Designing Valid XML Views

Ya Bing Chen, Tok Wang Ling, Mong Li Lee

School of Computing, National University of Singapore
(chenyabi, lingtw, leeml)@comp.nus.edu.sg

Abstract

With the increasing number of XML repositories on the World Wide Web, it is essential to provide support for XML views so that users can view the data from different perspective. Existing systems for XML views only support selection operation applied in the views and do not guarantee that the designed views are valid. In this paper, we propose a systematic approach to design valid XML views. There are three main steps in our approach. First, we transform the semistructured XML source documents into a semantically rich *O*bject-*R*elationship-*A*ttribute model designed for *S*emi*S*tructured data (ORA-SS). Second, we enrich the ORA-SS diagram with semantics such as participation constraints of object classes and distinguishing between attributes of object classes and relationship types, which cannot be expressed in the XML document. These additional semantics will allow us to validate XML views subsequently. Third, we use the additional semantics to develop a set of rules to guide the design of valid XML views. We identify four transformation operations for creating XML views, namely, selection, projection, join and swap operations, and develop a comprehensive algorithm that checks for the validity of XML views constructed by applying the 4 operations.

1. Introduction

The eXtensible Markup Language (XML) has become the standard for publishing and exchanging data on the Web. This has resulted in the increasing number of XML repositories. Since different users may want to view the data in these repositories in various ways, it is necessary to provide for views in XML [A99]. Several systems have been proposed to support XML views, including Active Views [AAC+97] and MIX [BGL+99]. The Active Views system defines views using the object-oriented approach, which allows not only data, but also methods. The MIX system integrates heterogeneous data sources and offers views based on the underlying data sources. While both systems provide for the definition of XML views, they do not validate the views that are created. Therefore, there is no guarantee that the views defined are valid.

In this paper, we propose a systematic approach to ensure the validity of XML views. There are three main steps in our approach. First, we transform the semistructured XML source documents into the ORA-SS schema diagram proposed in [DWL+00]. The ORA-SS model is a semantically rich *O*bject-*R*elationship-*A*ttribute model designed for *S*emi*S*tructured data. Second, we enrich the ORA-SS diagram with semantics, such as participation constraints of object classes and distinguishing between attributes of object classes and relationship types, which cannot be expressed in the XML document. These additional semantics will allow us to validate XML views subsequently. Third, based on the enriched ORA-SS schema diagram, we propose a set of rules to guide the design of valid XML views. We develop a comprehensive algorithm that checks for the validity of XML views.

The rest of the paper is organized as follows. Section 2 introduces the background of our work, particularly, the ORA-SS data model and the view definition language used. Section 3 describes our proposed approach to validate XML views. Section 4 discusses related work and we conclude in section 5.

2. Preliminaries

2.1 ORA-SS Data Model

The ORA-SS (Object-Relationship-Attribute model for SemiStructured data) data model comprises of three basic concepts: object classes, relationship types and attributes. An object class is similar to a set of entities in the real world, an entity type in an ER diagram, a class in an object-oriented diagram or an element in XML documents. A relationship type describes a relationship among object classes. Attributes are properties, and may belong to an object class or a relationship type. The ORA-SS data model has four diagrams: the schema diagram, the instance diagram, the functional dependency diagram and the inheritance diagram. A full description of the data model can be found in [DWL+00]. In this paper, we will focus on the schema diagram since it is sufficient for our purposes.

Figure 1 shows an ORA-SS schema diagram containing three object classes – *project*, *supplier* and *part*. An ORA-SS instance diagram of the schema is depicted in Figure 2. In the schema diagram, an object class is represented as a labeled rectangle. A relationship type between two object classes in an ORA-SS schema diagram can be described by *name*, *n*, *p*, *c*, where *name* denotes the name of the relationship type, *n* is an integer indicating the degree of the relationship type (n = 2 indicates binary, n = 3 indicates ternary, etc.), *p* is the participation constraint of the parent object class in the relationship type, and *c* is the participation constraint of the child object class in the relationship type. The participation constraints are defined using the min:max notation.

Attributes are denoted by labeled circles. Keys are filled circles. The attributes of an object class can be distinguished from attributes of a relationship type. The former has no label on its incoming edge while the latter has the name of the relationship type to which it belongs on its incoming edge.

It is clear that ORA-SS is a semantically rich data model. The model not only reflects the nested structure of semistructured data, but it also distinguishes between object classes, relationship types and attributes. In addition, ORA-SS provides for the specification of the participation constraints of object classes in relationship types and distinguishes between attributes of relationship types and attributes of object classes. Such information is lacking in other existing semistructured data models including OEM [AQM+97], XML DTD and XML Schema [XS]. We found that the additional semantics is essential for the verification of the validity of XML views. If the designed view does not violate the semantics, then it is a valid view. Otherwise, it is an invalid view. For this reason, we adopt ORA-SS as the data model for valid XML views design.

