Automatic Generation of XQuery View Definitions from ORA-SS 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. Many Internet-based applications have adopted XML as the standard data exchange format. These XML data are typically stored in its native form, thus creating the need to present XML views over the underlying data files, and to allow users to query these views. Using a conceptual model for the design and querying of XML views provides a fast and user-friendly approach to retrieve XML data. The Object-Relationship-Attribute model for SemiStructured data (ORA-SS) is a semantically rich model that facilitates the design of valid XML views. It preserves semantic information in the source data. In this paper, we develop a method that automatically generates view definitions in XQuery from views that have been designed using the ORA-SS model. This technique can be used to materialize the views and map queries issued on XML views into the equivalent queries in XQuery syntax on the source XML data. This removes the need for users to manually write XQuery expressions. An analysis of the correctness of the proposed algorithm is also given.

1 Introduction

XML is rapidly emerging as the standard for publishing and exchanging data for Internet-based business applications. The ability to create views over XML source data, not only secures the source data, but also provides an application-specific view of the source data [1]. Major commercial database systems provide the ability to export relational data to materialized XML views [18] [19] [20]. Among them, Microsoft's SQL server is the only one that supports querying XML views by using XPath. SilkRoute [13] [14] adopts two declarative language RXL and XML-QL to define and query views over relational data respectively. XPERANTO [5] [6] [12] uses a canonical mapping to create a default XML view from relational data, and other views can be defined on top of the default view. In addition, XQuery [17] is adopted to issue query on views of relational data in [15]. Xyleme [10] defines an XML view by connecting one abstract DTD to a large collection of concrete DTDs with an extension of OQL as the query language. ActiveView [2] [3] defines views with active features, such as method calls and triggers, on ArdentSoftware's XML repository using a view specification language. In addition, XML views are also supported as a middleware in integration systems, such as MIX [4], YAT [9] and Agora [21].

All these systems exploit the potential of XML by exporting their data into XML views. However, they have the following drawbacks. First, semantic information is ignored when presenting XML views in these systems. It is useful to preserve semantic information since it provides for checking of the validity of views [8] and query optimization. Second, users have to write complex queries to define XML views by using their own language in these systems. Although XPERANTO adopts XQuery [17], which is a standard query language for XML, it is not a user-friendly language as the XQuery expression can be long and complex.

In contrast, we propose a novel approach to design and query XML views based on a conceptual model. We adopt the Object-Relationship-Attribute model for SemiStructured data (ORA-SS) [11] as our data model because it can express more semantics compared to the DTD, XML schema or OEM. In our approach, XML files are first transformed into the ORA-SS source schema with enriched semantics. Valid ORA-SS views can be defined over the ORA-SS source schema via a set of operators such as select, drop, join and swap operators [8]. A graphical tool that allows users to design XML views graphically using these query operators has been developed in [7]. The validity of these views can be checked based on the semantics in the underlying source data.

In this paper, we examine how view definitions in XQuery can be generated automatically from the valid views that have been defined using the operators above based on the ORA-SS model. Thus, users do not have to manually write XQuery expression for views, which can be complex compared to simply manipulating a set of query operators. The generated view definitions can be directly executed against the source XML files to materialize the XML views. Further, users can use the same set of query operators to issue queries on ORA-SS views, which are subsequently mapped into equivalent XQuery queries on XML source data. Here, we develop a method to automatically generate view definitions in XQuery. This method can be used to materialize XML view documents, and map queries issued on ORA-SS views into equivalent queries in XQuery on the source XML data. The correctness of the proposed method is also provided.

The rest of the paper is organized as follows. Section 2 briefly reviews the ORA-SS data model and the importance of its expressiveness. Section 3 introduces a motivating example to illustrate why the automatic generation of XQuery view definitions is desirable. Section 4 presents the details of the algorithm to generate correct view definitions in XQuery from valid ORA-SS views. Section 5 describes how XML views can be queried using our approach, and we conclude in Section 6.

2 ORA-SS Data Model

The Object-Relationship-Attribute model for Semi-Structured data (ORA-SS) [12] comprises of three basic concepts: *object classes, relationship types* and *attributes*. An object class is similar to an entity type in an Entity-Relationship diagram or an element in XML documents. A relationship type describes a relationship among object classes. Attributes are properties that belong to an object class or a relationship type.

Fig. 1 depicts two ORA-SS schema diagrams *s1* and *s2* that have been transformed from two XML files. Each schema diagram contains several object classes, denoted by a labeled rectangle in the ORA-SS schema. Attributes of an object class are shown as circles in the schema. The key attribute of an object class is denoted by a filled circle. We observe that the value of the attribute *price* is determined by both *supplier* and *part.* It is an attribute of relationship type (*sp, 2, 1:n, 1:n*), where *sp* denotes the name of the relationship type, 2 indicates the degree of the relationship type, the first (*1:n*) is the participation constraint of the parent object class (*supplier* in this case), and the second (*1:n*) is the participation constraint of the child object class (*part* in this case) in the relationship type. Fig. 1 also shows a key-foreign key reference from *project* to *project'*, implying that each key value of *project* must appear as a key value of *project'*. In general, we assume that key-foreign key references do not exist within one ORA-SS schema. This assumption is reasonable because in most cases an object class usually refers to another object class in another schema.

