A Semantic Approach to the Design of Valid and Reversible Semistructured Views

Yabing Chen, Tok Wang Ling, Mong Li Lee
School of Computing, National University of Singapore
{chenyabi, lingtw, leeml}@comp.nus.edu.sg

Masatake Nakanishi
Faculty of Business Management, Nagoya Keizai University
nakanishi-m@nagoya-ku.ac.jp

Gillian Dobbie
Department of Computer Science, Auckland University
gill@cs.auckland.ac.nz

Existing systems that support semistructured views do not maintain semantics during the process of designing the views. Thus, these systems do not guarantee the validity and reversibility of the views. In this paper, we propose an approach to address the issue of valid and reversible semistructured views. We design a set of view operators for designing semistructured views. These operators are *select, drop, join* and *swap*. For each operator, we develop a complete set of rules to maintain the semantics of the views. In particular, we maintain the evolution and integrity of relationships once an operator is applied. We also examine the reversible view problem under our operators and develop rules to guarantee that the designed views are reversible. Finally, we examine the changes in the participation constraints of relationship types during the view design process, and develop rules to ensure the correctness of the participation constraints.

Categories and Subject Descriptors: H.2.1 [**Database Management**]: Logical Design - Schema and Subschema; H.2.3 [**Database Management**]: Languages - Query Languages

General Terms: Design, Theory

Additional Key Words and Phrases: Schema Maintenance; Data Models; Semistructured Views; Semistructured Data; XML

1. INTRODUCTION

XML has emerged as the dominant standard for publishing and exchanging data for Internet-based business applications. Given that a large amount of data has been stored in traditional databases such as the relational database, semistructured XML

views are constructed over these databases for exchange over the Web. Exporting underlying source data to semistructured XML views not only secures the source data, but it also provides application-specific views of the source data.

A lot of works have been done on XML views. The majority of the works are focused on presenting semistructured views over relational databases using query languages, such as SilkRoute [Fernandez M. 1999; 2001], XPERANTO [Carey M. 2000a; 2000b] and ROLEX [Bohannon P. 2002]. Other works provide a semistructured view mechanism over native XML data, such as Xyleme [Cluet S. 2001] and ActiveView [Abiteboul S. 1999]. [Hwang D. H. 2005] examine how XML views can be defined, materialized and incrementally updated using an object-relational database. [Mandhani B. 2005] introduce the notion of view answerability and design a method for maintaining a cache of materialized XPath views. Semistructured views are also presented as a middleware in data integration system, such as MIX [Baru C. 1999] and MARS [Deutsch and Tannen 2003]. [Rajugan R. 2005] consider XML views as a way of representing and processing non-XML data as XML. Based on the virtual XML concept, one can construct a default view or a specific view for a non-XML format and issue aggregate queries on an aggregate of XML or non-XML data. [Rajugan R. 2005] propose a three layer XML view model to facilitate the design and manipulation of XML data at a higher level of abstraction. The work also incorporate conceptual query operators such as select, project and join to allow the definition of view, however, it does not consider the swap operator and the design of valid and reversible views. [Ni W. 2003] design a graphical query language called GLASS for semistructured data.

We observe that if a system does not maintain the semantics implied in the source data during the design of the semistructured views, the designed views may violate the source semantics and become invalid. For example, since the designed views do not distinguish the attributes of object classes and attributes of relationship types, they are unable to maintain the integrity of relationship types in the source data. Further, existing systems do not address the problem of reversible views. A view is said to be reversible if the original source schema can be produced back by applying some operators to the view. Without a mechanism to guarantee the reversibility of the view, users will not be able to produce the original source schema back.

[Chen Y. B. 2002] puts forth an initial proposal to design valid XML views over native XML data. A conceptual schema for the source data is first extracted based on a semantically rich data model, the Object-Relationship-Attribute model for Semistructured data (ORA-SS) [Dobbie G. 2000]. The ORA-SS model is able to capture semantics that are not supported in data models such as OEM [Papakonstantinou Y. 1995], Dataguide [McHugh J. 1997] or XML DTD/Schema.[1] Next, XML views are created by applying four operators, select, drop, join and swap, on the source ORA-SS schema. The select and join operators are analogous to the select and join operators in relational data model. The drop operator is the opposite of the project operator in relational data model. The fourth operator swap is unique in semistructured data as it interchanges the positions of parent and child object classes. The swap operator raises the issue of view reversibility. That is, when we swap two object classes to construct a view schema, we can reconstruct

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

the original source schema from the view by carrying out a reverse swapping.

In this paper, we examine the problem of maintaining source semantics in the design of valid and reversible semistructured views. We present the complete set of rules with proofs to guarantee that the designed views are meaningful and reversible when any of the view operators are applied. We also further develop rules to ensure the correctness of the participation constraints of relationship types in the views. The proposed approach not only enables users to design flexible semistructured views, but also guarantees the designed views are meaningful and reversible.

The rest of the paper is organized as follows. Section 2 reviews the ORA-SS data model. Section 3 demonstrates how the semantics captured in ORA-SS allows us to design valid and reversible semistructured views. Section 4 presents the rules to maintain the semantics of the views for each operator. We also examine the reversible view problem in this section. Rules for the evolution of the participation constraints of object classes in relationship types are given in Section 5. Finally, we conclude in Section 6.

2. ORA-SS DATA MODEL

The ORA-SS model [Ling T. W. 2005] comprises of three basic concepts: object classes, relationship types and attributes. An *object class* models a set of real world entities and is related to other object classes through *relationship types. Attributes* are properties that describe an object class or a relationship type.

The ORA-SS schema diagram is a directed graph where each internal node is an object class and each leaf node is a complex attribute or an attribute. An object class is similar to an entity type in an Entity-Relationship diagram or an element in XML documents. It is represented as a labelled rectangle in an ORA-SS schema diagram. A relationship type describes a relationship among object classes in one hierarchical path. Each relationship type has a degree and participation constraints. The ORA-SS diagram uses a solid labelled directed edge connecting object classes to denote a relationship type. ORA-SS can express n-ary ($n \geq 2$) relationship types implied in XML source data.

An attribute of an object class or a relationship type is represented as a circle attached to the object class or the lowest participating object class of the relationship type. There can be many different types of attributes in ORA-SS schema, such as object identifier attributes, single-valued attributes and multi-valued attributes, etc. In ORA-SS schema, an object identifier attribute is denoted as a filled circle. A single-valued attribute is denoted as a circle. A multi-valued attribute is denoted as a circle with a star symbol *(0:n) or a plus symbol +(1:n) inside.

The ORA-SS data model also provides various notations to express the complex object structures: a circle with "ANY" symbol for attributes with unknown structure or whose structure is heterogeneous, a circle with a "|" symbol for disjunctive attribute, a diamond with a "|" symbol for disjunctive relationship type, a diamond with a "IDD" symbol for weak object class, a rectangle with a "<" symbol for the ordering on the attribute of object class, etc.

The following example illustrates the essence of ORA-SS.

EXAMPLE 1. *Figure 1 depicts an ORA-SS source schema, which is extracted from an XML document. This schema contains six object classes, such as part,*

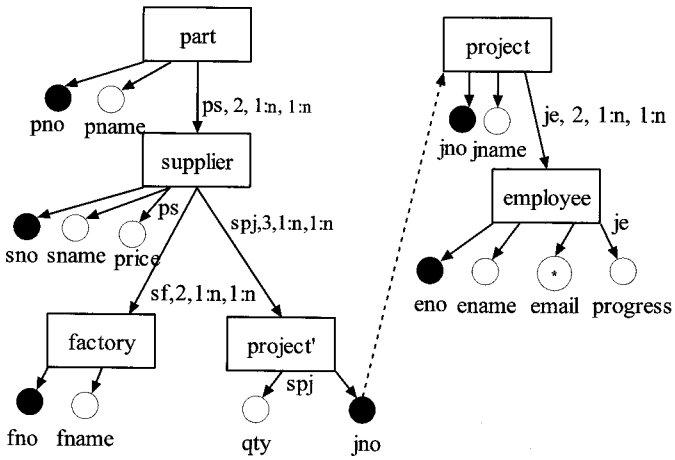


Figure 1. An ORA-SS source schema.

supplier and project, etc, which are represented as rectangles. Each object class has an object identifier attribute, such as the object identifier attribute pno of part. In addition, the attribute email of employee is a multi-valued attribute indicated with a star symbol.

There is a binary relationship type between object class part and supplier in Figure 1, which is labeled as "ps, 2, 1:n, 1:n" on the incoming edge of supplier, where ps denotes the name of the relationship type between supplier and part, 2 indicates the degree of the relationship type. The first "1:n" indicates the child participation constraints in the relationship type. That is, there can be minimum one supplier and maximum many suppliers for one part. The second "1:n" then indicates the parent participation constraints in the relationship type. That is, there can be minimum one part and maximum many parts supplied by one supplier. The relationship type ps also has one attribute called price in Figure 1. This attribute price is determined by both part and supplier and is attached to supplier with label ps on the incoming edge of the attribute.

In addition, there is a ternary relationship type "spj, 3, 1:n, 1:n" labeled on the incoming edge of project', which involves object classes supplier, part and project'. The first "1:n" in the relationship type indicates each pair of part and supplier can have minimum one project' and maximum many project's. The second "1:n" in the relationship type indicates one project' can have minimum one pair of part and supplier and maximum many pairs of part and supplier. The relationship spj also has one attribute qty attached to project', which indicates the quantity of a part supplied by a supplier in a project'. In general, the participating object classes of a relationship type are not explicitly presented on the label for the relationship type. However, when the participating object classes are not next to each other in the path of the schema, the participating object classes will then be explicitly expressed on the label such as dc(department, course), 2, 1:n, 1:1. (see Figure 3).