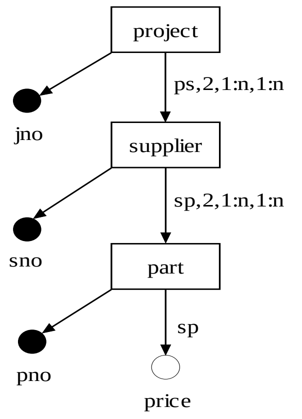

Figure 1: An ORA-SS Schema Diagram on project, supplier and part

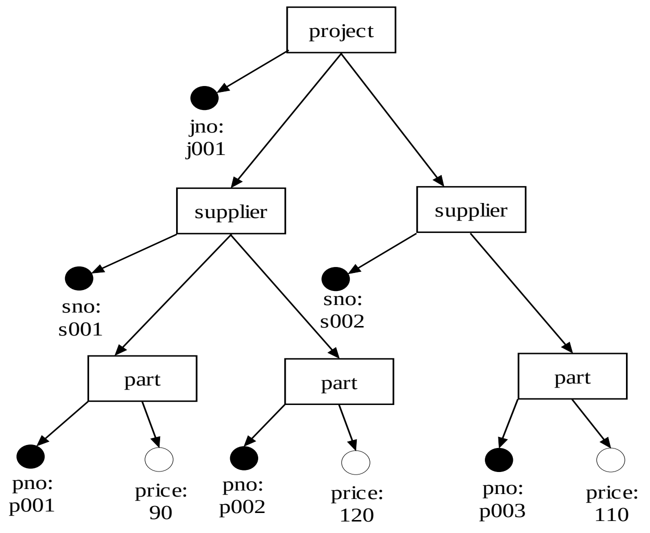

Figure 2: An ORA-SS Instance Diagram of the schema in Figure 1

2.2 View Definition Language

The World Wide Web Consortium recently proposed an XML query language called XQuery [XQ]. XQuery provides flexible query facilities to extract data from real and virtual documents on the Web. The basic form of an XQuery expression consists of For, Let, Where and Return (FLWR) expressions. Although XQuery currently does not provide for the definition of views, we can easily extend it to include the definition of views as follows:

"Create View As view name" followed by FLWR expression.

The For clause and/or Let clause serve to bind values to variables. The values to be bound to variables are represented by path expression. The For clause is used when iteration is needed, while Let clause is used when no iteration is required. The Where clause consists of a set of predicate expressions, which are used to filter the binding-tuples generated by For and Let clauses. The Return clause generates the structure of the view in XML notation. A full description of XQuery can be found in [XQ].

3. Valid XML Views Design

In this section, we will describe our approach to design valid XML views. There are three main steps in our approach:

a) Transform an XML document into an ORA-SS schema diagram.
b) Enrich the ORA-SS schema diagram with necessary semantics.
c) Define a set of rules to guide the design of valid XML views.

We will first illustrate concept of valid and invalid views. Note that the first two steps are actually the pre-cursor to the third step and are relatively straightforward. Therefore, we will only give the rough idea of how the first two steps can be carried out and focus on the third step.

3.1 Motivating Example

Invalid views arise when important semantics are not expressed in the underlying data model. We will illustrate this problem with Figure 3 that describes an XML document conforming to the schema in Figure 1. Note that there exists an implicit functional dependency in the document: *supplier, part* $\rightarrow$ *price*.

A user may use XQuery to design a view that swaps the location of the elements *supplier* and *part.* That is, *supplier* becomes a child of *part* and *part* becomes the parent of *supplier.* As a consequence, we need to decide where to place the element *price*. Since the XML document does not explicitly express the functional dependency: *supplier, part* $\rightarrow$ *price*, the element *price* may be placed under the element *part* in the designed view. This makes *price* an attribute of *part*. Figure 4 shows an instance of the view obtained. A new functional dependency: *part* $\rightarrow$ *price*, now holds in the view that violates the functional dependency *supplier, part* $\rightarrow$ *price* in the source document. We say that such a view is invalid. In order to obtain a valid view, the element *price* should be placed under the element *supplier* so that the original functional dependence is preserved. Figure 5 describes an instance of a valid view.

```
<db>
  <project jno=”j001”>
    <supplier sno=“s001”>
        <part pno=“p001”>
           <price> 100</price>
        </part>
    </supplier>
    <supplier sno=“s002”>
        <part pno=“p001”>
            <price> 100</price>
        </part>
    </supplier>
  </project>
</db>
```

