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. Existing systems for XML views only support selection operation applied in the views and cannot validate views. In this paper, we propose a systematic approach to design valid XML views. 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. 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 operation. Finally, we develop a comprehensive algorithm that checks for the validity of XML views constructed by applying the four operations.

1 Introduction

It is necessary to provide for XML views [1]. Several systems have been proposed to support XML views, including Active Views [2] and MIX [5]. 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. First, we transform XML documents into ORA-SS schema diagram proposed in [6], [7]. Second, we enrich ORA-SS schema diagram with semantics, such as distinguishing between attributes of object classes and relationship types. 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 also 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. Section 3 describes our proposed approach to validate XML views in detail. Section 4 discusses related work and we conclude in section 5. Note there is an appendix that contains XML instance documents and XQuery used in the paper.

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 an entity type in an ER 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. 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 [6]. In this paper, we will focus on the schema diagram because it is sufficient for our purposes.

For example, the left part of figure 1 shows an ORA-SS schema diagram, which contains three object classes – *project*, *supplier* and *part*, and the right part of figure 1 then shows an ORA-SS instance diagram of the schema diagram. 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.

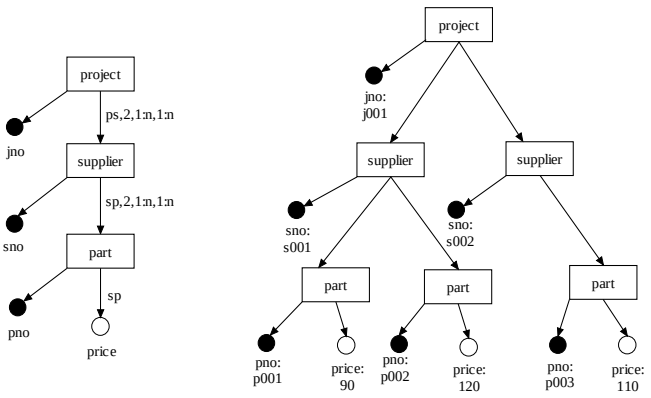

Fig.1. An ORA-SS Schema Diagram (left) and Instance Diagram (right)

In the ORA-SS schema diagram, labeled circles denote attributes, and 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 [3], XML DTD and XML Schema [9]. For this reason, we adopt ORA-SS as the data model for valid XML views design, because the additional semantics is essential for the verification of the validity of XML views. When an XML view does not violate the integrity constraint and semantics of the original XML document, we say the XML view is *valid*.

2.2 View Definition Language

The World Wide Web Consortium has proposed an XML query language called XQuery [8]. 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.

A full description of FLWR expression in XQuery can be found in [8].

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. The first two steps are preparatory stages for valid XML views design. The goal of the two steps is transform XML documents into ORA-SS schema diagram enriched with semantics, based on which, we may begin to design XML views. Therefore, we will cover them roughly in the paper.

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

We will present the rough idea of the first two steps and the detailed idea of the third step.

3.1 Motivating Example

Invalid views may be produced in the case where important semantics are not expressed in the underlying data model. We will illustrate this point with an XML document shown in XDoc 1 in Appendix. The XML document is conforming to the ORA-SS 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*. XDoc.2 in Appendix 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. XDoc.3 in Appendix describes an instance of a valid view.

The above example shows that invalid views may be 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 begin to introduce the two pre-processing steps for valid XML views design. First, we 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 the root object class in the ORA-SS schema diagram.
- Map each element that has attributes or sub-elements into an object class in the ORA-SS schema diagram.
- Map attributes of an element into the attributes of the object class corresponding to the element.
- Map the rest of the elements, which do not have attributes or sub-elements, into attributes of their corresponding parent object classes.

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. Users will be allowed to input this semantics in this step.

- Identify key attributes of each object class.
- Identify attributes that belong to object classes.
- Identify relationship types among object classes.
- Identify attributes of relationship types.

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 are designed by applying four transformation operations, which are selection, projection, join and swap. The first three are analogous to the selection, projection and join in relational databases. The fourth one is unique in XML settings 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.

Example 1

Suppose we want to design a view called *expensive-part* on the ORA-SS source schema diagram defined in Figure 1. The view definition is shown in XQuery.1 in Appendix. The 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.

Selection operations put predicates on the source schema to filter data. They do not restructure the source document. The resulting view schema will be the same as the source schema. Hence, such views will not violate semantics in the source schema. Then we do not need to set up rules to guarantee the validity of views when only selection operations are applied.