The dotted line between object classes project' and project denotes a reference from project' to project. This reference indicates that each object identifier value

of project' is referring to an object identifier value of project. It thus removes the duplicates of the attribute jname below project'. For the purpose of simplicity, the schema in Figure 1 only shows a reference between project' and project. As a matter of fact, there can also be references for object classes supplier and factory to remove the duplicates of sname and fname, respectively. □

The above example demonstrates that the ORA-SS model is able to capture semantics implied in XML or semistructured data. The major advantages of ORA-SS over existing semistructured data models such as OEM and Dataguide are its ability to distinguish between attributes and object classes, differentiate between attributes of object classes and attributes of relationship types, as well as express the degree of relationship types and the participation constraints on the object classes in the relationship types. These semantics which are explicitly expressed in the ORA-SS data model are important for designing "good" semistructured databases and defining meaningful views.

3. MOTIVATING EXAMPLE

Suppose we have the source schema as shown in Figure 2, and we want to design a view that swaps the object classes *course* and *student*. Figure 3 shows the resulting view which is not only valid but is also reversible.

Note that the participating object classes in the relationship type dc in Figure 3 have to be explicitly stated in the views as *"dc(department, course), 2, 1:n, 1:1"*. This is because these object classes are not located next to each other in the paths. There is an object class student between them in the same paths. If the participating object classes of the relationship type *dc* in Figure 3 are not specified, then the default participating object classes will be *student* and *course*.

A valid view requires that the attribute *grade* be moved down and be attached to *course* (as shown in Figure 3) to keep the semantics of the relationship type *cs* intact. In addition, the object class *lecturer* also needs to move down with *course* in Figure 3 to keep the semantics of the relationship type *cl* intact. The meaning of the attribute *workload* is still the same as in the source schema, that is, the workload of a lecturer under a course. Note that we do not need to move the object class *tutor* up with *student* although *tutor* and *student* are involved in the relationship type *cst*. This is because *tutor* needs to be attached to the lowest participating object class of *cst*, i.e., *course*. Thus, the semantics of the ternary relationship type *cst* remains unchanged.

Next, we illustrate an example of reversible views by applying another swap operator to swap *student* and *course* in Figure 3. The attributes of *student* and *course* will move together with their owner object classes. The relationship attribute *grade* is thus attached to the object class *student* again. Further, the object class *lecturer* will move up with *course* as a whole, thus keeping the semantics of the relationship type *cl* intact. The view obtained will be the same as the original source schema in Figure 2. We say that the view in Figure 3 is a reversible view of original source schema in Figure 2 since we can reproduce the original source schema from it.

Let us now consider the case of an invalid view. We have observed that it is important to distinguish between the attributes of object classes and attributes of relationship types. However, XML DTD and OEM do not differentiate these two

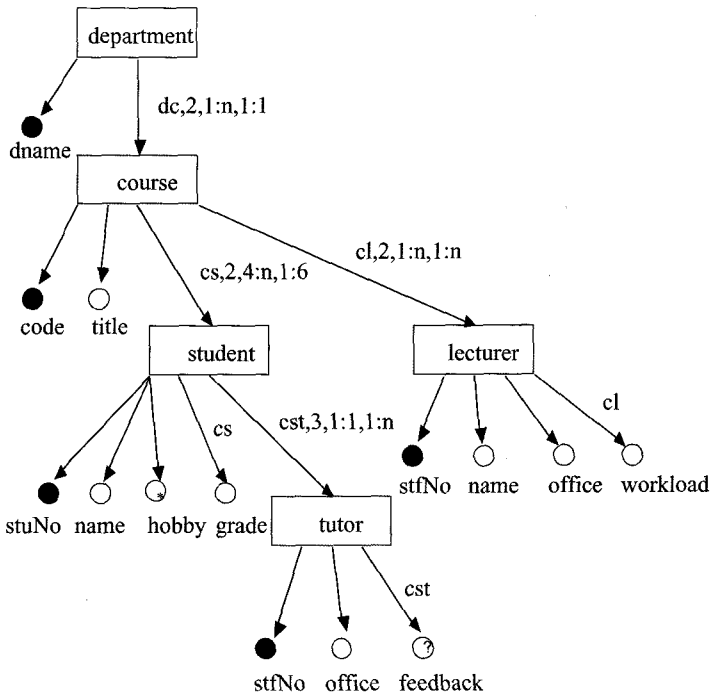


Figure 2. An ORA-SS source schema of course-student-lecturer.

types of attributes. If we design a view based on XML DTD or OEM graph, we will probably move the attribute *grade* (which is an attribute of the relationship type *cs*) together with the object class *student* when we swap the object classes *course* and *student*. The resulting view is shown in Figure 4. The view obtained in Figure 4 violates the functional dependency: $\{stuNo,\ code\} \rightarrow grade$ in the source schema.

Further, the source schema in Figure 2 has a binary relationship type called *cl* that involves *course* and *lecturer*. Without this additional information regarding the *cl* relationship type, we will probably keep the object class *lecturer* in the same position after swapping *course* and *student*. That is, *lecturer* is attached to *student* in the view. The relationship type *cl* will be lost in the view (see Figure 4) as *course* and *student* are now in two different paths. Thus, all the distinct *lecturers* will be repeatedly placed under each *student* in the corresponding XML view documents. Further, the attribute work load will become meaningless as it wrongly becomes an attribute of *lecturer* in the view and has nothing to do with *course*. For these reasons, the view in Figure 4 is invalid.

The above example illustrates the importance of maintaining semantics when designing semistructured views. By properly maintaining the semantics in the views, such as moving relevant attributes or object classes in the views, we can ensure that valid and reversible semistructured views are designed.

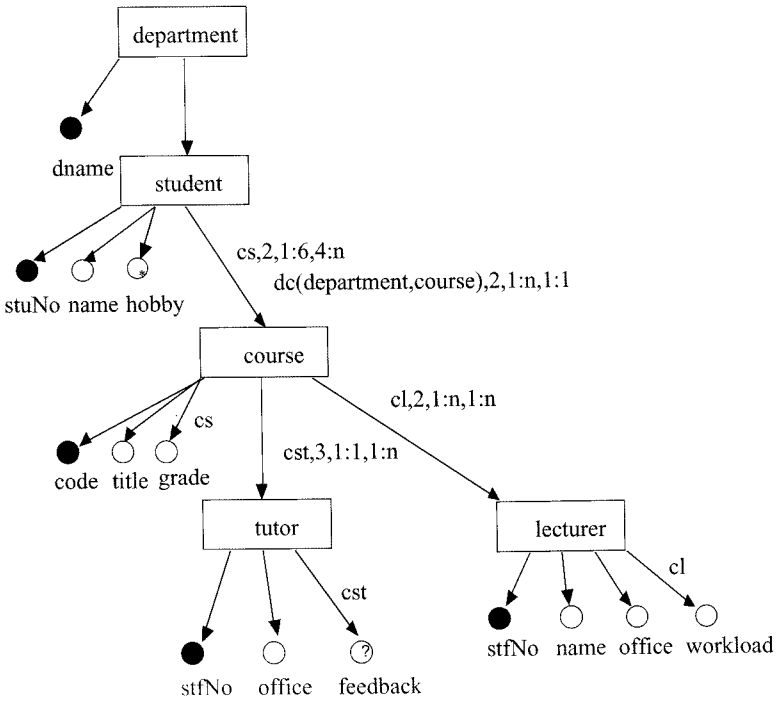


Figure 3. A valid and reversible ORA-SS view schema obtained by swapping *course* and *student* in Figure 2.

4. VIEW DESIGN RULES

In our context, a semistructured view conforms to an ORA-SS view schema. Thus, a semistructured view is valid if and only if its corresponding ORA-SS view is valid. In other words, an XML view is valid if and only if it conforms to a valid ORA-SS view.

Definition 4.1. **(Valid Semistructured Views)** Given an semistructured source data D, let S be the ORA-SS source schema extracted from D, V be an ORA-SS view based on S, and SV be an semistructured view conforming to V, SV is said to be *valid* iff its corresponding ORA-SS view V is *valid*.

Based on the above definition, the problem of valid XML views becomes the problem of valid ORA-SS views. In this section, we will present the rules to maintain the validity of ORA-SS views when *select, drop, join, swap* operators are applied on the source schema.

4.1 The Select Operator

The select operator is similar to select operator in relational data model. It filters data by applying predicates on attributes in an ORA-SS schema. There is no restructuring of the schema. Thus, the view schema will not violate semantics in the source schema, and we do not need rules for the validity of views when select operations are applied.