Figure 3: An XML document conforming to the schema in Figure 1

```
<db>
  <project jno=”j001”>
   <part pno=“p001”>
       <price>100</price>
       <supplier sno=“S001”/>
       <supplier sno=“S002”/>
   </part>
  </project>
</db>
```

Figure 4: Invalid view instance of the XML document in Figure 3

```
<db>
  <project jno=”j001”>
   <part pno=“p001”>
       <supplier sno=“S001”>
            <price>100</price>
       </supplier>
       <supplier sno=“S002”/>
            <price>100</price>
        </supplier>
   </part>
  </project>
</db>
```

Figure 5: Valid view instance of the XML document in Figure 3

The above example shows that invalid views can be easily designed if the underlying data model does not express explicitly the necessary semantics. This includes the participation constraints of object classes in relationship types, and distinguishing between attributes of relationship types and attributes of object classes, which are available in the ORA-SS model.

3.2 Transformation of XML into ORA-SS

In this section, we will give a brief outline of the transformation of an XML document into an ORA-SS schema diagram:

- Map root element of the XML document into a root object class in the ORA-SS schema diagram. Note that each XML document has one and only one root element, which will be directly mapped to the root of the ORA-SS schema diagram. For example, the root element *db* in Figure 3 will be mapped into the root object class in ORA-SS.
- Map each element that has attributes or sub-elements into an object class in the ORA-SS schema diagram. Note that an element may occur several times with identical paths. These are actually instances of the same object class. Therefore, we will merge them into only one object class in the ORA-SS schema diagram. For example, the element *supplier* in Figure 3 contains one attribute and will be mapped into an object class. It also occurs twice under one project with the same path */db/project/supplier*. These two instances will be merged into one object class. If two elements in the document have identical name, but have different paths, then they will be mapped into two different object classes.
- Map attributes of an element into the attributes of the object class corresponding to the element. This is a straightforward mapping. If one element is mapped into an object class, then its attributes in the document will be mapped into attributes in the ORA-SS schema diagram. For example, the attribute *sno* of element *supplier* in Figure 3 will be mapped into an attribute *sno* of the object class *supplier* in ORA-SS.
- Map the rest of the elements, which do not have attributes or sub-elements, into attributes of their corresponding parent object classes. For example, the element *price* in Figure 3 will be mapped into an attribute under the object class *part* in ORA-SS.

3.3 Semantic Enrichment of ORA-SS

The ORA-SS diagram obtained from Section 3.2 will basically reflect the tree structure of the XML document and distinguish between object classes and attributes. In order to support the validation of XML views, we need to enrich it with the following additional semantics:

- Identify key attributes of each object class. These key attributes will be represented as filled circles.
- Identify attributes that belong to object classes. The type and cardinality of these attributes should also be determined. A type of an attribute may be composite,

derived or ANY, etc. The cardinality of an attribute may be optional, required or multivalued, etc.

- Identify relationship types among object classes. A label for each relationship type will be put in the diagram. The label contains name, degree and participation constraints of the relationship type.
- Identify attributes of relationship types. Those attributes that are not identified in the previous steps will be labeled so that they belong to relationship types. The text in the label is the name of the relationship type.

These semantics are obtained by user input in this step. Our approach allows user to input necessary semantics directly in the ORA-SS schema diagram.

3.4 Validity of XML Views

After semantically enriching the ORA-SS schema diagram, we can now design XML views and determine its validity. The XML views considered here are created by applying four transformation operations on the source schema. The 4 operations are selection, projection, join and swap. The first three types are analogous to the selection, projection and join in relational databases. The fourth one is unique in XML because it exchanges the positions of parent and child object classes. An XML view may not be simply based on only one of the operations. For example, a view may first apply a selection operation then a join operation. We will now discuss how to guarantee valid XML views design when each operation applies.

3.4.1 Selection Operation

Selection operations basically filter data by using predicates. These are similar to selection operation in relational databases. The structure of source schema remains unchanged and will not cause any changes in the semantics of the source schema. Therefore, if an XML view only applies selection operations, it will be always valid. This operation is also supported in the Active Views system and MIX system.

Example 1
Suppose we want to design a view called *expensive-part* on the ORA-SS source schema diagram defined in Figure 1. This view depicts *projects* for which there exist *suppliers* for which there exist *parts* with a *price* > 80. Within those *projects* it only returns *suppliers* for which there exist *parts* with a *price* > 80. Within those *suppliers* it only returns the *parts* with a *price* > 80. The view definition in XQuery is given in Figure 6. The Let clause in the XQuery binds the variable $p to *parts* whose sub-element *price* is greater than 80. Then the XQuery uses a filter function to return a shallow copy of the nodes that are selected by the arguments, which are the parent of the parent of $p – *project*, the parent of $p – *supplier*, $p itself – *part* and *price*, preserving any relationships that exist among these nodes. Note that the structure of the original document is retained.

```
Create View As expensive-part
Let      $p := document("spj.xml")//part[price>80]
Return  filter( $p/../.. | $p/.. | $p | $p/price )
```

Figure 6: XQuery expression of the view *expensive-part*

Selection operations put predicates on the source schema, which filter data that do not satisfy those predicates. They do not restructure the source document and then have the same schema as the source. Hence, they will not violate semantics in the source schema. As a result, there is no need to set up rules to guarantee the validity of views only applying selection operations. In fact, one can easily design such views using data models such as OEM or XML DTD.