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* based on the ORA-SS schema diagram in Figure 1. This view removes the intermediate object class *supplier* (see Figure 2). 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.

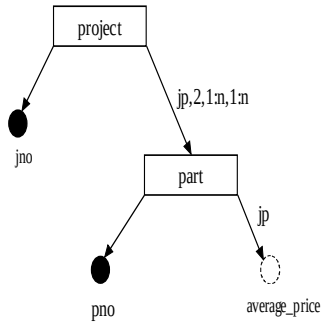

Fig.2. ORA-SS schema of the view *Project-Part*

Based on the view schema, we may write a view definition in XQuery expression, which is shown in XQuery.2 in Appendix.

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 are critical for designing valid XML 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 into attributes with some aggregate function, such as avg, max/min or sum, or mapped into attributes typed in bag of values 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 because the object class itself is dropped, or because 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 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 actually 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 two object classes together before combining all attributes of the two object classes so that they will become 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 such XML views as 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. Next we will illustrate join operation with the following example.

Example 3

Figure 3 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*.

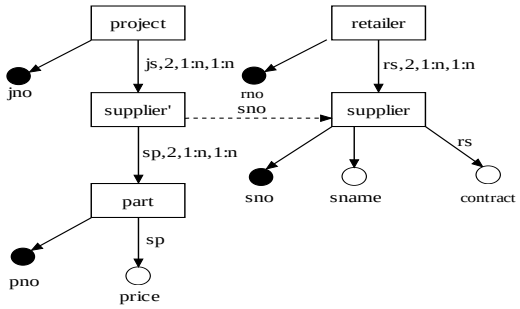

Fig.3. An ORA-SS schema diagram on project, supplier, part and retailer

We design a view called *join-supplier* shown in figure 4. The view joins *supplier* and *supplier'* together. The attributes *sno* and *sname* of *supplier* are moved under *supplier'* in the view. However, the attribute *contract* cannot be moved in the same way because it belongs to the relationship type *rs.* If it is moved under *supplier'*, then it will become an attribute of *supplier'*. The operation then violates the original semantics and makes the view invalid. Therefore, to keep the view valid, the attribute *contract* must remain in the source schema and not be moved in the view. XQuery.3 in Appendix gives the view definition of the *join-supplier.*

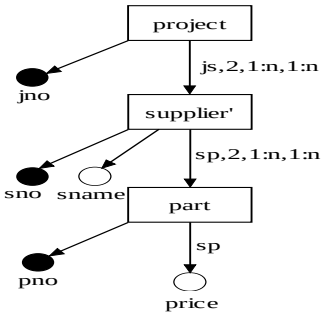

Fig.4. ORA-SS schema of the view *join-supplier*

Based on the example above, we give the rules for designing valid views that apply join operations.

- **Rule Join1.** If a referencing object class and a referenced object class are joined together in the view, and there exists such relationship types below the referenced object class as contain those object classes above the referenced object class, then attributes of such relationship types must be dropped, or mapped into attributes with some aggregate function.
- **Rule Join2.** If a referencing object class and a referenced object class are joined together in the view, and there only exists such relationship types below the referenced object class as do not contain those object classes above the referenced object class, then attributes of such relationship types can be selected or dropped according to the view requirement.

When we design views that join one referencing object class and one referenced object class together, we need to properly handle the relationship types and their attributes below the referenced object class. These relationship types can be divided into two types.

The first type of relationship types involves object classes above the referenced object class. Rule Join1 states that such relationship types and their attributes must be dropped or mapped into attributes with some aggregate function, which is the same as Rule Proj1. It is because those object classes above the referenced object class, which are ancestors of the referenced object class, will not exist in the view any more. Therefore, these relationship types involving these object classes will not exist too. Their attributes must be dropped or modified.

The second type of relationship types only involves 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. It is because the object classes below the referenced object class may still exist in the view. Then the corresponding relationship types may be included in the view also.

3.4.4 Swap Operation

Swap operations restructure the source schema by exchanging the positions of a parent object class and one of its child object class. We think this type of operation will be widely applied in XML views design because of the hierarchical nature of XML

data. Therefore we include it as one of four types of operations. The following example illustrates how to design valid XML views when swap operation is applied.

Example 4

Given the source schema in Figure 1, we design a view shown in Figure 5 called *swap-supplier-part*, which swaps the object class *supplier* and *part* hierarchically.

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 violate the semantics in the source schema.