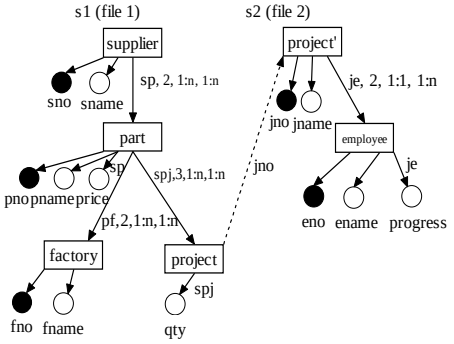


Fig. 1. An ORA-SS source schema transformed from XML files.

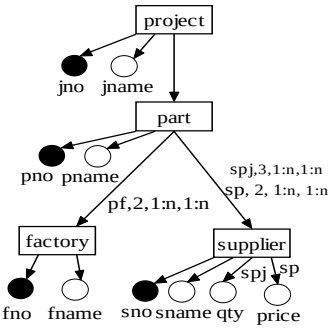


Fig. 2. An ORA-SS view of Fig. 1.

We observe that ORA-SS not only reflects the nested structure of semistructured data, but it also distinguishes between object classes, relationship types and attributes. Such semantics are lacking in existing semistructured data models including OEM, XML DTD and XML Schema [16]. In designing XML views, these semantics are critical in ensuring that the designed views are valid [8], that is, the views are consistent with the source schema in terms of semantics. Note we use four operators to design valid XML views: the select operator imposes some where conditions on attributes; the drop operator removes object classes or attributes; the join operator combines two object classes together; and the swap operator interchanges two object classes in a path. The following example illustrates how we use these operators to design valid views.

Example. Fig. 2 shows an ORA-SS view obtained by applying several query operators to the schemas in Fig. 1. First, we apply a join operator on *project* and *project'*: $join_{s1//project \rightarrow s2//project'}$, where *s1* and *s2* are the two source schemas. The attributes of *project'*, such as *jno* and *jname* can be grafted below *project* as its attributes. In addition, we can also graft the object class *employee* below *project* and maintain the rela-

tionship type *je* and its attribute, since we do not violate any semantics in the source schema. Following the join operator, we apply a swap operator to swap *supplier* and *project* and drop *employee*: $swap_{s/supplier \leftrightarrow s/supplier/part/project}\ drop_{s//employee}$, where *s* indicates the current view schema. Since *price* is an attribute of the relationship type *sp*, *price* cannot be placed below the object class *part* after the swap operator is applied. Instead, it will be placed automatically below *supplier* in our system as shown in the new view schema. Without the semantics captured in the ORA-SS schema diagram, we may design an invalid view in which the attribute *price* still remains with the object class *part*, thus resulting in inconsistency with the semantics in the original schema.

3 Motivating Example

After designing the views using the four operators, the next step is to generate view definitions in XQuery that can be executed to materialize these views. A naive solution is to write the view definitions manually according to the ORA-SS views. However, view definitions in XQuery can be very complex, as we will illustrate.

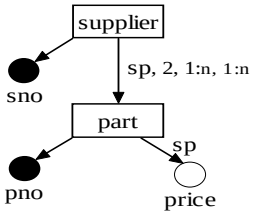


Fig. 3. An ORA-SS source schema

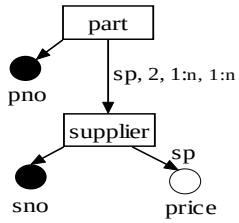


Fig. 4. An ORA-SS view schema

```
1. let $pno_set := distinct-values
       ($in//part/@pno)
2. let $sno_set := distinct-values
       ($in//supplier/@sno)
3. return <db>
4. for $p_no in $pno_set
5. let $p := $in//part[@pno=$p_no]
6. return <part pno="{$p_no}">
7.    {for $s_no in $sno_set
8.     where some $p1 in $in//part
9.     satisfies (exists($p1[@pno=$p_no])
           and  (exists($p1
           [ancestor::supplier/@sno=$s_no]))
10.    let $s :=$in//supplier[@sno=$s_no]
11.    return <supplier sno="{$s_no}">
12.              {$s/part[@pno=$p_no]/price}
13.           </supplier>
14.    }
15.  </part>
16. </db>
```

Fig. 5. View definition in XQuery expression

Consider the ORA-SS source schema in Fig. 3. Fig. 4 shows a view that has been designed using a swap operator on the object classes *supplier* and *part*, that is, $swap_{s/supplier \leftrightarrow s/supplier/part.}$ Note that the attribute *price* does not move up with *part*, because it is an attribute of the relationship type *sp*. The swap operator is able to handle this

automatically. Fig. 5 shows the XQuery expression for the view in Fig. 4. The variable *$in* represents the XML file corresponding to the source schema in Fig. 3.