3.4.2 Projection Operation

Projection operations select or drop object classes or attributes in the source schema. They essentially extract a subset structure of the source schema. Since the structure of the source schema is changed, the source semantics may be affected. Therefore it is possible to design an invalid view that violates the semantics in the source schema. This can be detected by designing a set of rules to check for the validity. We will first illustrate how to design a view applying projection operations. Then, we will give the rules to guarantee the validity of such views.

Example 2
Suppose we define a view called *project-part* on the ORA-SS schema diagram in Figure 1. This view removes the intermediate object class *supplier* (see Figure 7). This implies that the attribute *sno* has to be dropped too since attributes cannot exist without its owner object class. Next, we need to remove the relationship types – *js* and *sp*, both of which involves the object class *supplier* which has been removed. The attributes of these relationship types can be dropped too. Alternatively, we can map the attribute of the relationship type *sp* – *price* to an aggregate attribute called *average_price*, which represents the average price of one part in a given project.

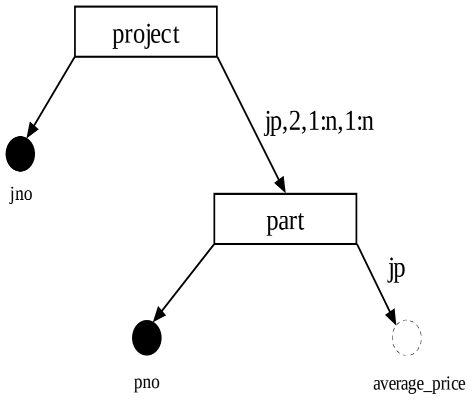

Figure 7: ORA-SS schema of the view *project-part*

```
Create View As project-part
For    $j In document("spj.xml")//project
Return
     <project jno={$j/@jno}>
       { For $pn In distinct($j//part/@pno)
          Let    $p := $j//part[@pno=$pn]
          Return
                 <part pno={$pn}>
                       <average_price>
                              {avg($p/price)}
                       </average_price>
                 </part>
        }
     </project>
```

Figure 8: XQuery expression of the view *project-part*

According to the view schema, one may easily write a view definition in XQuery expression. Figure 8 gives the view definition for the *project*-*part* view. This is a nested FLWR expression: the outer block binds $j to each *project*, while the inner block binds $p to *parts* under the *project*. We adopt this form of FLWR expression because it is able to combine all *parts* in a given *project* together. A distinct function in the inner for clause eliminates duplicate *part* numbers from the set of all *part* numbers under each *project* so that $pn is bound to each distinct *part* number. Then $p is bound to all *parts* that have the same *part* number in the *project* bound by $j. $p also serves as the argument to the aggregate function avg, which is one of the functions in XQuery core function library. The function computes the average value of *price* of all *parts* bound by $p.

This example shows that flexible views can be designed based on ORA-SS with its additional semantics. However, we need to handle the semantics properly so that meaningful views are guaranteed. The following rules can be used to design valid views that apply projection operations.

- **Rule Proj1.** If an object class has been dropped, then its attributes must be dropped too.
- **Rule Proj2.** If an object class has been dropped, then all relationship types containing the object class must be dropped too. The attributes of these relationship types must be dropped, or mapped using some aggregate function, such as avg, max/min or sum, or mapped into attributes typed in bag of value if they cannot be aggregated.

Rule Proj1 indicates that we cannot leave an attribute in the view if its object class has been dropped. Without its object class, the attributes will lose their meaning. When an object class is dropped, it may be due to the object class itself is dropped, or due to the key attribute of the object class is dropped.

On the other hand, rule Proj2 indicates that those relationship types containing the dropped object class must be dropped too. Although these relationship types will not be shown in XML document or XML schema, they need to be dropped to keep the semantics in the ORA-SS view schema consistent. The attributes of these relationship types can be dropped too. However, ORA-SS allow us to map the attributes of affected relationship types into some aggregate function attributes, such as avg, max/min, or sum, which make the view more expressive and more powerful. These modified attributes certainly should be meaningful in the view. In cases where the type of the attributes is string that cannot be aggregated, these attributes can be changed into attributes typed in bag of value.

