. Proposed Update Methodology

In this section we will propose the update methodology for each relationship to capture the conceptual constraints [7, 8, 9]. The update can be further classified into deletion, insertion and replacement.The methodology then will be implemented as a set of checking functions written in XQuery.

A. Update methodology for association relationship

For association relationship, we concentrate on the updates that involve key/keyref. Other constraints checking (adhesion and cardinality) will only be in the context of key/keyref. The checking for adhesion and cardinality in non key/keyref will be discussed in section 5.2.

*1) **Deletion in Association Relationship**:* Deletion is the simplest update operation, especially if we want to delete an attribute or an element that contains simple data. In this case we only require the binding to the parent and the child. We do not need to perform predicate checking on instances. However, if the attribute and element also have roles as key/keyref the checking on instances is required. This is to maintain the referential integrity constraint.

For our proposed methodology, we differentiate between the operation for key and keyref since we focus on the association relationship. For deletion however, there is no constraint checking required for keyref node since it is treated like simple data content. Thus, we will only show the algorithm for key node deletion using the following flow chart.

For implementation, we use the following XQuery functions used for predicates in key deletion. Function *checkKeyRefNode* checks whether the target key node has keyref node referred to it. This function highlights one reason why we select key/keyref instead of ID/IDREF. Now the implementer can check only the path where the keyref is defined. Since we have

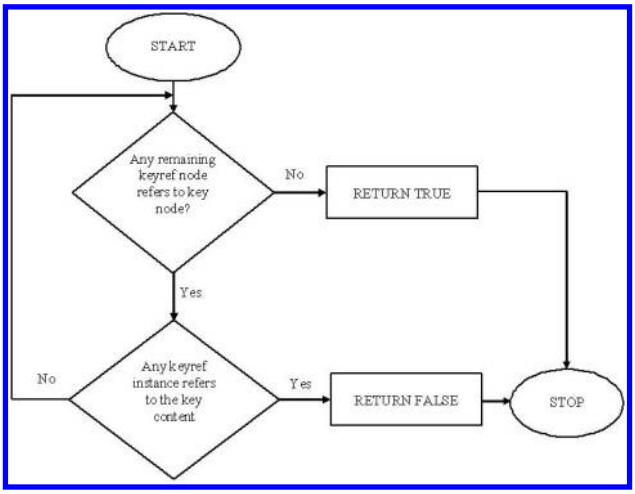

Fig. 12. Key Deletion Method in Association Relationship.

grouped all key and keyref declarations in the "GroupKey" element, we only need to check the "GroupKey" element.

Under a condition (see the function), the *checkKeyRefNode* function proceeds to function *checkKeyRefInstance*. This function checks that there is no instance that has keyref attribute/element referring to the target key content. This is the reason we also need to pass the content of the target key node. If the function returns false, we cannot perform the deletion.

```
FUNCTION checkKeyRefNode($gKBindingPath, $keyName,
 $keyContent) RETURN BOOLEAN {
LET $gkRef := $gKBindingPath/keyref/[@refer]
RETURN
    IF $gkRef = $keyName
       THEN getKeyRefInstance($gKBindingPath,
            $keyName, $keyContent)
       ELSE TRUE}

FUNCTION checkKeyRefInstance($gKBindingPath, $keyName,
        $keyContent) RETURN BOOLEAN
{
FOR $keyRefBinding IN $gKBindingPath/
                      keyref[@refer=$keyname]
LET $keyRefName:=$gKBindingPath/keyref/@name
    $keyRefPath:=$keyRefBinding/selector/xpath
    $keyRefInstance:=$keyRefPath[@$keyRefName,
                                 "$keyContent"]
RETURN
    IF exists ($keyRefInstance)
       THEN FALSE
       ELSE TRUE}

Example:
FOR $g IN document("Faculty.xml")/Faculty/GroupKey
    $p IN document("Faculty.xml")/Faculty/
          Subject(@SubjCode = "ADB41")
    $c IN $p/SubjCode
LET $cContent := "ADB41"
UPDATE $p{
WHERE checkKeyRef($g, "SubjCode", $cContent)
UPDATE $p{
DELETE $p (:delete the key and the siblings:)}}
```

The example following the functions shows the key deletion *SubjCode* in document "Faculty.xml" of instance with *SubjCode* equals to "ADB41". We know that there is a node *SubjPrereq* that refers to *SubjCode*. Thus, if there is any instance where *SubjPrereq* value refers to *SubjCode* "ADB41"', the deletion should be restricted.

Note that if the *checkKeyRef* function returns TRUE, we will delete whole elements and not only the key *SubjCode*. This is because we do not have trigger-based delete, which will delete the whole element if the key is deleted.

*2) **Insertion in Association Relationship**:* Unlike deletion, insertion requires the constructor of a new attribute or new element. Beside the method for key insertion, we have to propose keyref insertion as well, to make sure that the new keyref actually refers to an existent key. Fig. 13 depicts the algorithm for both insertions.

For implementation, function *checkKeyDuplicate* returns TRUE if the target is not duplicating an existing instance. The example following the function shows the checking if we want to insert a *SubjCode* with content "ADB41".

```
FUNCTION checkKeyDuplicate($bindingPath)
         RETURN BOOLEAN
{
RETURN
IF EXISTS($bindingPath)
   THEN FALSE
   ELSE TRUE}

Example:
FOR $g IN document("Faculty.xml")/
          Faculty(@FacID = "FSTE")
    $p IN $g/Subject(@SubjCode="ADB41")
LET $c:= SubjCode
    $cContent:="ADB41"
UPDATE $p{
WHERE checkKeyDuplicate($p)
UPDATE $p{
INSERT new_att($c, $cContent)}}
```

For keyref insertion, we propose three additional functions. The first is for the referential integrity constraint and the following two are for the cardinality constraint. The cardinality is required since a keyref can actually refer to more than one key instance.