It is clear that the XQuery expression is much more complex than the swap query operator that generates the view. In general, the complexity and length of XQuery view definitions increases dramatically as the number of object classes increases. The likelihood of introducing errors in the view definitions also increases if users are to manually define such views in XQuery. This problem can be addressed using our approach which provides a set of query operators for users to define views from which XQuery expressions can be automatically generated.

4 Generation of XQuery View Definitions

The main idea in the proposed algorithm is to generate the definition of each object class individually and then combine all the definitions together according to the tree structure of the view. The definition of an object class comprises of a FLWR expression in XQuery. The FLWR expression consists of *for, let, where* and *return* clauses. Basically, the algorithm first generates a where clause to restrict the data instances represented by the object class (say *o*). Then it generates a *for* clause to bind a variable to iterate over each distinct key value of *o* that are qualified by the *where* clause. Finally, it generates a *return* clause to construct the instances of *o*.

While it is relatively straightforward to generate the *for* and *return* clauses for each object class in a view, it is not a trivial task to generate the condition expressions in the *where* clause, which restrict the instances of the object class in the view. This is because there may exist relationship sets among the object classes. Thus, many different object classes may exert influences on a given object class in the view. In order to generate the correct condition expressions for an object class in a view, we use the following intuition, that is, the data instances for an object class in a view are determined by all the object classes in the path from the root to the object class.

Definition. Given an object class *o* in an ORA-SS view, the path from the root of the view to *o* is called the *vpath* of *o*. Object classes that occur in *vpath* of *o*, except for the root and *o* itself, determine all condition expressions for object class *o* in the view.

By analyzing the object classes in the vpath of an object class *o*, we can capture all their influences on *o* in a series of where conditions. In the next subsection, we first determine the possible types of object class that can appear in a *vpath*. Then we provide a set of generic rules to guide the generation of where conditions for each type of object class.

4.1 Analyzing Vpath

There are three types of object classes in the *vpath* of an object class *o* in any views designed by the 4 operators mentioned before. The object classes in a *vpath* are classified based on their origin in the source schema.

Type I: For any object class *o* in a view schema, a Type I object class in its *vpath* originates from some *o*'s ancestor or descendant in the source schema.

Type II: For any object class *o* in a view schema, a Type II object class in its *vpath* originates from some descendant of some *o*'s ancestor in the source schema. In other words, Type II object classes in *o*'s *vpath* are *o*'s siblings, descendants of *o*'s siblings, *o*'s ancestors' siblings, or descendants of *o*'s ancestors' siblings in the source schema.

Type III: For any object class *o* in a view schema, a Type III object class in its *vpath* originates from the object classes in another source schema, whose ancestor or descendant has a key-foreign key reference with *o*'s ancestor or descendant in *o*'s source schema. They are generated in the *vpath* by a single join operator only, or a single join operator and a series of swap operators together.

The three object types introduced above include all the object classes in the vpath of a given object class.

Example. Fig. 6 illustrates the three different object types. We design a valid view in Fig. 6(b) based on the source schema in Fig. 6(a) using our operators [8]. Consider the *vpath* of object class O in the view. The object classes B and P are the ancestor and descendant of O respectively in the source schema (see Fig. 6(a)). Therefore, B and P are Type I object classes in the *vpath* of O. On the other hand, the object class J is O's ancestor B's descendant in the source schema. Therefore, J is a Type II object class in the *vpath* of O. Finally, the object class K is from the source schema 1 in Fig. 6(a), whose parent F has a key-foreign key reference with D, which is the parent of O. Therefore, K is a Type III object class in O's *vpath* that is obtained by first applying a join operator to D and F so that K can be grafted below D as its child, and then applying swap operators so that K can become a parent of O.

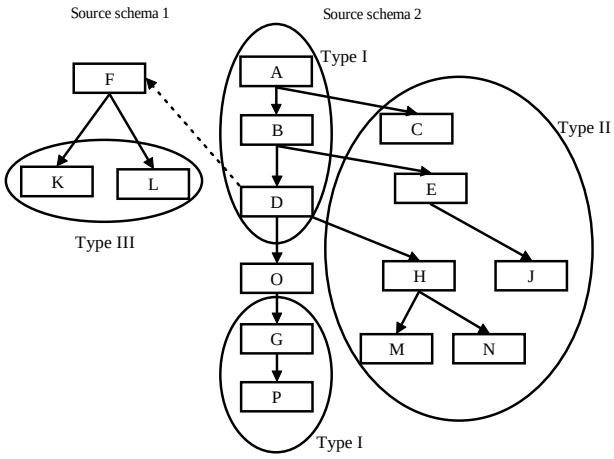


Fig. 6 (a). Two simplified ORA-SS source schema

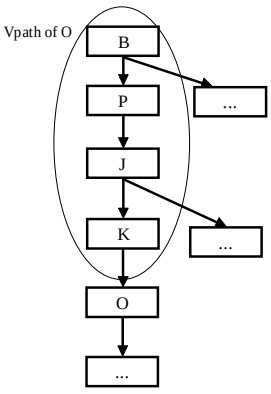
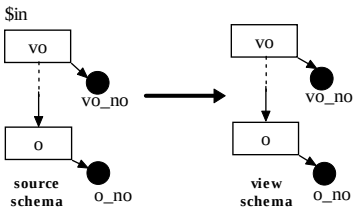