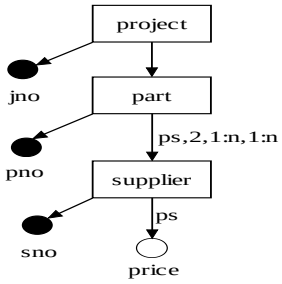

Fig.5. ORA-SS schema of the view *swap-supplier-part*

An XQuery expression of the *swap-supplier-part* view is described in XQuery.4 in Appendix. 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 involving 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 involving 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.

3.4.5 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 6 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 6, we design a view applying a swap operation, which swaps the object class *employee* and *child* (see Figure 7). 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 8 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.

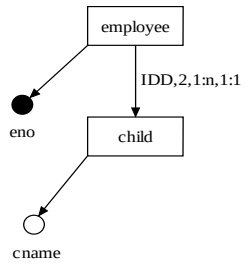

Fig.6. ORA-SS source schema diagram of an IDD relationship type

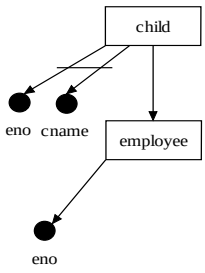

Fig.7. ORA-SS schema of the view swapping employee and child

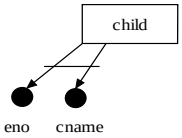

Fig.8. ORA-SS schema of the view dropping employee

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.

In summary, a theorem may be derived based on the above sections.

Theorem 1. XML views designed based on all the above rules do not violate the integrity constraints and semantics of the original XML documents.

3.4.6 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 so that the view is guaranteed to be valid.

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

```
Do
 Switch (Operation) {
  Case (Drop an object class):  //Projection operation
        if (the object class is a parent object class of an IDD relationship type){
            add the key attribute of the parent to the child object class of the IDD
                relationship type to construct a key for the child;
        }  //Rule Proj_IDD
        drop attributes of the object class;  //Rule Proj1
        drop relationship types containing the object class;  //Rule Proj2
        handle the attributes of relationship types (drop or modify according to
                the view requirement);  //Rule Proj2
        break;

    Case (Join two object classes):  //Join operations
        if (the referenced object class is a child object class of an IDD relationship
            type){
            add the key attribute of the parent object class of the IDD relationship
                type to the child to construct a key for the child;
        }  //Rule Join_IDD
        drop the relationship types that are below the referenced object class but
            contain object classes above the referenced object class; //Rule Join1
        handle the attributes of such relationship types (drop or modify according
            to the view requirement;  //Rule Join1
        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 require-
            ment);  //Rule Join2
    break;
```

```
    Case (Swap two object classes):   //Swap operations
        If (the two object classes compose an IDD relationship type){
            add the key attribute of the parent object class to the child object class
                to construct a key for the child;
        }   //Rule Swap_IDD
        move the attributes of each object class with them;   //Rule Swap1
        keep attributes of relationship types containing the two object classes
            below the lowest participating object class in the relationship types;
            //Rule Swap2
        break;

  }
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

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

	Active Views system [2]	MIX system [5]	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

Several prototype systems have been developed to support the design of XML views. The Active Views system [2] is built on top of Ardent Software's XML reposi-

tory [4], 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. MIX (Mediation of Information using XML) [5] is another system that offers a virtual XML view from its underlying heterogeneous sources. Table 1 compares our approach with the Active View system and MIX.

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 [3] 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 Conclusions

In this paper, we have proposed a systematic approach for valid XML views design. The approach is composed of three steps. The first two steps are preparatory stages. 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. In the future work, we will give more formal grounding, such as query algebra underlying view definition. We will also design query translation algorithm and provide support to update XML views in the future.

References

1. S. Abiteboul. On views and XML. In Proceedings of the Eighteenth ACM Symposium on Principles of Database Systems, ACM Press, pages 1-9, 1999.
2. 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.
3. 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.
4. Ardent Software. http://www.ardentsoftware.com.
5. 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.

6. 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.
7. 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.
8. http://www.w3.org/TR/xquery.
9. http://www.w3.org/XML/Schema.

Appendix: XML Document and XQuery in the paper

```
<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>
```

XDoc.1. 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>
```

XDoc.2. Invalid view instance of the XML document in XDoc.1

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

XDoc.3. Valid view instance of the XML document in XDoc.2

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

XQuery.1. XQuery expression of the view *expensive-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>
```

XQuery.2. XQuery expression of the view *project-part*

```
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>
```

XQuery.3. XQuery expression of the view *join-supplier*

```
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>
```

XQuery.4. XQuery expression of the view *swap-supplier-part*