Function *checkKeyInstance* passes the keyref name and content, and then checks the key being referred. Function *getMaxOccurs* and *checkMaxOccurs* are used to calculate the maxoccurs constraint of a keyref node. If the particular instance has the maximum number of keyref, we will disable the insertion of another keyref content.

```
FUNCTION checkKeyInstance($gKBindingPath, $keyRefName,
         $keyRefContent) RETURN BOOLEAN
{
FOR $keyRefBinding IN $gKBindingPath/
                      keyref[@name = $keyRefName],
LET $keyName:=$gkeyRefBinding/@refer,
    $keyPath:=$gKeyRefBinding/selector/xpath,
    $keyInstance:=$keyPath[@$keyName, "$keyRefContent")
RETURN
   IF EXISTS($KeyInstance)
      THEN TRUE}

FUNCTION getMaxOccurs($xsName, $parentName, $childName)
         RETURN INTEGER
{
FOR $pDef IN document($xsName)//xs:element
             [@name=$parentName],
    $cDef IN $pDef/xs:element[@name=$childName],
LET $cMaxOccurs:=$cDef/maxoccurs,
RETURN $cMaxOccurs}

FUNCTION checkMaxOccurs($bindingPath, $cMaxOccurs)
         RETURN BOOLEAN
{
LET $instanceOccurs:=count($bindingPath)
RETURN
    IF $cMaxOccurs >= $instanceOccurs + 1
       THEN FALSE
       ELSE TRUE}

Example:

FOR $g IN document("Faculty.xml")/Faculty/GroupKey
    $p IN document("Faculty.xml")/Faculty/
```

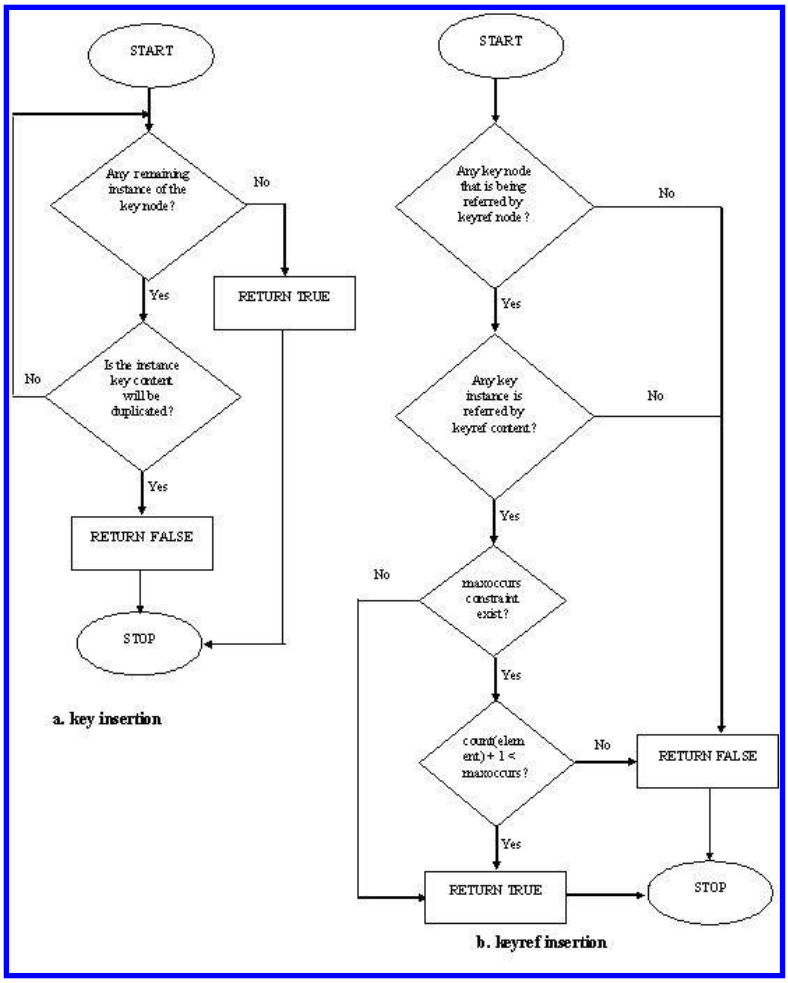

Fig. 13. Key/Keyref Insertion Method in Association Relationship.

```
        Subject(@SubjCode = "ADB41")
    $c IN $p/TeachBy
LET $cName := TeachBy
    $cContent := "MR03"
UPDATE $p{
WHERE checkKeyInstance($g, $cName, $cContent)
UPDATE $p{
LET $getMaxOccurs=("Faculty.xml", "Subject", $cName)
UPDATE $p{
WHERE checkMaxOccurs($p/TeachBy, $getMaxOccurs)
UPDATE $p{
INSERT <TeachBy>MR03</TeachBy>
}}}}
```

The example following the functions shows how we insert a keyref *TeachBy* with value "MR03" in the document. First we have to check whether there is an instance where the key value equals to "MR03". If there is, we check the cardinality. If the instance has not exceeded the maximum cardinality, the insertion is performed.

3) ***Replacement in Association Relationship****:* Since replacement can be seen as a combination of deletion and insertion, we can reuse the functions we have already described in the last two sub-sections. In fact, we do not need the functions for cardinality constraints during keyref replacement. Figure 14 depict the replacement process for key and keyref.

For replacement of a key, we have to check whether the new key content does not duplicate any existing instance. For replacement of a keyref, we just need to check whether the new keyref content refers to a valid instance. No cardinality checking is required since to accommodate a new keyref, we must have deleted another keyref.