3.4.3 Join Operation

Join operations join object classes and their attributes together by key – foreign key references. There may be one referencing object class and one referenced object class in an ORA-SS source diagram. The former object class has an attribute that is actually a key attribute of the later object class. Therefore, the former is able to refer to the later by the attribute, which plays the role of a foreign key. In our notion of join operations, we first combine the object classes together before combining all attributes of the object classes so that they will belong to a single object class. This is analogous to the join operations in relational databases, which joins two flat tables by key – foreign key references.

ORA-SS makes it possible to design XML views applying join operations and guarantee they are valid. This is because ORA-SS distinguishes between object classes and attributes so that two object classes can be joined. Furthermore, ORA-SS differentiates between attributes of object classes and attributes of relationship types so that attributes of relationship types will not be treated as attributes of the joined object class improperly. These features of ORA-SS are not offered by other existing semistructured data model, which will not allow the design of such views.

Example 3

Figure 9 shows an ORA-SS source schema diagram. The object class *supplier'* under *project* refers to another object class *supplier* under *retailer* by *sno*, which is the key attribute of *supplier*. There is a relationship type between *retailer* and *supplier* called *rs*, which has an attribute *contract* under *supplier*. Suppose we want to design a view called *join-supplier*, which joins *supplier* and *supplier'* together. We move the attributes of *supplier* so that they become attributes of *supplier'* in the view. The attributes *sno* and *sname* can be moved directly since they belong to *supplier* only. However, the attribute *contract* cannot be moved in the same way because it belongs to the relationship type between *retailer* and *supplier*. Otherwise, *contract* will become an attribute of *supplier'*, which will violate the original semantics and render the view invalid. Therefore, the object class *retailer* and the attribute *contract* will remain in the source schema and will not be mapped to the view. Figure 10 shows the view schema obtained.

Figure 11 gives the view definition of the *join-supplier* in XQuery expression. It first binds $j to the object class *project* in document "spjr.xml", which represents an XML document conforming to the schema in Figure 9. The join operation in the view can be evaluated as follows: for each *supplier'* ($s) under *project*, *sno* and *sname* of the *supplier'* is extracted from a *supplier* ($ref_s) under *retailer* whose *sno* is equal to *sno* of the *supplier'*. The two values – *sno* and *sname* are used to construct the joined *supplier*'s *sno* and *sname*.

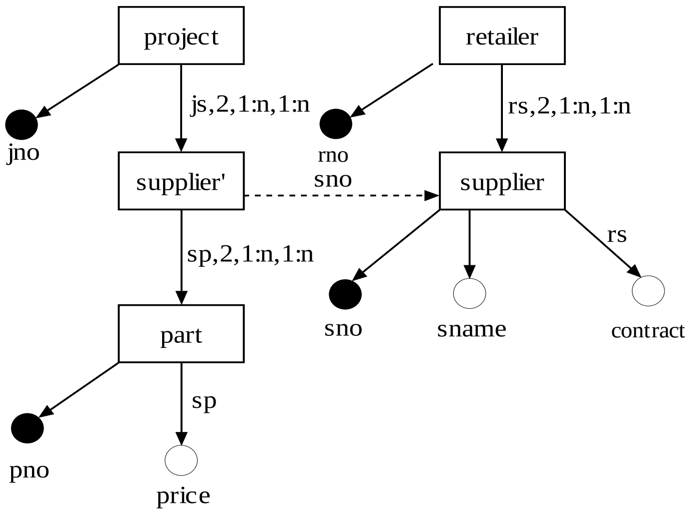

Figure 9: An ORA-SS schema diagram on project, supplier, part and retailer

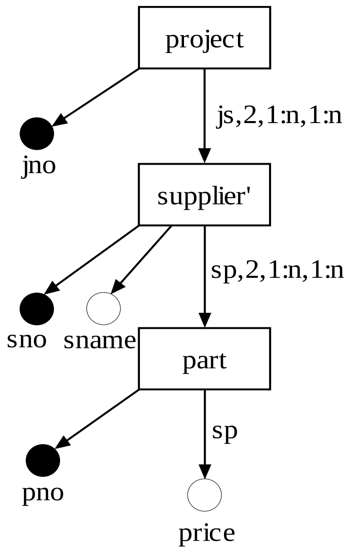

Figure 10: ORA-SS Schema of the view *join-supplier*

```
Create View As join-supplier
For $j In document("spjr.xml")//project
Return
  <project jno={$j/@jno}>
    { For $s In $j/supplier,
            $ref_s In
                      document("spjr.xml")//retailer
                      /supplier[@sno=$s/@sno]
       Return
         <supplier sno={$ref_s/@sno}
                       sname={$ref_s/@sname}>
                 {$s/part}
         </supplier>
    }
  </project>
```

Figure 11: XQuery expression of the view *join-supplier*

We will now give the rules to design valid views that applies join operations:

- **Rule Join1.** If a referencing object class and a referenced object class are joined together in the view, then the relationship types and their attributes below the referenced object class that contain those object classes above the referenced object class must be dropped.
- **Rule Join2.** If a referencing object class and a referenced object class are joined together in the view, then the relationship types and their attributes below the referenced object class that do not contain those object classes above the referenced object class can be selected or dropped according to the view requirement.