Fig. 6 (b). A simplified ORA-SS view schema

4.2 Generating Where Conditions

Next, we present a set of rules to guide the generation of *where* conditions for each type of object class. The generated *where* conditions, in bold, reflect the influence the object classes exerts on *o*. We will use the notation *vo* to an arbitrary object class in *o*'s *vpath* in the view. Note that *vo* is not the root of the views in the following figures. To simplify discussion, we just present a path of the views that contains *vo* and *o*. The key attributes of the two object classes will be referred to as *o_no* and *vo_no* respectively in the following rules. Since *vo* is an ancestor of *o* in the view, and a depth first search is employed to generate the query expression for each object class in the view, the query expression for *vo* is generated before *o*. The variable *$vo_no* denotes the current qualified key value of *vo*, and *$in* represents the XML source file.

Rule Type I_A. If *vo* is an ancestor of *o* in the source schema (see Fig. 7(a)), then Fig. 7(b) defines the *where* condition (in bold) to generate.

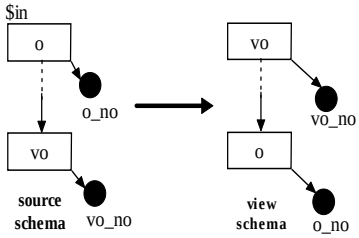



```
let $o_no_set := distinct-values($in//o/@o_no)
for $o_no in $o_no_set
where some $vo1 in $in//vo satisfies (
        exists( $vo1[@vo_no=$vo_no]) and
        exists($vo1[descendant::o/@o_no=$o_no]) )
```

Fig. 7(a). Case for Rule Type I_A **Fig. 7(b). Condition generated by Rule Type I_A**

To understand the context for the *where* condition, we have also shown the *let* and *for* clauses. Note that the let and for clauses are generated for the object class *o* only once. The entire *where* conditions for the object classes in the *vpath* of *o* are linked together using "*and*" in a single *where* clause. The *where* condition generated by Rule Type I_A indicates if an instance of *o* with key value *$o_no* is selected as a child of an instance of *vo* with the current qualified key value *$vo_no* in the view, then there must exist an instance of *vo* in the source that has a key value *$vo_no* and has a descendant instance of *o* with key value *$o_no*.

Rule Type I_B. If *vo* is a descendant of *o* in the source schema (see Fig. 8(a)), then Fig. 8(b) defines the *where* condition to generate.




```
let $o_no_set := distinct-values($in//o/@o_no)
for $o_no in $o_no_set
where some $vo1 in $in//vo satisfies (
        exists( $vo1[@vo_no=$vo_no]) and
        exists($vo1[ancestor::o/@o_no=$o_no]) )
```

Fig. 8(a). Case for Rule Type I_B **Fig. 8(b). Condition generated in Rule Type I_B**

Rule Type I_B is similar to Rule Type I_A except that the axis before *o* is an ancestor, instead of a descendant in the generated *where* condition.

In the case where *vo* is a Type II object class in *o*'s *vpath*, *vo* has no ancestor-descendant relationship with *o*. However, it still has influence on *o* through an intermediate object class – the Lowest Common Ancestor of *vo* and *o*. Consider Figure 9. If an instance of *vo*, say *vo1,* appears in the *vpath* of o in the view document, then *vo1* must be under an instance of the Lowest Common Ancestor of vo and o, say *lca1,* in the source document, which actually determines a set of instances of o, say (*o1, o2, …on*), under *lca1* in the source document. Therefore, *vo1* determines (*o1, o2, …on*) through *lca1*. The reason why we use the lowest common ancestor of the two object classes as the intermediate object class is that it correctly reflects the restriction of *vo* on *o*. Otherwise, we may introduce a wider range of instances of *o,* some of which are not determined by *vo.* We have the following two rules for Type II object classes.

Rule Type II_ A. If *vo* is a Type II object class in *o*'s *vpath* and the Lowest Common Ancestor of *vo* and *o*, say LCA, is also in the *vpath* of *o* in the view schema (see Fig. 9), then there is no need to generate a where condition for the restriction of *vo* on *o*.

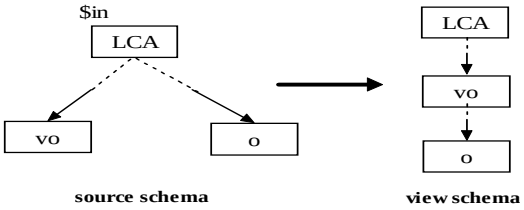


Fig. 9. Case for Rule Type II_A

Rule Type II_A states that we do not need to consider the influence of *vo* on *o* when the Lowest Common Ancestor of *vo* and *o* is also in the *vpath* of *o.* This is because this influence will be considered when the algorithm processes the Lowest Common Ancestor (LCA) as another object class in the *vpath* of *o.*

Rule Type II_B. If vo is a Type II object class in o's *vpath* and the Lowest Common Ancestor of *vo* and *o*, say LCA, is not in the *vpath* of *o* in the view schema (see Fig. 10 (a)), then Fig. 10 (b) defines the where condition generated.