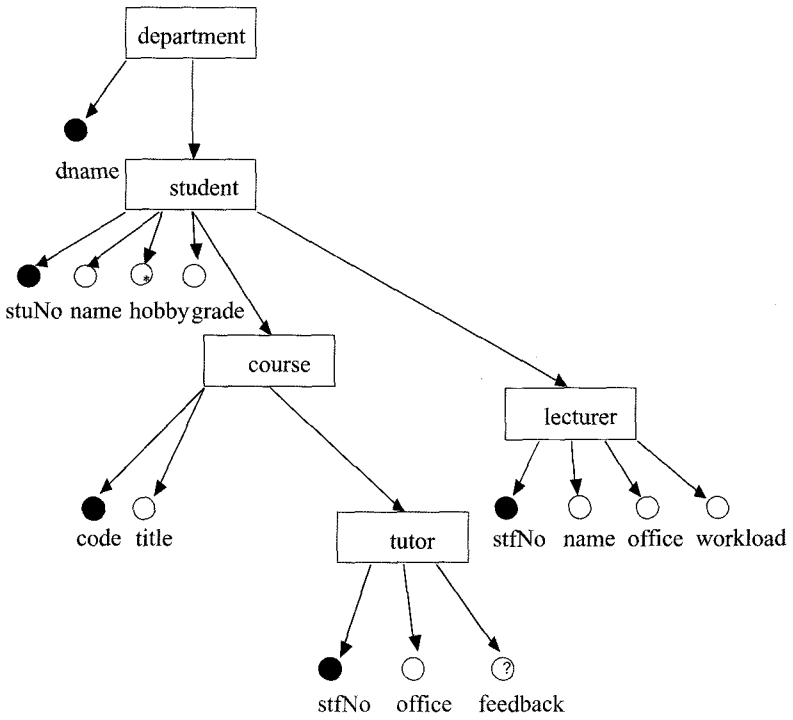


Figure 4. An invalid ORA-SS view schema obtained by swapping *course* and *student* in Figure 2.

EXAMPLE 2. *Suppose we design a view that applies a select operation (qty > 500) on the source schema in Figure 5. The resulting view schema is shown in Figure 6. This view will retrieve those suppliers, parts and projects together the relationship and attributes as shown in Figure 5, where each supplier supplies some part for some project with quantity larger than 500.* □

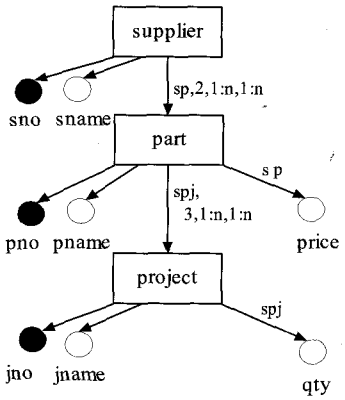


Figure 5. An ORA-SS source schema of supplier-project-part.

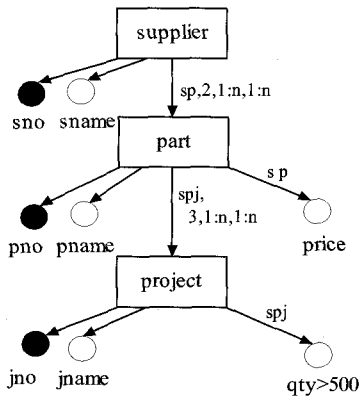


Figure 6. An ORA-SS view applied with a select operation applied in Figure 5.

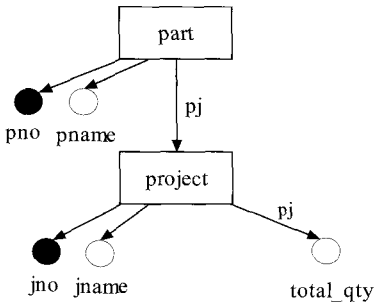


Figure 7. An ORA-SS view dropping supplier in Figure 5.

4.2 The Drop Operator

The drop operator drops object classes or attributes in the source schema. It is opposite to the project operator in relational data model. The drop operator will affect relationship types that involve the dropped object class. The following example illustrates the case where a drop operator is applied.

EXAMPLE 3. *Figure 7 shows a view that is based on the ORA-SS source schema in Figure 5. The object class supplier has been dropped. This view indicates that for a given part, all those projects which need this part are placed below as its children. Obviously, the attributes of supplier (i.e., sno and sname) have to be dropped too in the view schema as attributes cannot exist without its owner object class.*

We also have to remove the relationship type sp and spj since both of them involve the dropped object class supplier. The attribute of the relationship type sp (i.e., price) is also dropped in the view schema in Figure 7. Further, the attribute of the relationship type spj (i.e., qty) is mapped to an aggregate attribute (i.e., total_qty), which represents the total quantity of one part in a given project. It is actually an attribute of a new derived relationship type involving only project and part, which is derived from spj. □

This example shows that flexible views can be designed based on ORA-SS with its additional semantics. The following four rules guarantee the validity of XML views when drop operators are applied.

Rule Drop_1: *If an object class O in a source schema is dropped in designing a view; then the attributes of O are dropped too in the view.*

Rule Drop_2: *If an object class O in a source schema is dropped in designing a view; then each relationship type involving O is dropped too in the view.*

If a participating object class of a relationship type is dropped in the view, the relationship type will be broken. Although the relationship type will not be shown in XML document or XML schema, it needs to be dropped to keep the semantics in the ORA-SS view schema consistent.

After a relationship type is dropped, the rest of the object classes of the relationship type still have semantic connection in the view. The rules Drop_3 and Drop_4 specify how we can derive a new relationship type from the dropped relationship type.

Rule Drop_3: *If an object class O in a source schema is dropped in designing a view; then for each n-ary ($n \geq 2$) relationship type R involving O, a new relationship type is generated by projecting R on all object classes except O, and the attributes of R can be dropped, or mapped into attributes with aggregate function, or mapped into attributes typed in bag of values.*

Rule Drop_4: *If an object class O in a source schema is dropped in designing a view; then if O is the only common participating object class of two relationship types R_1 and R_2, and all participating object classes of R_1 and R_2 are in a continual path, and the participating object classes of R_1 is not a subset of the participating object classes of R_2 and vice versa; then a new relationship type is generated by joining R_1 and R_2 based on the object class O.*

Correctness of Rule Drop_3: Suppose object classes O_1, O_2, ..., O_n participate in a relationship type R in a source schema. We assume one of the object classes (say $O_i, i = 1, ..., n$) is dropped in designing a view. According to Rule Drop_3, a new relationship type R' is derived by projecting out O_i and all $R's$ attributes from R:

$$R' = \prod_{O_1,\ldots,O_{i-1},O_{i+1},\ldots,O_n} R$$

All the rest of the object classes of R except O_i are kept in the new relationship type. Obviously, R' does not violate the semantics implied in R according to the theory of relational data model. It is because the new relationship type keeps the rest of the semantic connection among the object classes in the view. The attributes of R can be dropped as it will not violate any semantics of the new relationship type R'. However, new attributes can be derived for R' via aggregate functions. In this way, the semantics among the rest object classes of R is correctly kept in the view and the view is still valid. □

EXAMPLE 4. *Consider the ORA-SS source schema in Figure 8. It contains three object classes: part, supplier and project. There is one binary relationship type called ps involving part and supplier, and another ternary relationship type called spj involving all the three object classes. The symbol "+" is a shorthand to indicate the 1:n participation constraint. Suppose we design a view by dropping object class supplier. According to the rules Drop_1 and Drop_2, the attributes of supplier and relationship types involving supplier are dropped too. Figure 9 shows a view of Figure 8 where we derive a new relationship type called pj from spj based on Rule Drop_3. A new attribute called total_qty is derived for pj by applying a sum function to the attribute qty in the source schema. Thus, this view keeps the semantic connection between part and project and is valid.*

Without Rule Drop_3 and the semantics captured in the ORA-SS source diagram, we may produce a view as shown in Figure 10, which does not indicate what relationship type exists between part and project, and attribute qty is still attached to the object class project. The attribute qty in Figure 10 violates the functional dependency {pno, sno, jno} → qty in the source schema. Thus, the view in Figure 10 is an invalid view. □

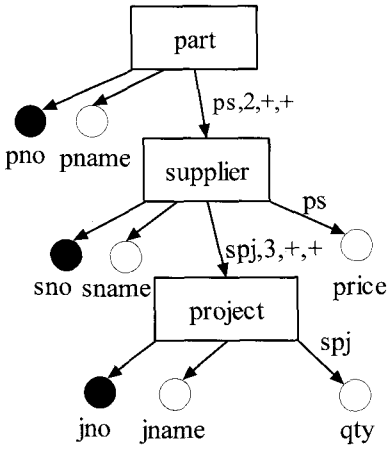


Figure 8. An ORA-SS source schema.

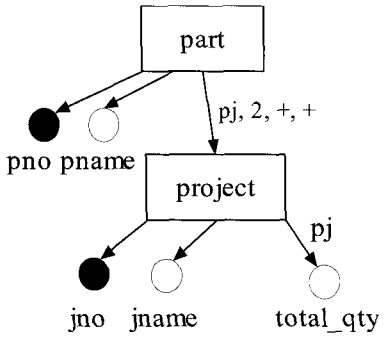


Figure 9. A valid view schema obtained by dropping supplier.

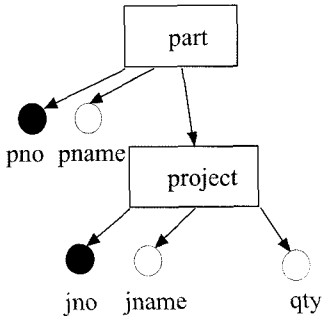


Figure 10. An invalid view schema obtained by dropping supplier.

The above example illustrates Rule Swap_3 and shows that an invalid view may be produced without this rule. In more complicated cases, the dropped object class may involve more than one relationship type in the source schema. Thus, we need to join those relationship types to keep the semantic connection among them. This is achieved by Rule Drop_4.

Correctness of Rule Drop_4:

(1) We will first show why the conditions in Rule Drop_4 are necessary. Suppose the first condition is false. That is, there are other common object classes for R_1 and R_2. Obviously, we do not have to join R_1 and R_2 in this case as the semantic connection between R_1 and R_2 is still explicitly expressed through the other common object classes.

Now suppose the second condition is false. That is, all participating object classes of R_1 and R_2 are not in the same path. In this case, we cannot join R_1 and R_2 as the object classes of the new relationship type will not be in the same path and the new relationship type will be meaningless.