XQuery below shows the example of replacement for keyref *TeachBy* element. Note that the query is almost similar to the query for insertion in the previous subsection. The difference is that now we do not require checking the cardinality and the content to check it is the new targeted content.

```
Example:

FOR $g IN document("Faculty.xml")/Faculty/GroupKey
    $p IN document("Faculty.xml")/Faculty/
       Subject(@SubjCode = "ADB41")
    $c IN $p/TeachBy
LET $cName := TeachBy
    $cContent := "WR01"
UPDATE $p{
WHERE checkKeyInstance($g, $cName, $cContent)
UPDATE $p{
REPLACE $c WITH <TeachBy>WR01</TeachBy>
}}
```

B. *Update methodology for aggregation relationship*

In this section we will show the proposed method to capture the document constraints associated with aggregation relationship. Notice that we can also encounter the key/keyref as a

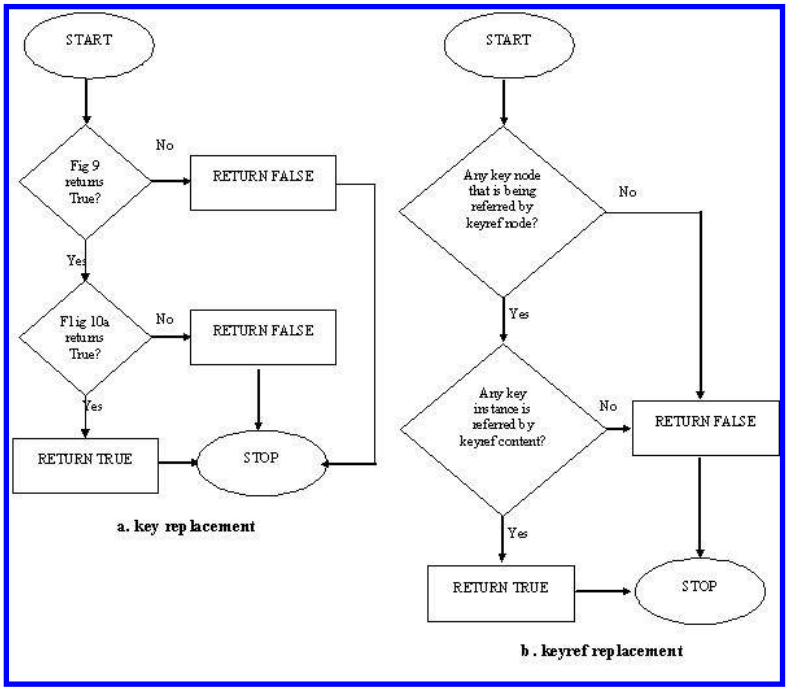

Fig. 14. Key/Keyref Replacement Method in Association Relationship.

"part" component of a "whole" component node. However, since the preservation of this constraint has been discussed in section 5.1, we will not repeat key/keyref "part" component in this section.

*1) **Deletion in Aggregation Relationship**:* The chart (see Fig. 15) shows the proposed checking method for the deletion update. It is divided into three parts. Part A shows the method for key node deletion. If the method returns TRUE, the deletion can be executed since the method has checked the semantics of the document. If it returns FALSE, we restrict the deletion since it will violate any semantics constraints. Part B shows the method for element deletion. The cardinality and adhesion constraint checking is performed. Part C shows the method for attribute deletion. The adhesion constraint checking is performed. Same as Part A, in Part B and Part C the functions might return TRUE or FALSE. The result will determine whether the deletion can be executed or not.

Note that the deletion of a key node (part A) resembles Fig. 12. Therefore, we will not repeat this section further. On the other side, part B and part C show the checking for deletion upon an element and an attribute respectively. In each of these methods, we perform checking for all aggregation constraints we have previously mentioned.

For implementation, the following XQuery shows the functions used for predicates (and binding) in element/attribute deletion. Function *checkChild* returns TRUE if the target node is an element. If it is an element, we proceed to function *getMinOccurs*, which checks and returns the cardinality constraint of the element. Function *checkMinOccurs* compares the value from *getMinOccurs* with the instance actual occurrence. If the deletion violates the cardinality constraint, the operation is restricted.

```
Example:
FUNCTION checkChild($xsName, $parentName, $childName)
         RETURN BOOLEAN
{
FOR $p IN document($xsName)//xs:element[@name=$parentName],
LET $c IN $p/xs:element[@name=$childName],
RETURN
    IF exists($c)
       THEN TRUE
       ELSE FALSE}

FUNCTION getMinOccurs($xsName, $parentName,
         $childName) RETURN INTEGER
{
FOR $p IN document($xsName)//xs:element
         [@name=$parentName],
    $c IN $p/xs:element[@name=$childName],
LET $cminoccurs:=$c/minoccurs,
RETURN $cminoccurs}

FUNCTION checkMinOccurs($cBindingPath, $cminoccurs)
         RETURN BOOLEAN
{
LET $cOccurs:=count($cBindingPath)
RETURN
    IF($cMinOccurs <= $cOccurs - 1)
       THEN FALSE
       ELSE TRUE}

FUNCTION getUse($xsName, $parentName, $childName)
         RETURN BOOLEAN
{
FOR $p IN document($xsName)//xs:element
          [@name=$parentName],
    $c IN $p/xs:attribute[@name=$childName],
LET $cUse:=$c/use,
IF ($cUse = "required"
      THEN FALSE
      ELSE TRUE)}
```

The following example shows the key deletion *subjcode* in document "Faculty.xml". Note that if the *checkKeyRefNode* function returns TRUE, we will delete the whole element and not only the key *subjcode*. This is because we do not have trigger-based delete, which will delete the whole element if the key is deleted.

```
FOR $g IN document("Faculty.xml")/Faculty
         (@FacID = "FSTE")
```