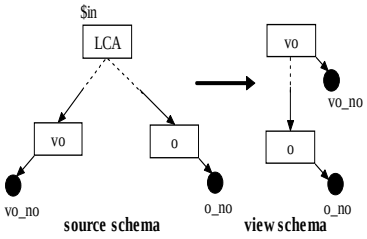


Fig. 10(a). Case for Type II_B

```
let $o_no_set := distinct-values($in//o/@o_no)
for $o_no in $o_no_set
where some $LCA in $in//LCA satisfies (
        exists($LCA//o[@o_no=$o_no]) and
        exists($LCA//vo[@vo_no=$vo_no]) )
```

Fig. 10(b). Condition generated in Rule Type II_B

Rule Type II_B presents the *where* condition in the case where the LCA does not occur in the *vpath* of *o*. The condition states that if an instance of *o* with key value *$o_no* is selected in the view under the instance of *vo* with the current qualified key value *$vo_no*, then there must exist an instance of LCA in the source that has both a descendant instance of *o* with key value *$o_no* and a descendant instance of *vo* with

key value *$vo_no*. In other words, the instances of *vo* and *o* must have a common ancestor instance of LCA.

Next, we process the case where *vo* is a Type III object class in the *vpath* of *o*. We have *vo* and *o* that are linked together by the referencing and referenced object classes of a join operator. Consider Figure 11 (a). We assume that *vo* and *o* originates from two different schemas (*$in1* and *$in2*). The influence of *vo* on *o* is as follows: an instance of *vo*, say *vo1*, has an ancestor instance of the referenced object class, say *referenced*$_1$, which in turn determines an instance of the referencing object class, say *referencing*$_1$, which refers to the *referenced*$_1$ by key-foreign key reference. As a descendant of *o*, the instance of the referencing object class must determine an instance of *o*, say *o1*. In this way, an instance of *vo* (*vo1*) determines an instances of o (*o1*) through the referencing and referenced object classes together.

Rule Type III_A. If *vo* is a descendant of the referenced object class and o is an ancestor or descendant of the referencing object class in the source schema, and the referencing object class is in *o*'s vpath in the view schema (see Fig. 11), then there is no need to generate a where condition.

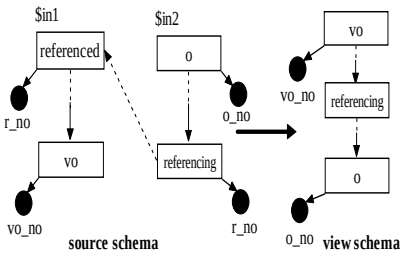


Fig. 11(a). O as an ancestor of the referencing in Rule Type III_A

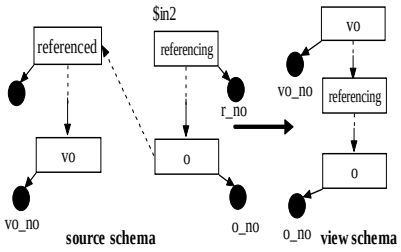


Fig. 11(b). O as a descendant of the referencing in Rule Type III_A

Since the influence of *vo* on *o* will be considered when processing the referencing object class, we do not need to generate a *where* condition for the restriction of *vo* on *o*. In this case, o can be an ancestor or descendant of the referencing object class.

Rule Type III_B. If *vo* is a descendant of the referenced object class and *o* is an ancestor of the referencing object class in the source schema, and the referencing object class is not in *o's vpath* in the view schema (see Fig. 12(a)), then Fig. 12(b) defines the *where* condition generated.

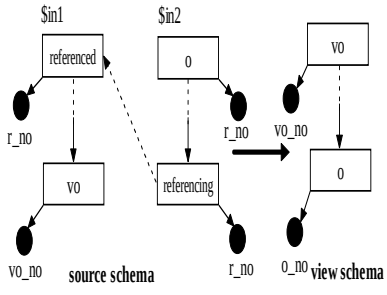


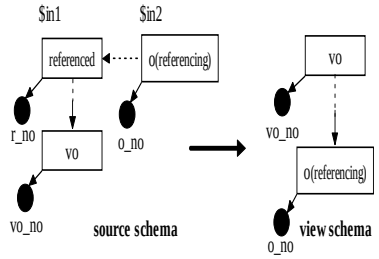
Fig. 12(a). Case for Rule Type III_B

```
let $o_no_set := distinct-values($in//o/@o_no)
for $o_no in $o_no_set
where some $referenced in $in1//referenced satisfies
(exists
  ($referenced[descendant::vo/@vo_no=$vo_no]) )
and
  some $referencing in $in2//referencing satisfies (
  exists($referencing[@r_no=$referenced/@r_no])
and
    exists($referencing[ancestor::o/@o_no=$o_no]) )
```

Fig. 12(b). Condition generated in Rule Type III_B

Rule III_B states that if an instance of *o* with key value *$o_no* is selected under the instance of *vo* with the current qualified key value *$vo_no*, then there must exist an instance of the referenced object class in source 1 (*$in1*) that has a descendant *vo* with key value *$vo_no*. Moreover, there must exist an instance of referencing object class in source 2 *($in2*) that has a key value equal to the instance of the referenced object class's key value and has an ancestor instance of *o* with key value equal to *$o_no*.