When we design views that will join one referencing object class and one referenced object class together, we need to handle the relationship types and their attributes below the referenced object class properly. These relationship types can be divided into two types. The first type of relationship types contains object classes above the referenced object class. Rule Join1 states that such relationship types and their attributes must be dropped, because those object classes above the referenced object class cannot exist in the view any more, and then these relationship types cannot exist too.

On the other hand, the second type of relationship types only contains object classes below the referenced object class. Rule Join2 states that these relationship types and their attributes can be dropped or selected according to the view requirement. This is because the object classes below the referenced object class may exist in the view, which will allow the corresponding relationship types to be included in the view too. This will not violate the semantics in the source schema.

3.4.4 Swap Operation

Swap operations restructure the source schema by exchanging the positions of a parent object class and its child object class. Swap operations are unique in XML because they can be applied only in hierarchical structure.

Example 4
Given the source schema in Figure 1, we design a view called *swap-supplier-part*, which swaps the object class *supplier* and *part* hierarchically. Figure 12 shows the view obtained. After the object classes have been swapped, we need to ensure that their attributes are relocated properly. The attributes *pno* and *sno* are also swapped in order to preserve their parent object classes. However, the attribute *price*, which belongs to the relationship *sp*, must stay with the new child object class *supplier* in order to preserve the semantics of the source schema. If it moves with the object class *part*, then it will violates the semantics in the source schema.

An XQuery expression of the *swap-supplier-part* view is described in Figure 13. Three nested FLWR expressions are used to construct the view structure. In order to swap *supplier* and *part*, the view constructs the object class *part*, before constructing *suppliers* of the *part*.

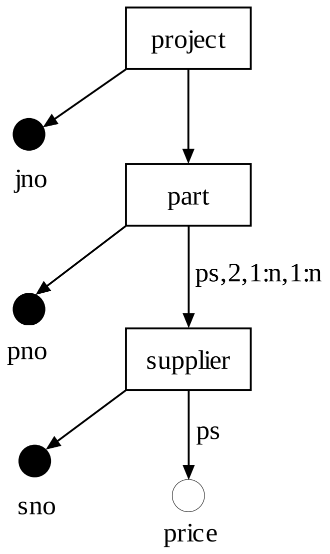

Figure 12: ORA-SS schema of the view *swap-supplier-part*

```
Create View As swap-supplier-part
For $j In document("spj.xml")//project
Return
        <project jno={$j/@jno}>
        {  For $pn In distinct($j//part/@pno)
            Return
              <part pno={$pn}>
              {  For $s In $j/supplier[part/@pno=$pn]
                  Return
                    <supplier sno={$s/@sno}>
                        {$s/part[@pno=$pn]/price}
                    </supplier>
              }
              </part>
        }
        </project>
```

Figure 13: XQuery expression of the view *swap-supplier-part*

The following rules guarantee that the design of views is valid when swap operations are applied.

- **Rule Swap1.** If two object classes are swapped in the view, then the attributes of each of the object classes must stay with the object class.

- **Rule Swap2.** If two object classes are swapped in the view, then the attributes of relationship types containing the two object classes must stay below the lowest participating object class in the relationship types.

When a swap operation is applied, the two swapped object classes may not involve any relationship type. In this case, we simply swap them and move their attributes with them, as stated in Rule Swap1. If there is any relationship type containing the two object classes, then we must keep the attributes of the relationship types below the lowest participating object class of the relationship types. If these attributes move with one of the object class, they will not belong to the relationship types and become attributes of the object class, which then violates the semantics in the source schema and lead to a meaningless view.

3.5 Design Rules for IDentifier Dependency Relationship

The previous sections present the design rules when projection, join and swap operations are applied in XML views. However, these rules are not enough when the views contain IDD (IDentifier Dependency) relationship types. An IDD relationship type is defined as follows:

Definition 1. An object class A is said to be ID dependent on its parent object class B if A does not have a key attribute, and an A object can be identified by its parent's key value (say k1) together with some of its own attributes (say k2). That is, the key of A is {k1, k2}. The relationship type between A and B is then called IDD relationship type.