Finally, suppose the third condition is false. Then either all participating object classes of R_1 participate in R_2 or vice versa. In this case, if the only common object class of R_1 and R_2 is dropped, all object classes of R_1 must have been removed in the view schema. Thus, we do not need to join R_1 and R_2 in the view schema.

(2) Next, we examine the validity of view obtained. Suppose $O_{11}, O_{12}, ..., O_{1n}$ participate in the relationship type R_1 in the order from ancestor to descendant and $O_{21}, O_{22}, ..., O_{2m}$ participate in the relationship type R_2 in the same order in the source schema. R_1 and R_2 are in the same path in the. We assume $O_{1n} = O_{21}$ is the only common object class of R_1 and R_2, which is dropped in designing a view. Satisfying all conditions of the Rule Drop_4, we derive a new relationship type R' in the view schema by joining R_1 and R_2 as follows:

$$R' = \prod_{o_{11}, o_{12}, ..., o_{21}, ..., o_{2m}} (R_1 \bowtie_{o_{1n}=o_{21}} R_2)$$

Notice the join operator is actually an equijoin for R_1 and R_2 based on the identifiers' value of O_{1n} and O_{21}. The derived relationship type R' does not violate the semantics of R_1 and R_2 according to the theory of relational data model. Instead, it keeps semantic connection among the two relationship types. Thus, the semantics among the rest object classes participating in R_1 and R_2 in the view is correctly kept and the view is still valid. □

The following example illustrates the Rule Drop_4 and shows an invalid view may be produced if the rule is not applied.

EXAMPLE 5. *Consider the ORA-SS source schema in Figure 11. There are three object classes project, staff and publication. The binary relationship type js between project and staff indicates which staff participates in a project. The binary relationship type sp between staff and publication indicates the publications of a staff.*

Suppose we design a view that drops the intermediate object class staff. In fact, there is still semantic connection between project and publication based on the source schema, which indicates all publications published by those staff participating in a given project. Thus, we need to generate a new derived relationship type jp between the two object classes in the view. In particular, jp is generated as follows.

$$jp = \prod_{project, publication} (js \bowtie_{js[staff]=sp[staff]} sp)$$

Note that the Join operator is an equijoin for js and sp based on the identifiers' value of staff in js and staff in sp. In this way, the rest two object classes are connected together through the semantics and the view is still meaningful. The valid view with the new relationship type is shown in Figure 12.

Without Rule Drop_4, we do not know the relationship because it is not clear how the view is derived from the source schema (see Figure 13). In this ambiguous view schema, the semantic connection between the two object classes will be lost. □

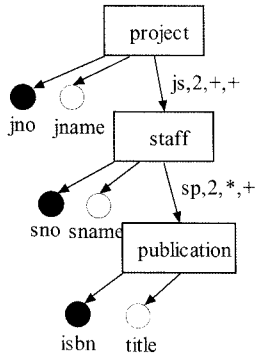


Figure 11. An ORA-SS source schema.

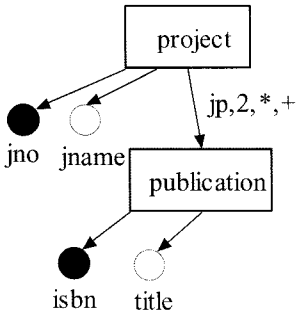


Figure 12. A valid view schema.

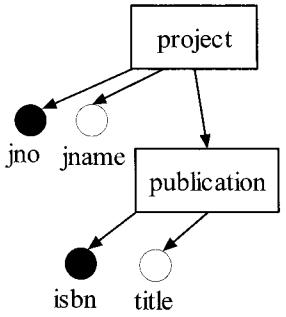


Figure 13. An ambiguous view schema.

4.3 The Join Operator

Referencing object classes and referenced object classes can occur in an ORA-SS source schema. Two object classes can be connected together by foreign key to key reference in the schema. Thus, these two object classes can be joined together with a join operator. When a join operator is applied, we remove the referenced object class in the view schema and attach all attributes of the referenced object class to the referencing object class.

EXAMPLE 6. *Figure 14 shows an ORA-SS source schema diagram. The object class supplier′ under project refers to an object class supplier with the object identifier attribute of supplier, denoted by a dotted line. There is a relationship type between retailer and supplier called rs, which has an attribute contract under retailer. The meaning of the relationship type is that for a given supplier, all the retailers having contracts with the supplier will be placed below as its sub-elements.*

Figure 15 depicts a view, which joins object classes supplier and supplier′ together. The join operator attaches the attributes sno and sname of supplier to supplier′ in the view. In addition, the object class retailer also moves below supplier′ and its attribute contract moves with retailer. As supplier′ refers to supplier with a foreign key to key reference in the source schema, supplier′ can play the role of supplier in the view schema. Thus, the relationship type rs between supplier and retailer are still kept in the view and actually become the relationship type between supplier′ and retailer. □

When a join operator is applied, we need to handle the object classes and relationship types in the path of the referenced object class. We develop two rules for the join operator. The first rule handles the descendants of the referenced object class and their relationship types. The second rule handles the ancestors of the referenced object class and their relationship types.

Rule Join_1: *If a referencing object classes O_i is joined with a referenced object class O_j in designing a view; then all attributes of O_j are attached to O_i in the view, and if there is a relationship type R involving no ancestors of O_j but descendants of O_j; then*

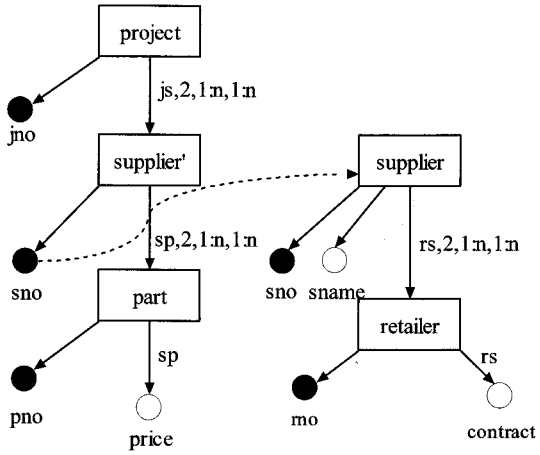


Figure 14. An ORA-SS schema diagram.

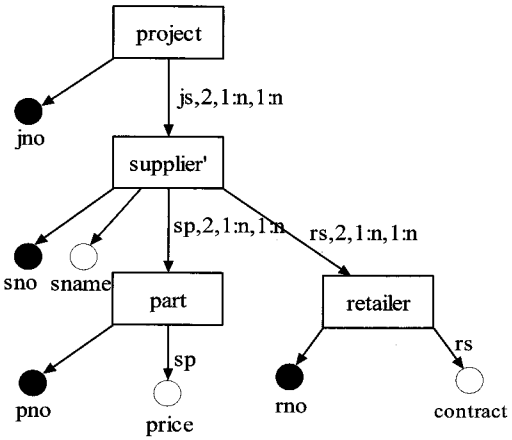


Figure 15. An ORA-SS view schema by joining *supplier'* and *supplier* in Figure 14.

—**Case 1**: *Keep R and all its participating object classes in the view.*

—**Case 2**: *Drop some of the object classes of R in the view to derive a new relationship type, and the attributes of R can be dropped, mapped into attributes with some aggregate function, or mapped into attributes typed in bag of values.*

Correctness of Rule Join_1. Rule Join_1 first attaches the attributes of O_j to O_i as O_i refers to O_j by a foreign key to key reference and O_i plays the role of O_j in the view. Next, it handles the relationship types involving descendants of O_j in the view. There are two cases for the relationship types. Suppose one of the relationship types is R. In Case 1, R is kept in the view. Thus, all participating object classes of R are also kept in the view and O_i plays the role of O_j in R. Thus, the semantics of R is still kept in the view and the view is valid.

In Case 2, a new relationship type is derived from R by dropping some of the participating object classes of R. The attributes of R can be handled properly based on users' requirements. According to Rule Drop_3, the new relationship type does not violate the semantics of R and the view is valid. □

We also need to handle the ancestors of O_j in the source schema and their relationship types, especially when the ancestors of O_j participates in relationship type with O_j or its descendants.

Rule Join_2: *If a referencing object class O_i is joined with a referenced object class O_j in designing a view; then all attributes of O_j are attached to O_i in the view, and if there is a relationship type R involving ancestors of O_j, then:*

—**Case 1**: *Keep R in the view and swap the ancestors of O_j involving R below O_j.*

—**Case 2**: *Drop the ancestors of O_j involving R in the view to derive a new relationship type, and the attributes of R can be dropped, mapped into attributes with some aggregate function, or mapped into attributes typed in bag of values.*

Correctness of Rule Join_2. Rule Join_2 handles the relationship types involving ancestors of O_j in the view. There are also two cases for processing the relationship types. Suppose one of the relationship type is R. In Case 1, R and the ancestors of O_j participating in R are needed in the view schema. Thus, the ancestors must be swapped first and become descendants of O_j so that they can be attached as O_i's descendants in the view schema. In this way, R is kept intact in the view and the view is valid. Notice a new operator, i.e. swap operator is utilized in this case. More details on swap operator will be given in the next sub section.

In Case 2, we simply drop all the ancestors of O_j involving R in the view. As O_i has its ancestors in the view already, the ancestors of O_j in the source schema cannot appear as ancestors of O_i in the view. After the drop of the ancestors, a new relationship type can be derived from R and the attributes can be handled properly in the view schema. In this way, the view will be kept valid. □

Without the two rules, invalid views may be produced, as the following example will illustrate.