Rule Type III_C. If *vo* is a descendant of the referenced object class and o is the referencing object class itself in the source schema(see Fig. 13 (a)), then Fig. 13 (b) defines the where condition generated.

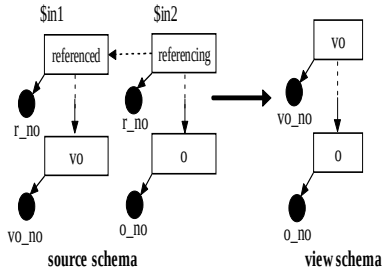



```
let $o_no_set := distinct-values($in//o/@o_no)
for $o_no in $o_no_set
where some $referenced in $in1//referenced satisfies
 ( exists($referenced[@r_no=$o_no]) and
  exists($referenced[ancestor::vo/@vo_no=$vo_no]))
```

Fig. 13(a) Case for Rule Type III_C **Fig. 13(b). Condition generated in Rule Type III_C**

Rule Type III_C states that if an instance of *o* with key value *$o_no* is selected under the instance of *vo* with the current qualified key value *$vo_no*, then there must exist an instance of the referenced object class, which has a key value equal to *$o_no* and a descendant instance of *vo* with key value *$vo_no*.

Rule Type III_D. If *vo* is a descendant of the referenced object class and *o* is a descendant of the referencing object class in the source schema, and the referencing object class is not in *o's vpath* in the view schema (see Fig. 14 (a)), then Fig. 14 (b) defines the where condition generated.




```
let $o_no_set := distinct-values($in2//o/@o_no)
for $o_no in $o_no_set
where some $referenced in $in1//referenced satisfies
(exist
($referenced[descendant::vo/@vo_no=$vo_no]) )
and
some $referencing in $in2//referencing satisfies (
  exists($referencing[@r_no=$referenced/@r_no])
and exist
($referencing[descendant::o/@o_no=$o_no]) )
```

Fig. 14(a). Case for Rule Type III_D **Fig. 14(b). Condition generated in Rule Type III_D**

Rule Type III_D is similar rule as Rule Type III_B except that *o* is now not an ancestor, but a descendant of the referencing object class.

Note the above set of rules for Type III object class handles the cases where *vo* is always a descendant of the referenced object class in the source schema. A similar set of rules can be derived for the cases where *vo* is an ancestor of the referenced object

class in the source schema. However, these sets of rules are not enough for all cases where *vo* is a Type III object class in the vpath of *o* in the view. Note these rules consider the case where vo is from the schema of the referenced object class and o is from the schema of the referencing object class. We observe that *vo* can also originate from the schema of the referencing object class and *o* from the schema of the referenced object class according to the definition of Type III in Section 4.1. In this case, similar set of rules can still be derived to generate the *where* condition of *vo* on *o*.