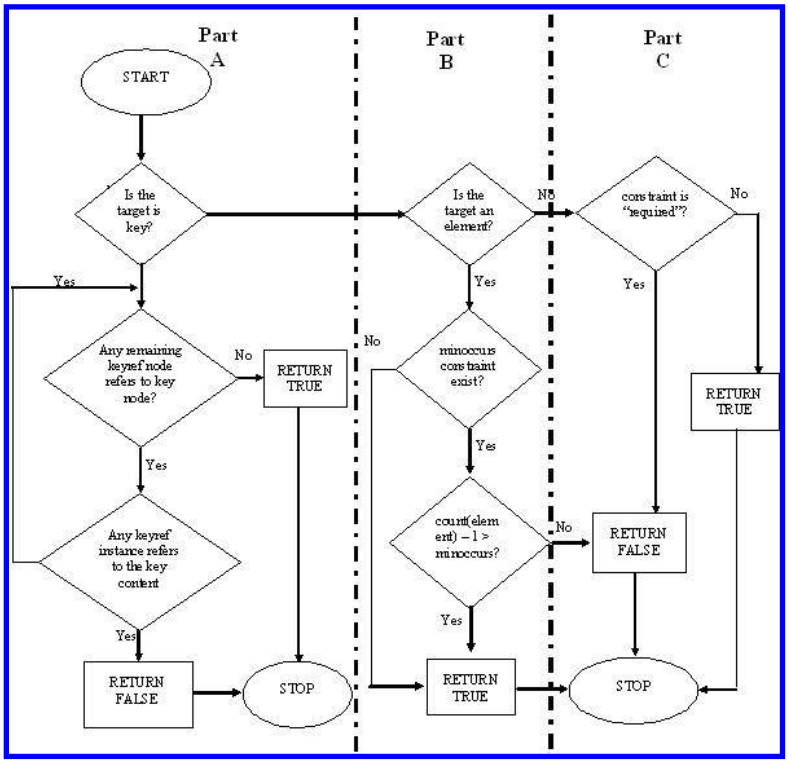

Fig. 15. Methodology for Delete Update in Aggregation Relationship.

```
    $p IN $g/Subject(@SubjCode = "ADB21")
    $c IN $p/SubjCode
LET $cContent := "ADB21"
UPDATE $p{
WHERE checkKey("Faculty.xml", "Faculty", "SubjCode")
UPDATE $p{
WHERE checkKeyRefNode($g, "SubjCode", $cContent)
UPDATE $p{
DELETE $p (:delete the key and the siblings:)
}}}
```

*2) **Insertion in Aggregation Relationship**:* The following chart (see Fig. 16) classifies the checking mechanism into three categories. Part A shows the method for key node insertion. If it returns TRUE, the insertion can be executed and otherwise if it returns FALSE. Part B checks the insertion of keyref node. Part C checks the insertion of simple node content, covering the element node and attribute node. Note that the insertion of a key node (part A) and keyref node (part B) resemble Fig.13a and Fig 13b respectively. Therefore, we will not repeat this section further.

On the other hand, part C shows the checking for insertion upon a simple element or attribute. Function *checkAttInstance* ensures that the target attribute is not duplicated. As for element insertion, we add two additional functions. Function *getMaxOccurs* and *checkMaxOccurs* are used to calculate the *maxoccurs* constraint of a keyref node. If the particular instance has the maximum number of keyref, we will disable the insertion of another keyref content.

```
FUNCTION checkAttInstance($bindingPath, $attName,
         $attContent) RETURN BOOLEAN
{
LET $attInstance:=$bindingPath/$attName(@$attName,
    "$attContent")
RETURN
   IF EXISTS($attInstance)
      THEN FALSE
      ELSE TRUE}

FUNCTION getMaxOccurs($xsName, $parentName,
         $childName) RETURN INTEGER
{
FOR $pDef IN document($xsName)//xs:element
    [@name=$parentName],
    $cDef IN $pDef/xs:element[@name=$childName],
LET $cMaxOccurs:=$cDef/maxoccurs,
RETURN $cMaxOccurs
}

FUNCTION checkMaxOccurs($bindingPath, $cMaxOccurs)
         RETURN BOOLEAN
{
LET $instanceOccurs:=count($bindingPath)
RETURN
    IF $cMaxOccurs >= $instanceOccurs + 1
       THEN FALSE
       ELSE TRUE
}
```

The example below shows how we insert an element *SubjDesc* in the document. Note that in this example we can determine the ordering constraint by using INSERT AFTER. In this case, we insert the node *SubjDesc* after the node *SubjName*.

```
FOR $g IN document("Faculty.xml")/Faculty(@FacID = "FSTE")
    $p IN $g/Subject(@SubjCode="ADB41")
    $l IN $p/SubjName
LET $cName := SubjDesc
    $cContent := "The subject covers the
                  concepts of OR Database and XML Database"
UPDATE $p{
WHERE checkKeyInstance($g, $cName, $cContent)
UPDATE $p{
LET $getMaxOccurs=("Faculty.xml", "Subject", $cName)
UPDATE $p{
WHERE checkMaxOccurs($p/SubjDesc, $getMaxOccurs)
UPDATE $p{
INSERT <SubjDesc> The subject covers the concepts of OR
         Database and XML Database </SubjDesc> AFTER $l
}}}}
```