EXAMPLE 7. *Figure 16 depicts an ORA-SS source schema that has a foreign key to key reference between object classes supplier′ and supplier. The ternary relationship type ysr involves object classes year, supplier and retailer. The attribute contract belongs to the relationship type ysr in the source schema. Suppose we design a view by joining object classes supplier′ and supplier.*

By applying Rule Join_2, we can design a valid view that joins the object classes supplier′ and supplier (see Figure 17). A new relationship type is derived from ysr, which involves supplier′ and retailer only. The attribute contract becomes a multi-valued attribute of the new relationship type where all the contracts signed by a given supplier′ and retailer in each year are aggregated into a bag of values.

However, the view schema in Figure 18 shows that the object class year does not exist and the relationship type ysr is unchanged. This thus violates Rule Join_2. In this case, this relationship type is meaningless in the view schema since one of its participating object classes year is not included in the view.

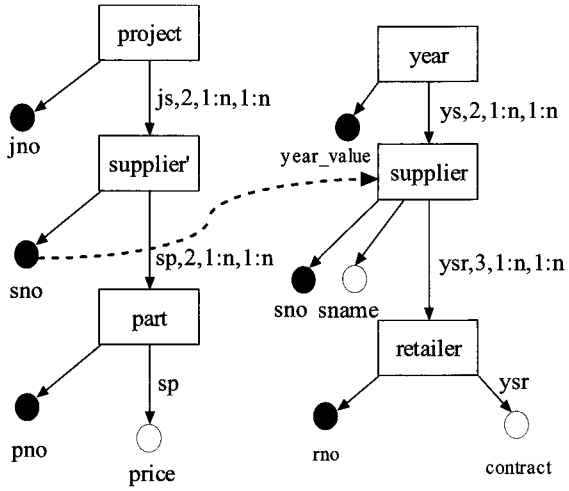


Figure 16. An ORA-SS source schema.

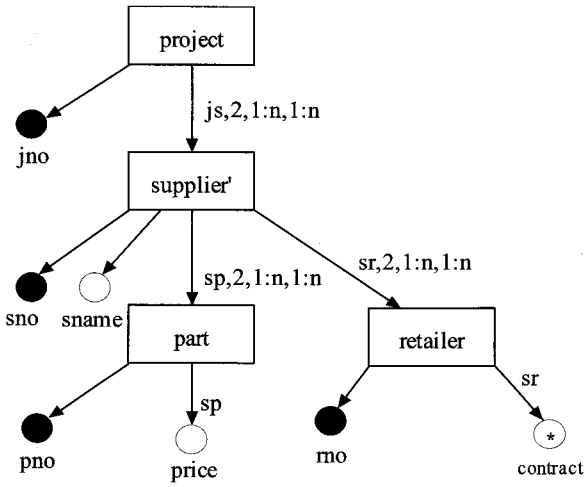


Figure 17. A valid view schema by joining *supplier'* and *supplier* in Figure 16.

The meaning of the attribute contract in the source schema is a contract signed by a supplier and a retailer in a given year. However, in the view schema in Figure 18, the meaning of the attribute contract is changed, which is an attribute of the ternary relationship type ysr involving object classes retailer, supplier and project. That is, it indicates a contract signed by a project, a supplier, and a retailer without any year specified. Thus, the attribute contract in the view schema in Figure 18 has a different meaning in the source schema. The view in Figure 18 is an invalid view. □

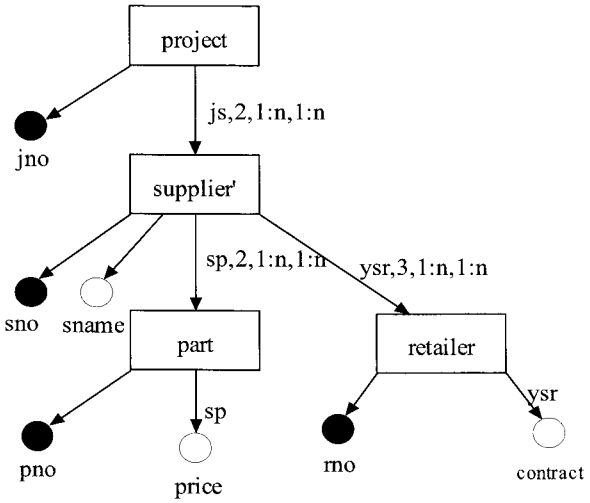


Figure 18. An invalid view schema by joining *supplier'* and *supplier*.

4.4 The Swap Operator

The swap operator restructures the source schema by exchanging the positions of a parent object class and its child object class. It is unique in XML because it can be applied only in hierarchical structure. Further, swap operator also introduces the issue of view reversibility. That is, when we swap two object classes to construct a view schema, we can reconstruct the original source schema from the view by carrying out a reverse swap.

EXAMPLE 8. *Figure 19 shows a source schema involving the object classes supplier, part and project. Suppose we want to design a view that swaps supplier and project hierarchically. Figure 20 shows the view obtained. After supplier and project have been swapped, we need to ensure that their attributes are relocated properly.*

It is clear that the attributes sno and sname should move together with their owner object class supplier. Likewise, the attributes jno and jname should move together with their owner object class project. However, the attribute price, which belongs to the relationship type sp, must remain with the new child object class of sp, that is, supplier in order to preserve the semantics of the source schema, that is, the functional dependency, $\{sno, pno\} \rightarrow price$.

If the attribute price remains with the object class part, then it will violate the functional dependency in the source schema. Similarly, the attribute qty of the relationship type spj is attached to the lowest participating object class of spj, that is, supplier. □

When a swap operator is applied to design a view, we not only need to maintain semantics for the view, but we also need to address the issue of reversible view. We develop a set of rules to meet both requirements in this section.

Rule Swap_1: *If an object classes* O_i *and its descendant object class* O_j *in a source schema are swapped in designing a view; then the attributes of* O_i *and* O_j

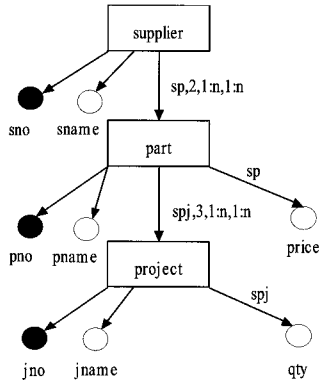


Figure 19. An ORA-SS source schema.

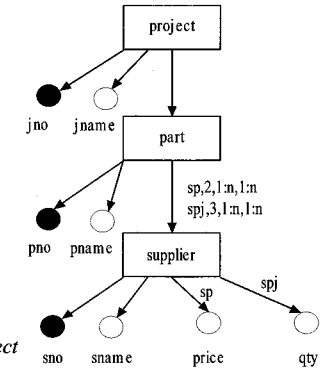


Figure 20. An ORA-SS view schema by swapping *supplier* and *project* in Figure 19.

must remain attached to O_i and O_j respectively in the view.

Rule Swap_1 is straightforward and ensures that the attributes of O_i and O_j do not become meaningless in the view after O_i and O_j are swapped.

We observe that the relationship types in the source schema that involve O_i and/or O_j are affected since the hierarchical positions of O_i and O_j have been interchanged after a swap operator is applied. Given two object classes O_i and O_j where O_j is a descendant of O_i in an ORA-SS schema, the relationship types that are affected after a swap of O_i and O_j can be classified into the following three categories.

(1) The first category is the set of relationship types which do not involve any other object classes but O_i and/or O_j and/or the ancestors of O_i or O_j in the ORA-SS source schema. In other words, these relationship types involve object classes that occur in the straight path of O_i and O_j (see Figure 21).

(2) The second category is the set of relationship types which involve at least both O_i and object classes in the branch paths between O_i and the parent of O_j, as shown in Figure 22.

(3) The third category is the set of relationship types which involve at least both O_j and its descendants, as shown in Figure 23.

These three categories of affected relationship types are handled by the rules Swap_2, Swap_3 and Swap_4, respectively. These rules not only maintain the semantics of the views, but also guarantee the reversibility of the views.

When O_i and O_j are swapped, all object classes of a relationship type in the first category are still in one same path. However, the lowest object class of these relationship types will be changed. Thus, we need to handle the attributes of these relationship types properly. Rule Swap_2 processes these relationship types.

Rule Swap_2: *Suppose an object classes O_i and its descendant object class O_j in a source schema S are swapped in designing a view. Let S be the set of relationship types which do not involve any descendants of O_j, but involve the ancestors of O_i*

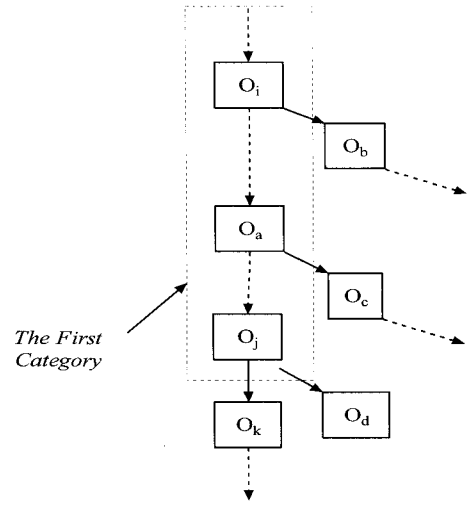


Figure 21. *First category* of affected relationships in an ORA-SS source schema.