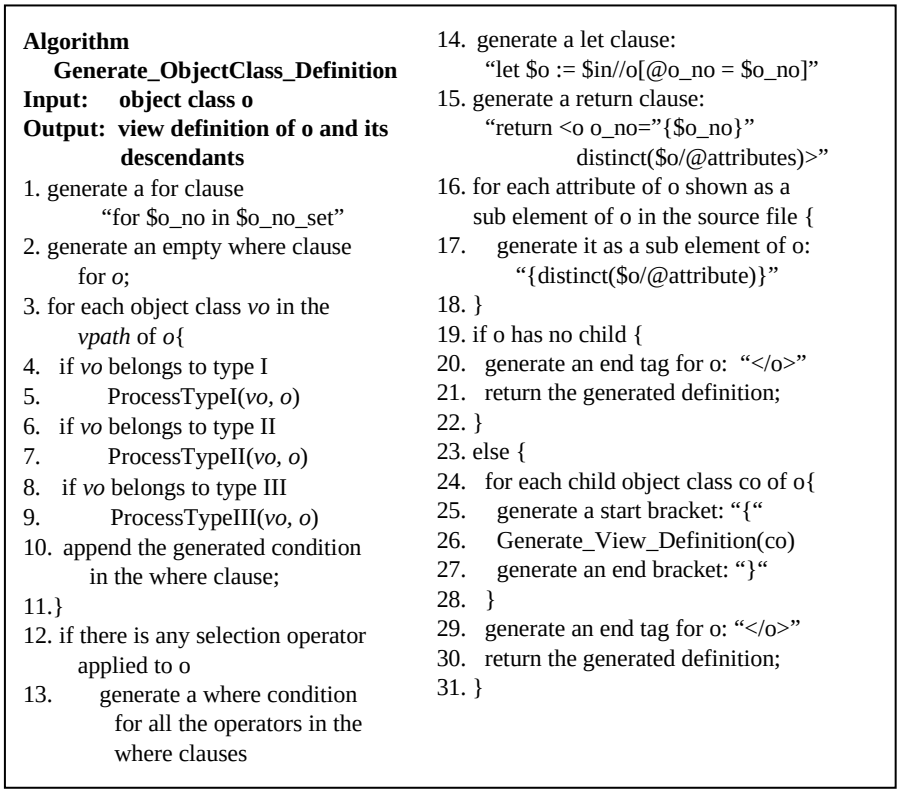
4.3 Algorithms

Fig. 15 presents the details of the algorithm Generate_View_Definition to generate view definitions in XQuery. The inputs are valid ORA-SS views. The algorithm first generates a set of let clauses for all object classes in the view using a depth first search method. Each of these clauses binds a global variable to all possible distinct key values of a different object class. Next, it generates a root element for the view because each XML document must have a root element. The root is above the first object class in the ORA-SS views. By default, this root element is not shown as an object class in the ORA-SS views. Finally, for each child object class of the root, say *o*, it calls the algorithm Generate_ObjectClass_Definition to generate a definition for *o* and all its descendants. Each of the definition is contained in a pair of curly brackets, indicating that they are sub-elements of the root element.

Algorithm **Generate_View_Definiton**
Input: **view schema v;** **source schema s**
Output: view definition of v
1. for each object class o in the view
2. generate a let clause:
   "let $ono_set := distinct-values($in//o/@o_no)"
3. generate the start tag for root of the view:
   "return <root>"
4. for each child o of the root of v {
5. generate a start bracket: "{"
6. Generate_Objectclass_Definition(o)
7. generate a end bracket: "}"
8. }
9. generate the end tag for root of the view:
   "</root>"

Fig. 15. Algorithm to generate view definition

Generate_ObjectClass_Definition (Fig. 16) returns the view definition of *o* and all its descendants in *v*. The functions ProcessTypeI, ProcessTypeII and ProcessTypeIII take *vo* and *o* as inputs and generate where conditions that reflect the restriction of *vo* on *o* based on the rules described in the previous section. Note attributes that are below *o* and shown as sub elements of *o* in the source file are generated as sub-elements of *o* (line 16-18). For each child of *o*, the same algorithm is invoked recursively until all the descendants of *o* have been processed (line 24-28).



Algorithm
Generate_ObjectClass_Definition
Input: **object class o**
Output: **view definition of o and its descendants**

1. generate a for clause
   "for $o_no in $o_no_set"
2. generate an empty where clause for *o*;
3. for each object class *vo* in the *vpath* of *o*{
4. if *vo* belongs to type I
5. ProcessTypeI(*vo*, *o*)
6. if *vo* belongs to type II
7. ProcessTypeII(*vo*, *o*)
8. if *vo* belongs to type III
9. ProcessTypeIII(*vo*, *o*)
10. append the generated condition in the where clause;
11. }
12. if there is any selection operator applied to o
13. generate a where condition for all the operators in the where clauses
14. generate a let clause:
    "let $o := $in//o[@o_no = $o_no]"
15. generate a return clause:
    "return <o o_no="{$o_no}"
    distinct($o/@attributes)>"
16. for each attribute of o shown as a sub element of o in the source file {
17. generate it as a sub element of o:
    "{distinct($o/@attribute)}"
18. }
19. if o has no child {
20. generate an end tag for o: "</o>"
21. return the generated definition;
22. }
23. else {
24. for each child object class co of o{
25. generate a start bracket: "{"
26. Generate_View_Definition(co)
27. generate an end bracket: "}"
28. }
29. generate an end tag for o: "</o>"
30. return the generated definition;
31. }

Fig. 16. Algorithm to generate object class definition

4.4 Correctness of Algorithm

The intuition behind Generate_ObjectClass_Definition is that the data instances represented by an object class in an ORA-SS view are determined by all the object classes in its *vpath*. A pre-condition for the algorithm is: *o* is an object class of an ORA-SS view and the number of the descendants of *o* in the view is *n* ($n \geq 0$). After executing the algorithm with *o* as input, we have *result* = Generate_Objectclass_Definition(*o*). Then a postcondition states what is to be true about the generated result which is given by *result* = XQuery expression of a sub tree rooted at *o*. The proof of correctness takes us from the precondition to the postcondition.

(a) $n = 0$. This is the base case where *o* has no children. For each object class in the *vpath* of o, we generate the *where* condition according the rules that correspond to the object type. A return clause is generated to construct the result of *o*. Thus, the algorithm generates and returns the correct XQuery expression for *o* itself.

(b) $n > 0$. In this inductive step, *o* will have children. We have an inductive hypothesis that assumes Generate_Objectclass_Definition(*o*,*v*) returns the correct XQuery expression of a sub tree rooted at *o* for all the object class *o* such that $0 \leq j \leq$ n-1 where *j* is the number of descendants of *o*. From the base case, the algorithm first

generates the correct XQuery expression for *o* itself. Then it processes each child of *o*, say *c*. By the inductive hypothesis, GenerateViewDefinition(*c*) will return the correct XQuery expression of a sub tree rooted at *c* since $0 \leq j \leq$ n-1 where *j* is the number of descendants of *c*. By combining the query expressions of *o* and its children, the algorithm returns the correct XQuery expression of a subtree rooted at *o*.