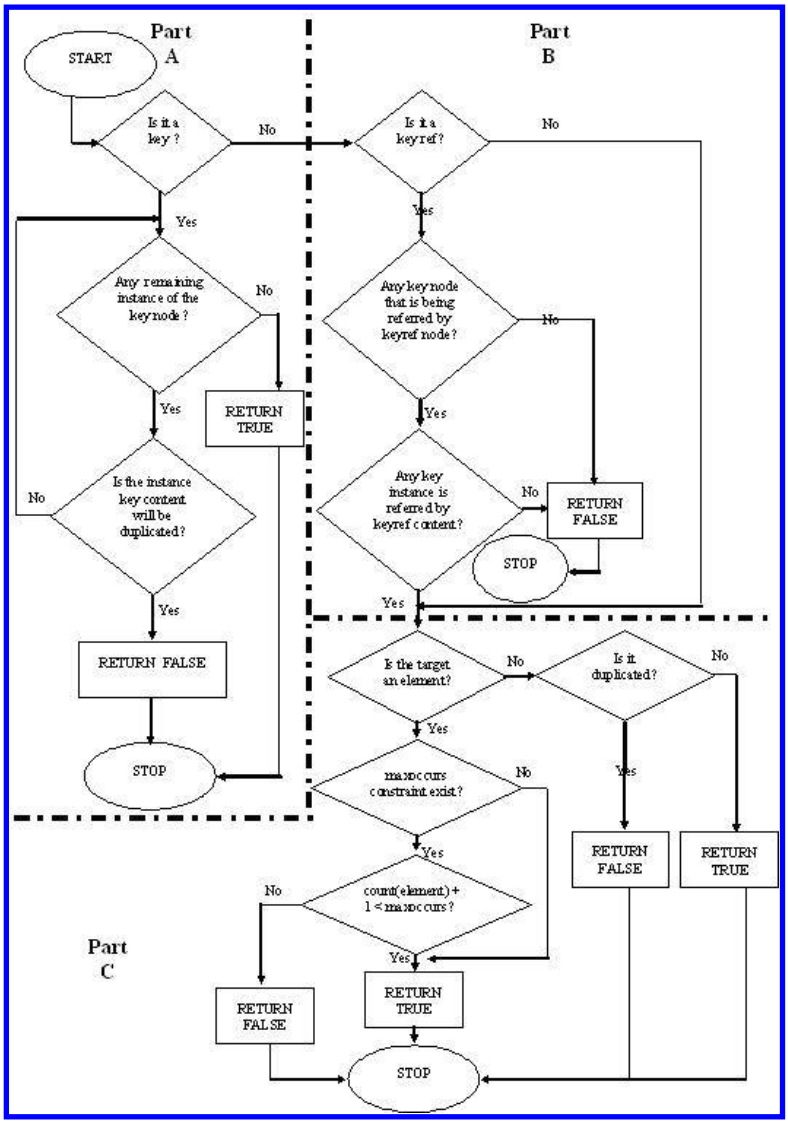

Fig. 16. Methodology for Insert Update in Aggregation Relationship.

*3) **Replacement in Aggregation Relationship**:* The following chart (see Fig. 17) shows the checking for replacement upon a key node (part A) and keyref node (part B), respectively. If the checking mechanism returns TRUE, we can execute the replacement. Otherwise, the replacement is restricted. Note, there is no checking required for simple attribute/element since no semantics violation occurs during replacement of node with simple content.

For implementation using XQuery, we want to replace node *SubjName* into "Foundation of Database System". Since the target is neither a key nor a keyref, we just need to check whether the target is an element or an attribute. Once we know, we can select the appropriate statement.

```
FOR $g IN document("Faculty.xml")/Faculty
        (@Faculty_Name = "FSTE")
    $p IN $g//Subject(@SubjectID = "DB21")
    $c IN $p/subjname

UPDATE $p{
WHERE checkChild("Faculty.xml", "Subject", "subjname")
UPDATE $p{
REPLACE $c WITH <subjname>Foundation of
                Database System</subjname>}
WHERE checkChild("Faculty.xml", "Subject", "subjname")
     = FALSE
UPDATE $p{
REPLACE $c WITH new_att(subjname, "Foundation of
                Database System")}}
```

C. Update methodology for inheritance relationship

In this section we will show the proposed method to capture the document constraints associated with inheritance relationship. In our transformation methodology (section 4.3) we preserve the exclusive disjoint constraint through a new class type and how we associate the super-class and sub-class are through the key of this new class. Therefore, the following update methodology will have some similarities with the methodology for association relationship (section 5.1).

*1) **Deletion in Inheritance Relationship**:* The following chart (see Fig. 18) shows the mechanism for the deletion in an inheritance relationship. It can be applied both for single and multiple inheritance. The difference lies on the target of the checking.

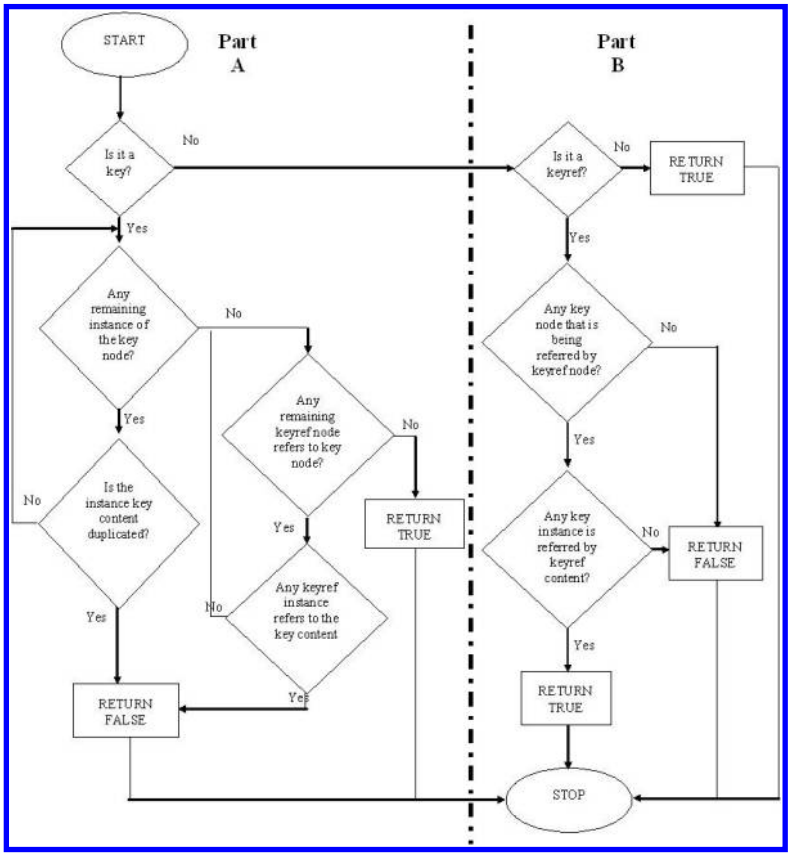

Fig. 17. Methodology for Replace Update in Aggregation Relationship.

