THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Lower Kent Ridge Road, Singapore 119260

TRA9/04

A semantic approach to XML schema integration

Qi HE and Tok Wang LING

September 2004

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

JAFFAR, Joxan
Dean of School

Title: A Semantic Approach to XML Schema Integration

Authors: Qi He, Tok Wang Ling

Affiliation: School of Computing, National University of Singapore

Email: {heqi, lingtw}@comp.nus.edu.sg

Abstract. In this paper, we adopt a semantic rich model, Object-Relationship-Attribute model for Semi-Structured data (or ORASS) to represent XML schemas. A challenge in XML schema integration comes from the hierarchical structure of XML. For example, two sets of XML elements from two sources may constitute the same relationship type, but in different hierarchies. Then in the integrated schema, we need decide a "good" hierarchy of these elements. In general, we require an integrated schema preserves the semantics of source schemas, has minimum redundancy and leads to low cost data transformation. Guided by these criteria, we developed algorithms to merge equivalent elements and equivalent relationship types among elements from source schemas, and proposed a top-down approach to integrate ORASS schemas, meeting challenges caused by the hierarchical structure of XML.

Keywords: XML schema integration, ORASS.

1 Introduction

Currently, XML is becoming the de facto standard for data publishing and exchange data. Information integration for XML data becomes an issue for its business advantage and great challenges [Matt03]. XML schema integration plays an important role in building an integration system for either transaction or analytical processing purpose, but with different requirements.

A system for transaction processing (usually a mediated system, e.g., [CCS00, CDSS98, DHW01, DRR03, May01]) has a virtual integrated view (i.e., a mediated schema) which provides a unified access to heterogeneous data in sources. In this case, to integrate source schemas into the mediated schema, we have two criteria:

Criterion 1: the mediated schema should preserve the semantics (information and constraints) of source schemas;

Criterion 2: the mediated schema should be concise, i.e., minimizing the number of redundant elements.

These two criteria also apply to the schema integration in building a repository of XML schemas [Plot01]. The purpose of such a system is to provide a centralized management of XML schemas to track usage of schemas, avoid proliferating redundant schemas and so on.

On the other hand, a system for analytical processing (sometimes called a *decision support system*) requires consolidating source data into a single physical store to provide fast, highly available and integrated access to related information. Data transformation/integration is a real process instead of performed on the fly. In data transformation, data from discrepant sources are cleaned, transformed into a unified form, and then merged; redundancy is removed. Traditionally, a decision support system need integrate data with a magnitude of GB to TB. As the quantity of information available on the Internet is rocketing, data transformation/integration itself becomes a bottle neck in the system. In this case, schema integration is a preliminary step to guide the following data transformation. Besides the two

criteria mentioned above, we have the 3rd one for schema integration in a system for analytical processing:

Criterion 3: minimize the cost of data transformation.

Another interesting application of XML data integration is the cache of search engines on the Web. Currently, as the Web is HTML-based, search engines are based on information retrieval technology. When data sources are well defined and structured Web sites (financial Web sites, electronic libraries, business sites using XML based document exchange), more complex queries with some structure conditions can be supported. For example, given the title of a book, search the lowest price of the book from bookstore web sites. To speed up such queries, people may use robots crawling to cache book information. Unlike current caches which are simply repositories of HTML documents or URLs (with many redundancies and inconsistencies), the cache for an XML based search engine should have integration capability to support complex queries. Because of the huge quantity of information on the Web, the criteria of XML schema integration in this case are similar to those for analytical systems, i.e., Criterion 1, 2 and 3. Another similar application is the XML Web cache in the Web based information delivery systems which provide Internet access to the information in large legacy infrastructures [GMT02].

Most of existing work on XML schema integration is based on the grammar model of DTD [JH01, RM01]. For the inadequacies of DTDs (will be discussed in Section 1.1 below), those integration methods may lose semantics and the integrated schema may be redundant [YLL03]. In [YLL03], Yang et al. resolved structural conflicts in the integration of ORASS schemas. Instead of considering the three criteria mentioned above, they assigned each source schema a weight of importance, and tried to keep the characteristics of important source schemas in the integrated schema. We will compare their wok with ours in Section 6 below.

In this paper, we will introduce a new approach to XML schema integration, guided by the three criteria and focusing on the challenges caused by the hierarchical structure of XML. Our work is useful in information integration systems based on XML data. In the rest of this section, we first review the ORASS model in brief, and then analyze some challenges in XML schema integration. Finally, we present some assumptions and the organization of this paper.

1.1 Semantic Data Model — ORASS

Unlike most work on DTD, we adopt a semantic-rich model, ORASS [WLL01], to represent XML schemas. ORASS has four kinds of schema constructs: *object class* (correspond to element in DTD), *relationship type* (describe associations among object classes), *attribute of object class* and *attribute of relationship type.* In ORASS, an XML schema is represented as a tree (or graph) with object classes as rectangles and attributes as circles (filled circles denote *keys* of the owning object classes). A relationship type among object classes is specified on the last edge in the path linking those object classes.