5 Querying ORA-SS Views

Having automatically generated the view definition in XQuery, we can now execute it against XML files using existing XQuery engines. This allows users to browse the materialized view documents. In this section, we demonstrate how this approach can be used to support queries on the ORA-SS views.

In general, users may only be interested in a particular item of the view with some selection conditions. That is, the queries on views consist of only selection operations. We can compose these queries with the generated XQuery view definition to rewrite the query on the view. Specifically, we insert the conditions into the corresponding where clauses in the view definition. Then we execute the rewritten query against source XML files to generate the query result.

In situations where users issue more complex queries involving swap or join operators, we will directly apply the query to the view and generate the ORA-SS result tree of the query, which is treated as an ORA-SS view in our system. We can then use the same proposed algorithm to generate its view definition in XQuery, which is in fact the rewritten query on source XML files. Thus, we map any query on ORA-SS views into an equivalent XQuery on source XML files.

A query that is composed of several operators typically requires a rather complex XQuery expression. Compared to approaches that directly employ XQuery or other query languages to issue queries on views, we offer a much simpler solution with our approach that exploits a conceptual model and a set of query operators.

6 Conclusion

In this paper, we have described a method to automatically generate XQuery view definitions from views that are defined using the ORA-SS conceptual model. The proposed technique can be used to materialize the views and map queries issued on XML views into the equivalent queries in XQuery syntax on the source XML data. This removes the need for users to manually write XQuery expressions. Although visual query languages proposed for XQuery language such as XML-GL [22] also aim to solve the problem, these visual query languages do not have a mechanism that guarantees that the constructed views are valid. In contrast, our approach provides such a facility based on ORA-SS data model. To the best of our knowledge, this is the first work to employ a semantic data model for the design and query of XML views. Using

a conceptual model for the design and querying of XML views provides a fast and user-friendly approach to retrieve XML data. Ongoing work aims to generate query definitions for ORA-SS views in the case where XML source data are stored into an object-relational database by employing the semantics in the source data.

References

1. S. Abiteboul. On views and XML. 18th ACM Symposium on Principles of Database Systems, pp. 1-9, 1999.
2. S. Abiteboul, S. Cluet, L. Mignet, et. al., “Active views for electronic commerce”, VLDB, pp.138-149, 1999.
3. S. Abiteboul, V, Aguilear, S, Ailleret, et. al., “XML repository and Active Views Demonstration”, VLDB Demo, pp.742-745, 1999.
4. C. Baru, A. Gupta, B. Ludaescher, et. al., “XML-Based Information Mediation with MIX”, ACM SIGMOD Demo, 1999.
5. M. Carey, J. Kiernan, J. hanmugasundaram, et. al., “XPERANTO: A Middleware for Publishing Object-Relational Data as XML Documents”, VLDB, pp. 646-648, 2000.
6. M. Carey, D. Florescu, Z. Ives, et. al., “XPERANTO: Publishing Object-Relational Data as XML”, WebDB Workshop, 2000.
7. Y.B. Chen, T.W. Ling, M.L. Lee, “A Case Tool for Designing XML Views”, DIWeb Workshop, 2002.
8. Y.B. Chen, T.W. Ling, M.L. Lee, “Designing Valid XML Views”, ER Conference, 2002
9. V. Christophides, S. Cluet, J. Simeon,“On Wrapping Query Languages and Efficient XML Integration”, SIGMOD, pp. 141-152, 2000.
10. S. Cluet, P. Veltri, D. Vodislav, “Views in a large scale xml repository”, VLDB, pp. 271-280, 2001.
11. G. Dobbie, X.Y Wu, T.W Ling, M.L Lee, “ORA-SS: An Object-Relationship-Attribute Model for SemiStructured Data”, Technical Report TR21/00, School of Computing, National University of Singapore, 2000.
12. C. Fan, J. Funderburk, H. Lam, Et. al., “XTABLES: Bridging Relational Technology and XML”, IBM Research Report, 2002.
13. M. Fernandez, W. Tan, D. Suciu, “Efficient Evaluation of XML Middleware Queries”, ACM SIGMOD, pp. 103-114, 2001.
14. M. Fernandez, W. Tan, D. Suciu, “SilkRoute: Trading Between Relations and XML”, World Wide Web Conference, 1999.
15. J. Shanmugasundaram, J. Kiernan, E. Shekita, et. al., “Querying XML Views of Relational Data”, VLDB, pp. 261-270, 2001.
16. “XML Schema”, W3C Recommendation, 2001.
17. “XQuery: A Query Language for XML”, W3C Working Draft, 2002.
18. Microsoft Corp. http://www.microsoft.com/XML.
19. Oracle Corp. http://www.oracle.com/XML.
20. IBM Corp. http://www.ibm.com/XML.
21. I. Manolescu, D. Florescu, D. Kossmann, “Answering XML Queries over Heterogeneous Data Sources”, VLDB Conf, 2001, pp.241-25
22. S. Ceri, S. Comai, E. Damiani, et. al., “XML-GL: a graphical language of querying and restructuring XML documents”, WWW Conf, pp. 151-165, 1999