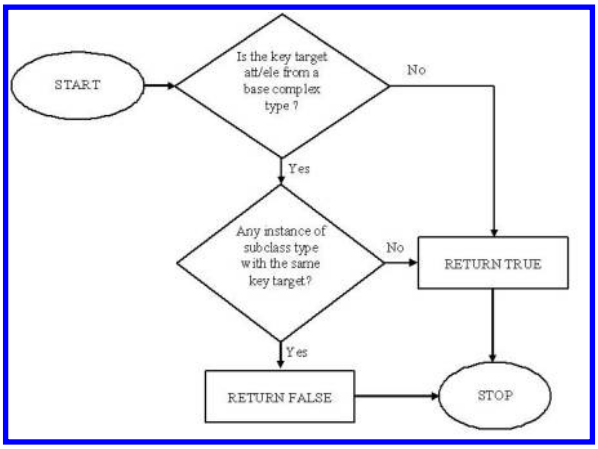

Fig. 18. Checking for Deletion in Inheritance Relationship.

The checking is required if the deletion is conducted upon a super-class instance, where we have to check whether there is any sub-class instance with the same key node. We are not concerned with the deletion of a sub-class instance because the same key node might exist in another sub-class instance or in the super-class instance.

As an example, the following XQuery shows functions used for predicates in the deletion process of a single inheritance. Function *checkSubClassInstance* checks whether any sub-class instance (under *New-Class* element) has the same key node with the super-class key node. If the function returns false, we cannot perform the deletion.

```
FUNCTION checkSubClassInstance($BindingPath, $newSuperClass,
         $cName, $cContent) RETURN BOOLEAN
{
LET $subClassInstance:=$BindingPath/
    $newSuperClass[@$cName, "$cContent"]
RETURN
    IF exists ($subClassInstance)
       THEN FALSE
       ELSE TRUE
}
```

The following example shows the key deletion *StaffID* (equals to “MR01”) in document “Faculty.xml”. We have to check the New-Staff instance with *StaffID* equals to ”MR01”. If it exists, the deletion has to be restricted. Note that if the *checkSubClassInstance* function returns TRUE, we will delete whole elements and not only the key *StaffID*. This is because we do not have trigger-based delete, which will delete the whole element if the key is deleted.

```
FOR $g IN document("Faculty.xml")
    $p $g/Staff(@StaffID = "MR01")
    $c IN $p/StaffID
LET $cName := StaffID
    $cContent := "MR01"
    $newSuperClass := New-Staff
UPDATE $p{
WHERE checkSubClassInstance($g, $newSuperClass,
      $cName, $cContent)
UPDATE $p{
DELETE $p (:delete the key and the siblings:)
}}
```

*2) **Insertion in Inheritance Relationship**:* For insertion, we have to check sub-class instance to avoid the choice constraint violation. On the other hand, we do not have to check

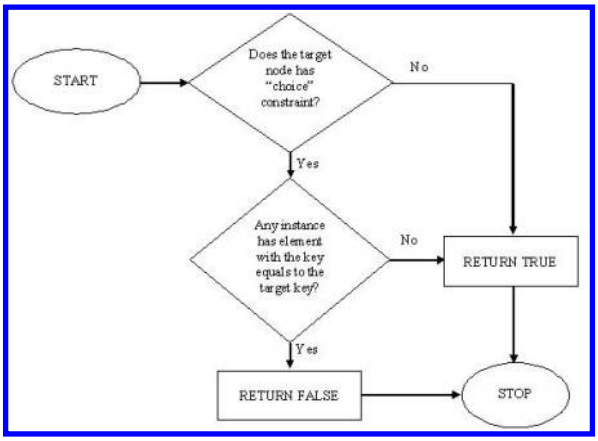

Fig. 19. Checking for Insertion in Inheritance Relationship.

the insertion of a new super-class instance. The following chart (see Fig. 19) depicts the mechanism of the insertion update both applicable for single and multiple inheritance. The first condition separates the union and the mutual-exclusion inheritance.

Function *checkChoiceConstraint* returns TRUE if the target does not have the choice constraint. Otherwise, it will call the function checkSubClassInstance shown in the previous section. The example shows the checking of *StaffID* insertion with content "MR01" under *New-Staff* element. If we assume the *New-Staff* has choice constraint, the *checkSubClassInstance* will be activated and will return true if there is no duplication.

```
FUNCTION checkChoiceConstraint ($BindingPath,
         $newSuperClass, $cName, $cContent) RETURN BOOLEAN
{
LET $choiceConstraint:=$BindingPath/$newSuperClass/choice
RETURN
    IF exists ($choiceConstraint)
       THEN checkSubClassInstance(($BindingPath,
            $newSuperClass, $cName, $cContent)
       ELSE TRUE
}
```

The following example shows the checking of *StaffID* insertion with content "MR01" under *New-Staff* element. If we assume the *New-Staff* has choice constraint, the *checkSubClassInstance* will be activated and will return true if there is no duplication.

```
FOR $g IN document("Faculty.xml")
    $p $g/New_Staff(@StaffID = "MR01")
    $c IN $p/StaffID
LET $cName := StaffID
    $cContent := "MR01"
    $newSuperClass := New-Staff
UPDATE $p{
WHERE checkChoiceConstraint($g, $newSuperClass,
      $cName, $cContent)
UPDATE $p{
INSERT new_att(StaffID, "MR01")
}}
```