Example 5
Figure 14 shows an IDD relationship type between the object class *employee* and *child*. The object class *child* does not have a key attribute, but can be identified by the key attribute of *employee* – *eno* and its own attribute – *cname*. When we design a view over the IDD relationship type, additional rules are needed to keep the view meaningful.

Based on Figure 14, we design a view applying a swap operation, which swaps the object class *employee* and *child* (see Figure 15). Unlike the previous view applying swap operations, this view still duplicates the key attribute of *employee* – *eno* for the object class *child* so that *eno* and *cname* can combine a key for the object class *child*. It is because the object class *child* cannot be identifiable without *eno*. Note this view need to be enforced with a constraint, which says the *eno* under the object class *child* must be the same as the *eno* under the object class *employee*. The straight line between the incoming edges of the attributes *eno* and *cname* denotes {*eno*, *cname*} is a composite key for the object class *child*.

We can also design a view applying projection operation. For example, Figure 16 depicts a view that drops the object class *employee*. To make the object class *child* identifiable, the key attribute of *employee* – *eno* is also combined with the attribute *cname* to construct a key for the object class *child*.

The similar situation exists if a join operation is applied in a source schema containing an IDD relationship type.

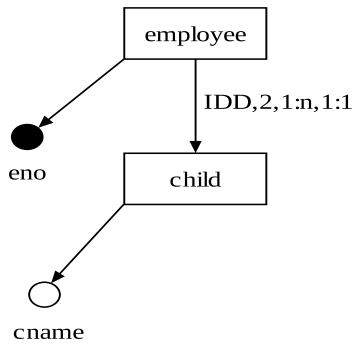

Figure 14: ORA-SS source schema diagram of an IDD relationship type

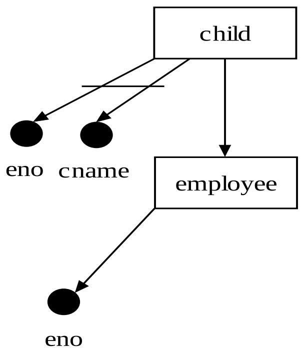

Figure 15: ORA-SS schema of the view swapping employee and child

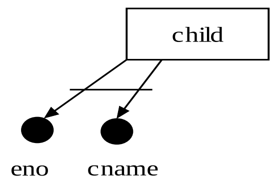

Figure 16: ORA-SS schema of the view dropping employee

These examples show that when we design a view that destroys an IDD relationship type, the key attribute of the parent object class of the IDD relationship type should be added to the child object class to construct a key for the child. The following additional rules indicate for each operation, how XML views should be designed when IDD relationship types are involved.

- **Rule Proj_IDD.** If an object class is a parent object class of an IDD relationship type and is dropped in the view, then its key attribute must be added to the child object class of the IDD relationship type to construct a key for the child.
- **Rule Join_IDD.** If an object class is a child of an IDD relationship type and is referenced by another object class in the source schema, and a view is designed to join the two object classes together, then the key attribute of the parent object class of the IDD relationship type must be added to the child to construct a key for the child.
- **Rule Swap_IDD.** If two object classes compose an IDD relationship type and are swapped in the view, then the key attribute of the parent object class must be added to the child object class to construct a key for the child.

3.6 View Validation Algorithm

In this section, we summarize all the design rules into an algorithm to validate XML views. Our algorithm will automatically modify related part of the view schema according to different operations to ensure that the view is valid.

Algorithm ValidateView
Input: ORA-SS schema diagram
Output: A valid ORA-SS view schema diagram

```
1.  Do
2.      Switch (Operation) {
3.          Case (Drop an object class):   //projection operations
4.                  if (the object class is a parent object class of an IDD relationship type){
5.                          add the key attribute of the parent to the child object class of the IDD
                                relationship type to construct a key for the child;
6.                  }   //Rule Proj_IDD
7.              drop attributes of the object class;   //Rule Proj1
8.              drop relationship types containing the object class;   //Rule Proj2
9.              handle the attributes of relationship types (drop or modify according to
                     the view requirement);   //Rule Proj2
10.             break;
11.
12.         Case (Join two object classes):   //join operations
13.              if (the referenced object class is a child object class of an IDD relationship
                 type){
14.                   add the key attribute of the parent object class of the IDD relationship
                           type to the child to construct a key for the child;
15.              }   //Rule Join_IDD
16.             drop the relationship types and their attributes that are below the referenced
                   object class but contain object classes above the referenced object class;
                   //Rule Join1
17.              handle the relationship types and their attributes that are below the
                   referenced object class and do not contain object classes above the
                   referenced object class (drop or keep according to the view requirement);
                   //Rule Join2
18.         break;
19.
20.         Case (Swap two object classes):   //swap operations
21.               If (the two object classes compose an IDD relationship type){
22.                     add the key attribute of the parent object class to the child object class
                           to construct a key for the child;
23.               }   //Rule Swap_IDD
24.               move the attributes of each object class with them;   //Rule Swap1
25.               keep attributes of relationship types containing the two object classes
                    below the lowest participating object class in the relationship types;
                    //Rule Swap2
26.               break;
27.
28.    }
29. while (view design is not done);
```