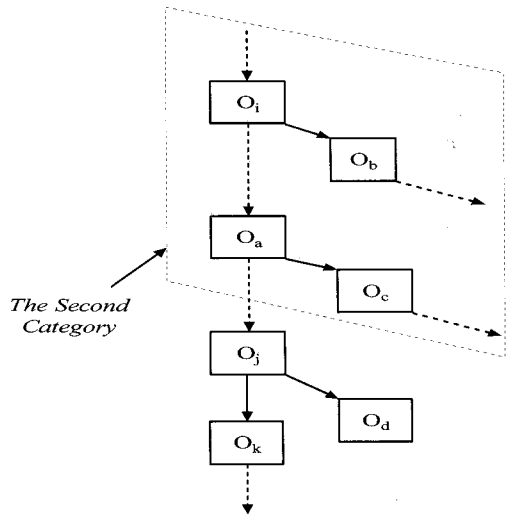


Figure 22. *Second category* of affected relationships in an ORA-SS source schema.

or O_j in the source schema. For each relationship type R in S, the attributes of R are attached to the lowest participating object class of R in the view.

On the other hand, when O_i and O_j are swapped, all object classes of a relationship type (if any) in the last two categories may not be in one same path or some gap may be produced in between them.

Rule Swap_3: *Suppose an object class O_i in a source schema is swapped with its descendant object class O_j in designing a view. If there exists a relationship type which involves at least O_i and O_c, where O_c is a descendant of an object class O_a*

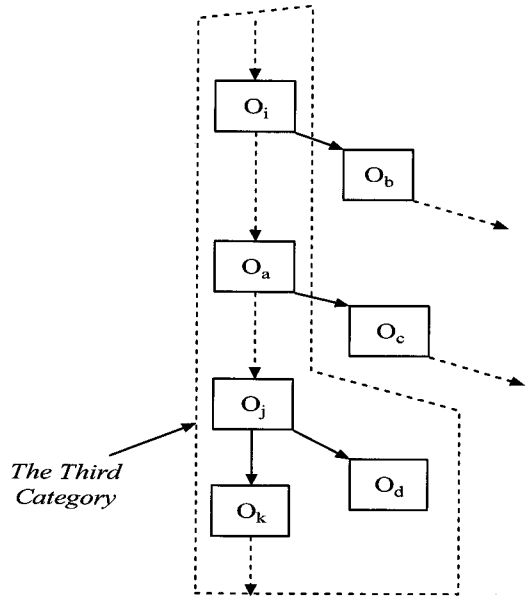


Figure 23. *Third category* of affected relationships in an ORA-SS source.

that lies in the path between O_i and O_j (including O_i) but O_c does not lie in the path between O_i and O_j in the ORA-SS source schema, then the subtree rooted at O_c is attached to O_i in the view.

This rule handles relationship type in the second category. Note O_a may be O_i itself. We study the correctness of this rule.

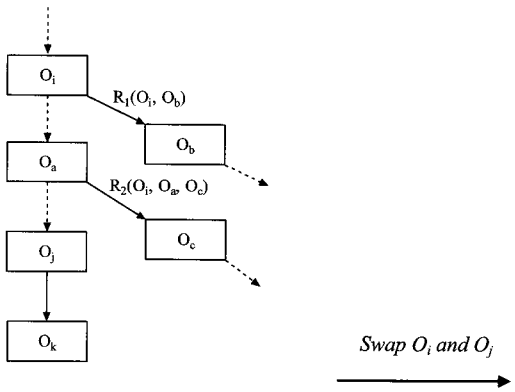


Figure 24. An ORA-SS source schema for swapping O_i and O_j

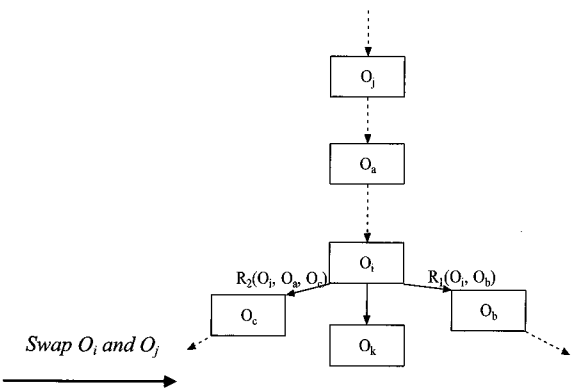


Figure 25. An ORA-SS view schema for swapping O_i and O_j

Correctness of Rule Swap_3. Figure 24 depicts a simplified ORA-SS source schema, which contains two relationship types in the second category. More specifically, there is one relationship type R_1 involving O_i and its child O_b, where O_b is

not in the path between O_i and O_j. Obviously, R_1 is in the second category. It is depicted as $R_1(O_i, O_b)$ for simplicity. The second relationship type R_2 involves O_i, O_a and O_c, where O_c is a child of O_a. Notice O_a is in the path between O_i, and O_j and O_b is not in the path. R_2 is thus still in the second category and is depicted as $R_2(O_i, O_a, O_b)$ for simplicity. When O_i and O_j are swapped to design a view as shown in Figure 25, sub trees rooted at O_b and O_c need to move with O_i and are attached to O_i to keep the semantics of R_1 and R_2 intact in the view. Otherwise, the relationship types will be broken in the view. In addition, if there are any attributes of R_1 and R_2 attached to O_b and O_c, the attributes will also be attached to O_b and O_c in the view. Thus, the Rule Swap_3 appropriately maintains the semantics in the view and the view is still valid. Further, the preserved relationship types in the view also make the reversible view possible. In other words, if the second swap operator is applied to swap O_i and O_j in the view, the original source schema may be produced back because the two relationship types are kept intact in the view. The next rule will handle such cases so that reversible views are guaranteed to be produced. □

Rule Swap_4: *Suppose an object class O_i in a source schema is swapped with its descendant object class O_j in designing a view. For each child O_d of the object class O_j, let T be the subtree that is rooted at O_d. Let S be the set of relationship types which involve at least O_j and its descendants in T. If O_l is the lowest participating object class among the relationship types in S that lie in the path between O_i and O_j after the swap, then the subtree rooted at O_d is attached to O_l.*

This rule handles relationship types in the third category. More specifically, this rule maintains the semantics in all the relationship types for each subtree rooted at a child of O_j. We demonstrate the correctness of the rule as follows.

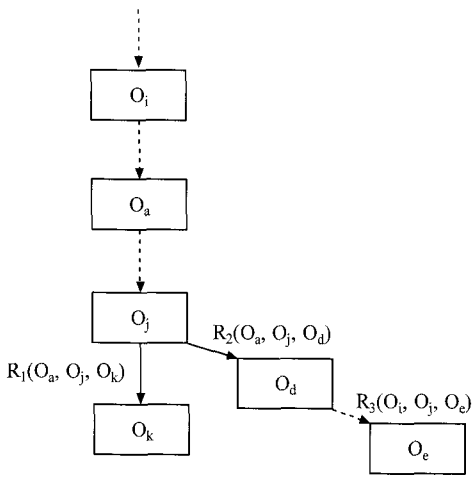


Figure 26. An ORA-SS source schema for swapping O_i and O_j.

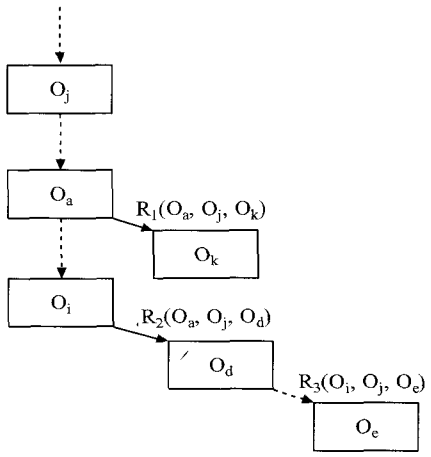


Figure 27. An ORA-SS view schema for swapping O_i and O_j.

Correctness of Rule Swap_4. Figure 26 depicts a source schema having relationship types (R_1, R_2 and R_3) in the third category, that is, they involve at least O_j and the descendants of O_j. O_j has two children in the source schema, namely O_k and O_d. Suppose we design a view swapping O_i and O_j, as shown in Figure 27. We will maintain all the relationship types in one subtree rooted at a child of O_j at one time. Firstly, for subtree rooted at O_k in the source schema, there is only one relationship type R_1, which involves O_a, O_j and O_k. Since O_a is the lowest participating object class of R_1 after the swap, the subtree rooted at O_k is attached to O_a. In contrast, the subtree rooted at O_d has two relationship types R_2 and R_3. Since O_i is the lowest participating object class among all the participating object classes in R_2 and R_3 after the swap, the subtree rooted at O_d is attached to O_i. In this way, the relationship types in the third category in the view are kept intact and the view is still valid. In general, if a relationship type R in the third category does not involve any ancestors of O_j, that is, it only involves O_j and the descendants of O_j , then the subtree rooted at a child of O_j (say O_d) remains attached to O_j after the swap, because O_j is the lowest participating object class of R in the view. However, if R involves some ancestors of O_j, then the subtree rooted at O_d will be attached in the view to the lowest participating object class among all the relationship types in the subtree. □

4.5 Reversible Views

Rule Swap_3 and Rule Swap_4 not only maintain the semantics of the view so that it is kept valid, but they also guarantee the reversibility of the view.

Definition 4.2. **(Reversible View)** A valid view schema V of a source schema S is called a reversible view if the source schema is a valid view of V under our view operators, i.e. *select, drop, join* and *swap.*

A view is reversible if the original source schema can be restored back by applying some operators to the view. Among our view operators, it is clear that a view will not be reversible if the *select* or *drop* operator is applied. This is because some data will be lost in the view and it is impossible to recover the data back from the view. The *join* operator joins two object classes together. Based on the rules for join operator, the source data may not be lost in the view in some cases. However, we need to introduce new operators to restore the referenced object class back in order to make the view reversible. Thus, we will not consider the join operator here. Finally, *swap* operator swap two object classes in the view and the view can be reversible by applying another swap operator. Therefore, we consider swap operator only for the issue of reversible view.