*3) **Replacement in Inheritance Relationship**:* As in the association and aggregation relationship, we can utilize the insertion and deletion functions for replacement update. XQuery below shows the example of replacement for *StaffID* element in the *Staff* super-class element. We want to replace the old value "MR01" into "WR01". Note that the query is similar to the query for deletion in the previous subsection. The value to check is the old value, which is "MR01".

```
FOR $g IN document("Faculty.xml")
    $p $g/Staff(@StaffID = "MR01")
    $c IN $p/StaffID
LET $cName := StaffID
    $cContent := "MR01"
    $newSuperClass := New_Staff
UPDATE $p{
WHERE checkSubClassInstance($g, $newSuperClass,
      $cName, $cContent)
UPDATE $p{
REPLACE $C WITH new_attr(StaffID, "WR01")
}}
```

D. Update Methodology for Collection of Children

As it is mentioned in section 4.4, there are three options in XML Schema to accommodate collection of children in an XML document: using new element, group container and union/list data types. We will show how update checking can be employed to preserve the constraint using these three options.

If we use a new element to store a set collection, the checking mechanism will be the same with the methodology mentioned in the previous sections, specifically those that are related with aggregation relationship. This is because by using a new element, the collection will become another simple aggregation relationship. For example, if we update the element *LotPrice*, we know its siblings because they are already located under the same element, which is *Lot*. The constraints checking such as cardinality, adhesion, etc. of this *LotPrice* element will only associate within its own collection. If we do not use the proposed transformation and implement the *LotPrice* as in Fig. 7, we can delete the price of a specific lot name and yet not receive any complaint from the database. If there is at least one price in that stall, the update can still be performed.

If we use a new element for list collection, we can propose an additional update methodology specifically for insertion. Fig. 20 shows the checking mechanism. We have to check whether there is any instance of the same target node. If there is, we have to compare its value with the target node value and then determine whether to insert it after or before the current node.

The implementation of this mechanism using XQuery will be more or less the same as our previous section. Note that before returning TRUE (see Fig. 20), we still have to check the constraint of the particular node such as the cardinality, adhesion, etc. For deletion, the same checking as simple aggregation will be applicable. No additional checking is required.

Finally if we use union/list type for collection storage, there is no checking mechanism that can be proposed at this stage. This implementation (see section 4.4) treats the collection as a simple element with complex content value. If we want to perform a checking mechanism, we need to divide the value content and get the partition that meets our interest. We do not think the constraint preservation benefit is worth the complexity of the process. Therefore, we do not elaborate this method further.

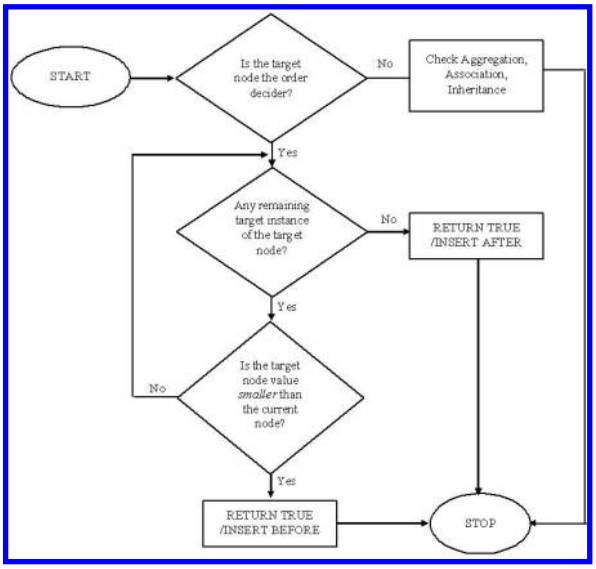

Fig. 20. Checking for Insertion in Collection using Group Container.

VI. Discussions

In section 2 we introduced different conceptual semantic constraints in XML documents. These constraints were classified based on the structural relationship namely association, aggregation and inheritance. Using semantic network diagram [2] and UML [12] we can depict constraints such as number of participant types, referential integrity, adhesion, cardinality, homogeneity, ordering, share-ability, exclusive disjoint and number of ancestor types.

In addition, we also added another complexity in aggregation relationship. It comes when we need the collection constraints among the children of a parent component. Based on the duplication and ordering properties, we differentiate the collection into set, list and multiset. Even though these constraints are widely used in the object-oriented conceptual model [11], very rarely does XML storage application use this semantic.

In section 3 we proposed the transformation methodology of the conceptual model level constraints into a logical model using schema language. This process gave two contributions. First is providing clear procedure for database modelling process. Second is providing a foundation before we can proceed to the XML update methodology. Our update methodology will require the document structure validation that is provided by the schema language.

We have a selection of schema language. We select XML Schema based on its modeling strengths [2, 16]. XML Schema also has features that can capture the relationship constraints depicted in our semantic network diagram.

In our XML Schema, the whole/parent/super-class component is defined as a complex type element and the part/child/sub-class component can take form as an element or an attribute. The component can also be differentiated based on its role in referential integrity into key and keyref.

Despite the limitation of XML Schema, we have tried to utilize the current features to model the constraint. For example, currently XML Schema cannot capture the multiple inheritance. To solve the problem we have to implement the super-classes as the element of the sub-classes. Therefore, if a sub-class instance inherits properties from two super-classes, the instance will have two child elements of the super-class types. There are few other adjustments made. The main idea is to utilize the available language to answer as many problems.

The schema derived from the previous rules is used for the next step. Section 5 discusses the proposed methodology for updating XML document. The method is divided into three operations: deletion, insertion, and replacement. For implementation, we apply the query language XQuery. Not only because it is the most complete W3C-standardized language (W3C, 2001a), but also because it is used by most NXD products [3, 4, 5, 10, 13].