The algorithm first uses a do-while clause to monitor the process of designing view until the view is done. Then it uses a selection statement – switch clause to handle the three operations – projection, join and swap, which may be repeated in the view. Once an operation is applied in the view, the algorithm first checks if an IDD relationship type is involved. If so, then it applies the corresponding additional rule for the operation. After that, the algorithm applies the normal rules for the operation. In this way, the view will be guaranteed to be valid once it is done.

4. Related Work

Several prototype systems have been developed to support the design of XML views. The Active Views system [AAC+99] is built on top of Ardent Software's XML repository [ARD], which is based on the object-oriented O2 system. In the Active Views system, a view is presented as an object, which allows not only data, but also methods. The Active Views system utilizes XML document as its data model, which can only support views that apply selection operations. Projection, join and swap operations may be applied, but there is no guarantee that valid views are created.

MIX (Mediation of Information using XML) [BGL+99] is another system that offers a virtual XML view from its underlying heterogeneous sources. MIX utilizes XML DTD as its data model. Similar to XML document, XML DTD cannot express necessary semantics for valid XML views design, and their views can only support selection operations too. Table 1 compares our approach with the Active View system and MIX.

	Active Views system [AAC+99]	**MIX system [BGL+99]**	**Our approach**
Data model	XML	XML DTD	ORA-SS
View definition language	OQL-style language	XMAS language	XQuery language
Query language	Lorel language	XMAS language	XQuery language
Support projection, join and swap operations	No	No	Yes
Support view validation	No	No	Yes
Support graphical views design	No	No	Yes

Table 1: Comparison of ActiveViews system, MIX system and our approach

Our approach adopts the semantically rich ORA-SS data model to express both the source and view schemas. This allows us to support a richer set of views compared to Active Views and MIX. The Active Views system uses the Object Query Language as a view definition language, and the Lorel language [AQMWW97] as its query language over the views. This requires the users to be familiar with two different languages. MIX develops its own XMAS language as the view definition language and query language. In contrast, our approach directly adopts the W3C standard, XQuery as the query/view language over the views. A view definition is differentiated from a query by its additional view declaration clause before FLWR expression. Finally, both the Active Views system and MIX system do not provide for the validation of views. As a consequence, these two systems cannot support valid XML views that apply projection, join and swap operations.

5. Conclusion

In this paper, we have proposed a systematic approach for valid XML views design. The approach is composed of three steps. In first step, we transform an XML document into an ORA-SS schema diagram. In second step, we enrich the ORA-SS schema diagram with necessary semantics for valid XML views design. In third step, we develop a set of rules to guide the design of valid XML views. We also give an algorithm to validate views. We have implemented our approach into a CASE tool for designing XML views. Our future work includes providing support to update XML views.

References

[A99] S. Abiteboul. On views and XML. *In Proceedings of the Eighteenth ACM Symposium on Principles of Database Systems*, ACM Press, pages 1-9, 1999.

[AAC+99] S. Abiteboul, B. Amann, S. Cluet, A. Eyal, L. Mignet, and T. Milo. Active views for electronic commerce. *In Int. Conf. on Very Large DataBases (VLDB)*, Edinburgh, Scotland, pages 138-149,1999.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J.Widom, and J. L. Wiener. The lorel query language for semistructured data. *International Journal of Digital Libraries*, Volume 1, No. 1, pages 68-88, 1997.

[ARD] Ardent Software. http://www.ardentsoftware.com.

[BGL+99] C. Baru, A. Gupta, B. Ludaescher, R. Marciano, Y. Papakonstantinou, and P. Velikhov. XML-Based Information Mediation with MIX. *ACM-SIGMOD*, Philadelphia, PA, pages 597-599, 1999.

[DWL+00] Gillian Dobbie, Xiaoying Wu, Tok Wang Ling, Mong Li Lee. ORA-SS: An Object-Relationship-Attribute Model for Semi-Structured Data. *Technical Report TR21/00*, School of Computing, National University of Singapore, 2000.

[LLD01] Tok Wang Ling, Mong Li Lee, Gillian Dobbie. Application of ORA-SS: An Object-Relationship-Attribute Model for Semi-Structured Data. *In Proceedings of the Third International Conference on Information Integration and Web-based Applications & Services (IIWAS)*, Linz, Austria, 2001.

[XQ] http://www.w3.org/TR/xquery.

[XS] http://www.w3.org/XML/Schema.