On closer examination, we observe that the rules Swap_3 and Swap_4 can address the reversible view problem. Let us revisit the motivating example in Section 3.

EXAMPLE 9. *Suppose we want to design a view in Figure 4 based on the source schema in Figure 2. This view swaps the object classes course and student. Based on the rules Swap_1 and Swap_2, we first move the attributes of the two object classes with their owner object classes, and the relationship type cs's attribute grade is attached to course, that is, the new lowest participating object class of cs. It is because cs is in the first category as defined above and Rule Swap_2 applies. With*

the rule Swap_3, the object class lecturer will move down with course in the view. Thus, the relationship type cl (in the second category) involving course and lecturer is kept intact and the view is valid. Moreover, the object class tutor does not move up with student. Instead, it is attached to the lowest participating object class of cst (in the third category), i.e. course, as stated in the rule Swap_4. Thus, the semantics of the ternary relationship type cst remains unchanged and the resulting view in Figure 3 is valid.

Let us now apply a swap operation to the view in Figure 4 by swapping student and course again. Applying the rules Swap_1 and Swap_2, the attributes of student and course will move with their owner object classes. The relationship attribute grade is thus attached to the object class student again. Applying the rule Swap_4, the object class lecturer will move up with course as a whole since course is the lowest participating object class of cl. On the other hand, tutor will be attached to student because student is the lowest participating object class of cst. In this way, the semantics of the two relationship types are kept intact. Furthermore, the view obtained is the same as the original source schema in Figure 2. Thus, the view in Figure 4 is a reversible view because we can produce the original source schema back by applying swap operator on it. □

5. PARTICIPATION CONSTRAINTS

When designing a semistructured view with the operators above, new relationship types may be derived in the view from existing relationship types. Further, the view may change the order of participating object classes of an existing relationship type. For both cases, we need to recalculate the participation constraints of the relationship type in the view.

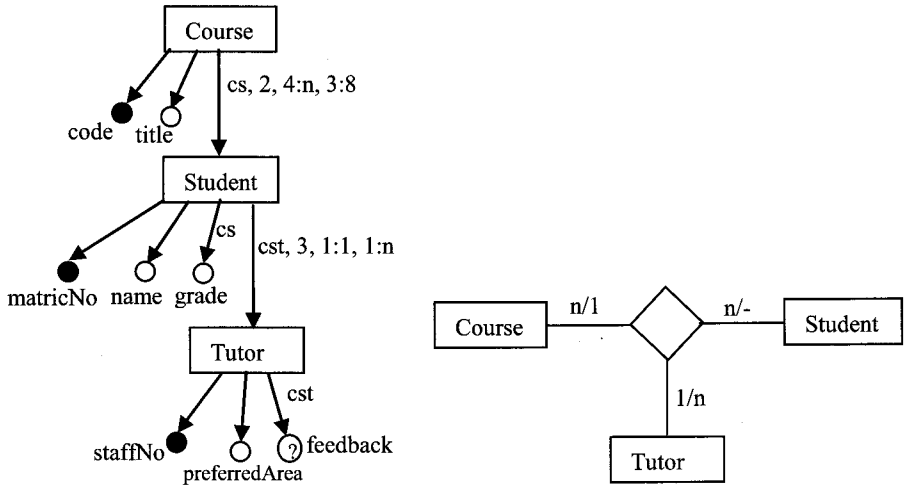
EXAMPLE 10. *Consider the source schema in Figure 28 together with its FD diagram. The following functional dependencies hold in the source schema:*

$$code, matricNo \rightarrow staffNo$$

$$staffNo \rightarrow code$$

Suppose we design a view by swapping course and tutor as shown in Figure 29. The view still keeps the two relationship types in the source schema after the rules of swap operator are applied. However, new participation constraints must be derived in the view schema for cst because the ordering of participating object classes is changed. The new parent and child participation constraints in cst become 1:n and 1:n respectively. Further, the participation constraints in cs are also changed 3:8 and 4:n.

Figure 30 shows a view where student is dropped from the source schema in Figure 28. Thus, for a given course, all distinguished tutors teaching the course are placed below as its sub-elements. In this case we derive a new relationship type ct by projecting cst. From the functional dependency diagram in Figure 28(b), a tutor can teach in only one course, but a course may have more than one tutors, so the participation constraint for course in ct in the view is 1:n, and for tutor is 1:1. Note that the attribute feedback becomes an attribute of ct with cardinality "".* □



(a) Source Schema

(b) Functional Dependency Diagram of Source Schema

Figure 28. Example source schema and its functional dependency diagram.

We develop four rules to handle the participation constraints for relationship types in semistructured views under our view operators. Rule PC_1 and Rule PC_2 handle the cases where the order of participating object classes of binary relationship types and n-ary ($n > 2$) relationship types are changed respectively. Rule PC_3 handles the case where new relationship types are derived by projecting existing relationship types. Finally, Rule PC_4 handles the case where new relationship types are derived by joining existing relationship types.

We use p and c to denote the parent and child participation constraints of an original relationship type R respectively. Likewise, we use p' and c' to denotes the parent and child participation constraints of a derived relationship type R'.

Rule PC_1: *If R' is derived in the view by swapping two participating object classes of an existing binary relationship type R in the source schema; then $p' = c$ and $c' = p$.*

When a swap operator is applied on two participating object classes of a binary relationship type, the order of the two participating object classes will then be reversed in the view schema. Thus, in the new relationship type in the view, the participation constraints will also be reversed.

Rule PC_2: *If R' is derived in the view by swapping two participating object classes in an existing n-ary ($n > 2$) relationship type R in the source schema, and O_1, O_2, ..., O_n is participating object classes of R' in the order from ancestor to descendant in the view schema; then*

(1) *For p': If there exists a functional dependency $\{O_1, O_2, ..., O_{n-1}\} \rightarrow O_n$ in the functional dependency diagram, then set p' to be 1:1, otherwise set p' to be 0:n (or *).*

(2) *For c': if there exists a functional dependency: $O_n \rightarrow \{O_1, O_2, ..., O_{n-1}\}$ in*

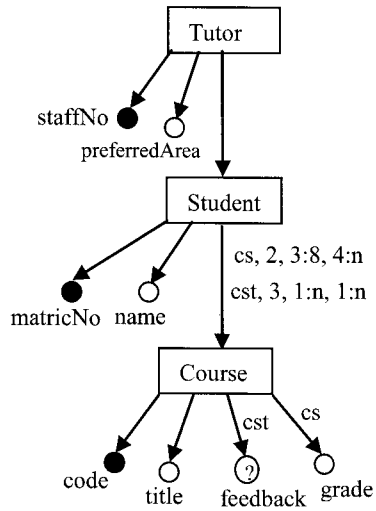


Figure 29. Changes in participation constraints due to the application of a swap operator.

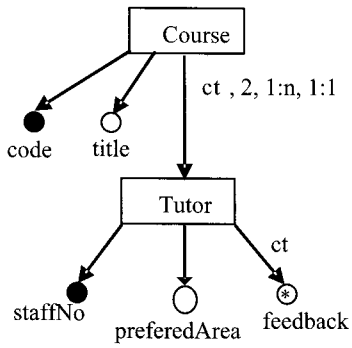


Figure 30. Changes in participation constraints due to the application of a project operator.

*the functional dependency diagram, then set c' to be 1:1, otherwise c' is set 0:n (or *).*

This rule handles the case where a swap operator is applied on an n-ary ($n > 2$) relationship type. Firstly, we use the functional dependency diagram to determine the value of p' and c'. When there is a corresponding functional dependency, we directly use it to determine p' and c'. On the other hand, there may be no functional dependencies between $O_1, O_2, ..., O_{n-1}$ and O_n in the functional dependency diagram. Without loss of generality, we assign 0:n to p' or c' in this case.

Rule PC_3: *If R' is derived in the view by projecting an existing relationship type R in the source schema, and $O_1, O_2, ..., O_n$ is participating object classes of R' in the order from ancestor to descendant in the view schema; then*

(1) *For p': If there exists a functional dependency $\{O_1, O_2, ..., O_{n-1}\} \rightarrow O_n$ in the functional dependency diagram, then set p' to be 1:1, otherwise set p' to be 0:n (or *).*
(2) *For c': if there exists a functional dependency: $O_n \rightarrow \{O_1, O_2, ..., O_{n-1}\}$ in the functional dependency diagram, then set c' to be 1:1, otherwise c' is set 0:n (or *).*

Similar to the Rule PC_2, the Rule PC_3 also utilizes the information of the functional dependency diagram to decide how to generate p' and c'. We do not provide detailed proof here for the three rules above as they are straightforward.

Rule PC_4: *If R' is derived in the view by joining one relationship type R_1 $(O_{11}, O_{12}, ..., O_{1n})$ with another relationship type R_2 $(O_{21}, O_{22}, ..., O_{2m})$, where $O_{1n} = O_{21}$ is the common object class they are joined on, then*

(1) *For p': If there exists a functional dependency*

$$\{O_{11}, O_{12}, ..., O_{1(n-1)}, O_{22}, ..., O_{2(m-1)}\} \rightarrow O_{2m}$$

or a functional dependency

$$\{O_{22}, O_{23}, ..., O_{2(m-1)}\} \rightarrow O_{2m}$$

or the two functional dependencies

$$\{O_{11}, O_{12}, ..., O_{1(n-1)}\} \rightarrow O_{1n} \quad \textit{and} \quad \{O_{21}, O_{22}, ..., O_{2(m-1)}\} \rightarrow O_{2m}$$

then set p' to be 1:1, otherwise, set p' to be 0:n.
(2) *For c': If there exists a functional dependency*

$$O_{2m} \rightarrow \{O_{11}, O_{12}, ..., O_{1(n-1)}, O_{22}, ..., O_{2(m-1)}\}$$

*in the functional dependency diagram, then set c' to be 1:1, otherwise set c' to be 0:n (or *).*

Rule PC_4 handles the participation constraints in the derived relationship types by joining existing relationship types, which is because the common object class (O1n/O21) of the two relationship types is dropped in the view.