This methodology enables database users to perform some checking mechanisms during the XML update. Now XML Database users have the same flexibility as the Relational Database users in term of update ability. In addition, by using XQuery as the implementation, we can increase the usability of this XML query language and subsequently increase the power of Native XML Database in database community.

VII. Conclusions

Many XML applications still use DBMS based on established data model (such as Relational Model, Object-Relational Model, etc) for the document repository. Among different arguments, the poor capability of XML query languages is one of the most frequently mentioned. The XML query languages lack support for update operations.

In this paper, we propose methodology to preserve semantic constraints during an XML update. The update operations are divided into deletion, insertion, and replacement. Each operation considers different type of target node such as key, key reference, and simple data content whether it is an element or an attribute.

Since update requires a document structure validation, we also propose the transformation of the document conceptual constraints into a schema language. The constraints captured are classified based on the XML structural relationship. In this paper, we use XML Schema for the schema option.

For implementation of the update methodology, we use W3C-standardized query language XQuery. With this extension, XML query languages are becoming more powerful. Concurrently, it can increase the potential of using tree-form XML repository such as Native XML Database.

References

[1] R. Bourett (2003) *XML and Databases*. http://www.rpbourret.com/xml/XMLAndDatabases.htm

[2] L. Feng, E. Chang, T. S. Dillon (2002) A Semantic Network-Based Design Methodology for XML Documents. *ACM Trans. Information System* **20***(4)*: 390–421

[3] Ipedo (2004) *Ipedo XML Database*. http://www.ipedo.com/html/products.html

[4] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakhsmanan, A. Nierman, S. Paprizos, J. M. Patel, D. Srivastava, N. Wiwattana, Y. Wu, C. Yu (2002) TIMBER: A native XML database. *VLDB Journal* **11***(4)*: 279–291

[5] W. M. Meier (2003) eXist Native XML Database. In: A. B. Chauduri, A. Rawais, R. Zicari (eds.), *XML Data Management: Native XML and XML-Enabled Database System*: 43–68. Addison Wesley

[6] D. Mercer (2001) *XML A Beginner's Guide*. Osborne McGraw Hill
[7] E. Pardede, J. W. Rahayu, D. Taniar (2004) On Updating Inheritance Relationship in XML Documents. *Proceedings of iiWAS*: 405–415.
[8] E. Pardede, J. W. Rahayu, D. Taniar (2004) Preserving Referential Constraints in XML Document Association Relationship Update. *Lecture Notes in Computer Science* **3283**: 256–263
[9] E. Pardede, J. W. Rahayu, D. Taniar (2004) Preserving Aggregation Semantic Constraints in XML Document Update. *Lecture Notes in Computer Sciences* **3306**: 229–240
[10] J. Robie (2004) XQuery: A Guided Tour. In H. Kattz (ed.) *XQuery from the Experts*: 3–78. Addison Wesley
[11] J. Rumbaugh et al. (1991) *Object-Oriented Modelling and Design*. Prentice Hall
[12] J. Rumbaugh, I. Jacobson, G. Booch (2004) *The Unified Modeling Language Reference Manual*. Addison Wesley
[13] SODA Technology (2004) *SODA*. http://www.sodatech.com/products.html, 2004
[14] K. Staken (2001) *Introduction to Native XML Databases*. http://www.xml.com/pub/a/2001/10/31/nativexmldb.html
[15] I. Tatarinov, Z. G. Ives, A. Y. Halevy, D. S. Weld (2001) Updating XML. *Proceedings ACM SIGMOD*: 413–424
[16] E. V.-D. Vlist (2002) *XML Schema*. O'Reilly
[17] W3C (2001) *XML Schema*. http://www.w3.org/XML/Schema
[18] W3C (2001) *XQuery 1.0: An XML Query Language*. http://www.w3.org/TR/xquery
[19] XML DB (2002) *XUpdate – XML Update Language*. http://www.xmldb.org/xupdate/

This article has been cited by:

1. Paulo Caetano da Silva, Valéria Cesário Times, Ricardo Rodrigues Ciferri, Cristina Dutra de Aguiar Ciferri. 2014. Analytical Processing Over XML and XLink. *International Journal of Data Warehousing and Mining* **8**:1, 52-92. [CrossRef]
2. Eric Pardede, J. Wenny Rahayu, David Taniar. 2008. XML data update management in XML-enabled database. *Journal of Computer and System Sciences* **74**:2, 170-195. [CrossRef]
3. Xuan Thi Dung, Wenny Rahayu, David Taniar. 2007. A high performance integrated web data warehousing. *Cluster Computing* **10**:1, 95-109. [CrossRef]
4. Eric Pardede, J. Wenny Rahayu, David Taniar. 2006. Object-relational complex structures for XML storage. *Information and Software Technology* **48**:6, 370-384. [CrossRef]
5. Rajugan Rajagopalapillai, Professor Elizabeth Chang, Professor Tharam S. Dillon, Dr Ling Feng. 2006. Modeling views in the layered view model for XML using UML. *International Journal of Web Information Systems* **2**:2, 95-118. [Abstract] [PDF]
6. Mourad Ykhlef. 2006. A logical foundation for nested semi-structured data and web forms. *International Journal of Web Information Systems* **2**:1, 3-18. [Abstract] [PDF]
7. Joseph Fong, San Kuen Cheung, Herbert Shiu, Chi Chung Cheung. 2005. Visualization of XML conceptual schema recovered from XML schema definition. *International Journal of Web Information Systems* **1**:4, 209-222. [Abstract] [PDF]
8. Paulo Caetano da SilvaMultidimensional Data Analysis Based on Links: 212-281. [CrossRef]