Correctness of Rule PC_4. For p', if there exists the functional dependency:

$$\{O_{11}, O_{12}, ..., O_{1(n-1)}, O_{22}, ..., O_{2(m-1)}, O_{2m}\} \rightarrow O_{2m}$$

in the functional dependency diagram, then it is obvious that p' is set to be 1:1. If there exists the functional dependency:

$$\{O_{22}, O_{23}, ..., O_{2(m-1)}\} \rightarrow O_{2m}$$

in the functional dependency diagram, then by the augmentation property, we can deduce the functional dependency:

$$\{O_{11}, O_{12}, ..., O_{1(n-1)}, O_{22}, ..., O_{2(m-1)}, O_{2m}\} \rightarrow O_{2m}$$

Therefore p' is set to be 1:1. Finally, suppose there exist two functional dependencies:

$$\{O_{11}, O_{12}, ..., O_{1(n-1)}\} \rightarrow O_{1n} \quad \textit{and} \quad \{O_{21}, O_{22}, ..., O_{2(m-1)}\} \rightarrow O_{2m}$$

Since $O_{1n} = O_{21}$ is the common object class of the two relationship types, then by the pseudo-transitivity property, we can deduce the functional dependency:

$$\{O_{11}, O_{12}, ..., O_{1(n-1)}, O_{22}, ..., O_{2(m-1)}, O_{2m}\} \rightarrow O_{2m}$$

Therefore p' is set to be 1:1. In all the other cases, p' must be set to be 0:n. For c', there is only one case. That is, if the functional dependency

$$O_{2m} \rightarrow \{O_{11}, O_{12}, ..., O_{1(n-1)}, O_{22}, ..., O_{2(m-1)}\}$$

exists in the functional dependency diagram, then c' is set to be 1:1. Otherwise, c' is set to 0:n. □

THEOREM 5.1. *Semistructured views designed based on all the above rules do not violate the semantics, i.e. functional dependencies, relationship types, and key and foreign key constraints implied in the underlying semistructured/XML data. Further, the original source schema can be a valid view of the semistructured views under the rules. In other words, the semistructured views are valid and reversible views.*

Outline of Proof. All the rules above are clearly correct or have been proven to be correct. It is also clear that they do not violate the semantics in the source schema. Thus, the view schema designed based on the rules is valid and reversible.

6. CONCLUSION

Existing systems that support semistructured views do not maintain the semantics that are implied by the source schema during the process of designing views. Thus, they do not guarantee the validity and reversibility of the views. This work addresses these two issues. We utilize a semantically rich semistructured data model, and employ a set of view operators for designing semistructured views. The operators consist of select, drop, join and swap operators. For each type of operator, we develop a complete set of rules to maintain the semantics of the views. More specifically, we maintain the evolution and integrity of relationships once an operator is applied. We also examine the reversible view problem under our operators and develop rules to guarantee the designed views are reversible views. Finally, we examine the possible changes for participation constraints of the relationship types and propose rules to keep the participation constraints correct. To the best of our knowledge, this is the first work to employ a semantic data model for maintaining semantics of semistructured views and solving the reversible view problem. The proposed approach provides for a more robust view mechanism so that we can exploit the potential of XML/semistructured data to exchange data on the Web.

REFERENCES

ABITEBOUL S., CLUET S., MIGNET L., ET AL. 1999. Active views for electronic commerce. VLDB, 138–149

BARU C., GUPTA A., LUDAESCHER B., ET AL. 1999. XML-based information mediation with mix. ACM SIGMOD Demo, 597–599.

BOHANNON P., KORTH H., NARAYAN P., GANGULY S., AND SHENOY P. 2002. Optimizing view queries in rolex to support navigable tree results. VLDB. 119–130.

CAREY M., FLORESCU D., IVES Z., ET AL. 2000. XPERANTO: Publishing Object-Relational Data as XML. WebDB Workshop. 105–110.

CAREY M., KIERNAN J., SHANMUGASUNDARAM J., ET AL. 2000. Xperanto: A middleware for publishing object-relational data as XML documents. VLDB, 646–648.

CHEN Y. B., LING T. W., LEE M. L. 2002. Designing valid XML views. ER Conference, 463-478

CLUET S., VELTRI P., VODISLAV D. 2001. Views in a large scale XML repository. VLDB, 271–280.

DEUTSCH, A. AND TANNEN, V. 2003. Mars: A system for publishing XML from mixed and redundant storage. VLDB, 201–212.

DOBBIE G., WU X. Y., LING T. W., LEE M. L. 2000. Ora-ss: An object-relationship-attribute model for semistructured data. Tech. rep., Technical Report TR21/00, School of Computing, National University of Singapore.

FERNANDEZ M., TAN W., SUCIU D. 2001. Efficient evaluation of XML middleware queries. ACM SIGMOD, 103–114.

FERNANDEZ M., TAN W., SUCIU D. 1999. Silkroute: Trading between relations and XML. World Wide Web Conference.

HWANG D. H., KANG H. 2005. XML view materialization with deferred incremental refresh: the case of a restricted class of views. *Journal of Information Science Engineering 21,* 6, 1083–1119.

LING T. W., LEE M. L., DOBBIE G. 2005. Semistructured database design. Springer Science+Business Media, Inc.

MCHUGH J., ABITEBOUL S., GOLDMAN R., QUASS D., WIDOM J. 1997. Lore: A database management system for semistructured data. *SIGMOD Record 26,* 3, 54–66.

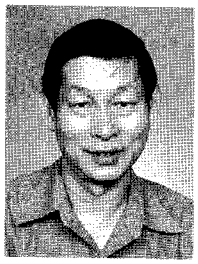
MANDHANI B., SUCIU D. 2005. Query caching and view selection for XML databases. VLDB, 469–480.

NI W., LING T. W. GLASS: A Graphical Query Language for Semi-Structured Data. DASFAA, 363–370.

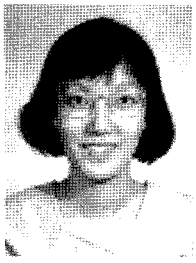
PAPAKONSTANTINOU Y., GARCIA-MOLINA H., WIDOM J. 1995. Object exchange across heterogeneous information sourcess. ICDE, 251–260.

RAJUGAN R., CHANG E., DILLON T. S., FENG L. 2005. A three-layered XML view model: A practical approach. ER Conference, 79–95.

Yabing Chen is a solution architect in HP. He obtained his PhD degree in School of Computing from National University of Singapore in 2005, his master degree in Institute of System Engineer from Tian Jin University in 1999 and his Bachelor degree in Computer Science from Tian Jin University in 1996. His PhD thesis examines semantic validity, SQL query definition generation and XQuery definition generation issues in XML/Semistructured data environment. His research interests included XML Query, XML View and XML data integration in heterogeneous data environment. His work has been published in database conferences such as database conceptual modeling conference (ER) and Database Systems for Advanced Applications (DASFAA).



Tok Wang Ling is a Professor in the School of Computing at the National University of Singapore. His research interests include Data Modeling, Entity-Relationship Approach, Normalization Theory, and Semistructured Data Model. He has published over 160 papers, co-authored a book on designing semistructured databases, and co-edited 12 conference and workshop proceedings. He organized and served as conference co-chair and PC co-chair of 7 and 5 conferences respectively, and as PC member of more than 100 conferences. He was vice chair and chair of the steering committees of ER and DASFAA conferences, and a member of the steering committee of DOOD conference. He is now a member of ER conference. He is an editor of 5 international database journals. He is a senior member of IEEE and Singapore Computer Society, and a member of ACM.



Mong Li Lee is an Associate Professor in the School of Computing at the National University of Singapore. She received her Ph.D degree in Computer Science from the National University of Singapore. Her research interests include the design and integration of heterogeneous and semistructured data, database performance issues in dynamic environments, data quality and biomedical informatics. Mong Li has published over 100 technical papers in various international conferences and journals. She has co-authored two books on designing semistructured databases and spatio-temporal data mining, as well as contributed chapters to numerous books. She has also served in the program committee and organizing committee of various international conferences such as VLDB, SIGMOD, ICDE, DASFAA and ER.

Masatake Nakanishi is a Professor at Nagoya Keizai University in Japan. He graduated from University of Tsukuba. After graduate school of education in the same university, he joined Nihon Systemix inc., where he was engaged in creating a data dictionary/directory system and serving a data-driven information systems development methodology in practical areas. He received his Ph.D. in Engineering from Nihon University. From April 2005 till March 2006 he was a visiting researcher at the School of Computing, National University of Singapore. He is a member of the Information Processing Society of Japan, the Japan Society of Management Information, and the Systems Auditors Association of Japan. His main research interests include conceptual data modeling, information systems control, XML and Xforms.

Gillian Dobbie is the Director of Software Engineering at the University of Auckland. Her main areas of interest pertain to databases and the web. She has worked in the foundations of database systems, defining logical models for various kinds of database systems, and reasoning about the correctness of algorithms in that setting. She has published 50 papers in journals and international refereed conferences, and she is currently program co-chair for the Australasian Computer Science Conference (ACSC08), local organising chair for the International Conference in Conceptual Modeling (ER07), and general co-chair of the International Conference on Very Large Databases (VLDB08).