Example 1.1: the schema of Figure 1.1 represents a ternary relationship type SPM among object classes SUPPLIER, PROD and MONTH. The number beside a relationship type name indicates the *degree*, the number of object classes in the relationship type. Attributes under an object class may belong to the object class or a relationship type, e.g., attribute MONTH is an attribute (the key) of the object class with the same name, while PRICE is an attribute of the

relationship type SPM. □

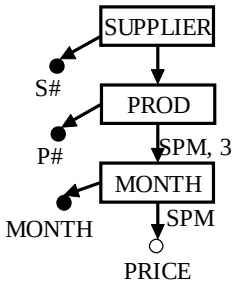


Figure 1.1: An ORASS schema

Participation constraints of object classes in a relationship type and *quantifiers* (?, +, *) of attributes can be specified in ORASS also, which are omitted here for convenience.

Comparing with ORASS, DTD or XML Schema [1] does not provide much semantics for effective schema integration, i.e.,

- DTD or XML Schema can only express binary relationships between elements and sub-elements.
- DTD or XML Schema does not distinguish attributes of object classes or attributes of relationship types. Therefore, it is difficult to preserve semantics in schema transformation (e.g., when swapping an element and a sub-element).
- In DTD or XML Schema, the type of an element is defined by the element name and the types of sub-elements. The nesting definition makes it costly to identify equivalent elements which should have the same name and sub-elements.

1.2 Challenges in the Integration of ORASS Schemas

In the integration of ORASS schemas, challenges come from three factors: hierarchical structure, semantic heterogeneities and semi-structuredness. In this paper, we will only focus on the resolution of the first challenge.

- *Hierarchical structure of XML.* Equivalent object classes from different sources may constitute equivalent relationship types, but in different hierarchies. We need decide a unified hierarchy of these object classes in the integrated schema (Section 2). More general, several relationship types may have some common object classes constituting equivalent sub relationship types which should be merged (Section 3 and 4).
- *Semantic heterogeneities.* As the integration of relational or entity-relationship schemas, in XML schema integration, a bundle of conflicts among source schemas need to be solved, e.g., naming conflicts (homonyms and synonyms), structural conflicts (i.e., the same concept is modeled using different schema constructs in source schemas) [YLL03], schematic discrepancies (i.e., data values in one source correspond to schema labels of another) [HL04], and constraint conflicts.
- *Semi-structuredness of XML.* An integrated schema would be semi-structured because source schemas would be not only semi-structured by themselves, but also heterogeneous from each other. In particular, in source schemas, equivalent elements may have either different attributes and sub-elements, or the same attributes and sub-elements but with

[1] Without causing confusion, we capitalize the first letter of "Schema" to denote the W3C recommendation of XML Schema.

different participation constraints.

1.3 Assumptions and Paper Organization

In our proposal, we assume source schemas are in ORASS model (some semi-automatic tools can help users define an ORASS schema from DTD), and schema matching (i.e., the equivalent relations) between object classes, relationship types and attributes in source schemas are already known through some machine learning techniques or/and human specifications [RB01]. To simplify the presentation, we assume semantic heterogeneities among source schemas, e.g., naming conflicts, structural conflicts etc, have been resolved if any. We also ignore the conflicts of participation constraints among ORASS schemas.

We focus on XML data in hierarchical structure instead of "flat" XML data (or record-oriented XML). Flat XML data usually come from relational databases, keeping tuple-like structure of relational data. The integration of relational data has been studied for decades [BLN86], but the hierarchical structure of XML brings some new challenges which will be studied in this paper. ID reference does not bring much trouble to our solution. For convenience, we will treat ORASS schemas as trees.

We assume each object class has a *key* attribute (but not an "object id" assigned by systems), e.g., isbn numbers of books. In data integration, two objects with the same value on the key attribute can be merged. Further, we assume in source XML data, the objects of an object class have already been sorted by the key values. If this assumption does not hold, our method also applies, but some optimization strategies (to reduce the cost of data transformation) may become meaningless.

In the rest of the paper, we first study how to integrate relationship types, i.e., paths in schema trees in Section 2, 3 and 4, then schemas which are regarded as collections of relationship types in Section 5. Section 6 compares our work to others. Section 7 concludes the whole paper.

2 Deciding a Hierarchy of Object Classes in Equivalent Relationship Types

In ORASS schemas, a relationship type is represented as a path, i.e., a hierarchy of object classes. It is possible that two relationship types are ***equivalent*** to each other (i.e., they model the same association involving the same set of object classes) [2], but have different hierarchies of object classes. To integrate the equivalent relationship types, we should decide a unified hierarchy of the object classes in the relationship type of the integrated schema (or *integrated relationship type* for short). For an integrated schema requiring Criterion 3 (see Section 1), a "good" hierarchy should minimize the (I/O) cost to transform/integrate data from source schemas to the integrated schema. On the other hand, if Criterion 3 is not required, e.g., in a mediated system for transaction processing, we can randomly select a hierarchy for the integrated relationship type, as it would not make difference to the performance of query processing.

In this and the following two sections, we will consider all the three criteria in schema integration. In Section 2.1, we first propose a method to reorder object classes in a relationship type (i.e., change the hierarchical order of the object classes in a relationship type). The

[2] Note the instance sets of two equivalent relationship types may not be the same, so do equivalent object classes.

reordering preserves the semantics of relationship types, attributes of object classes and attributes of relationship types in ORASS schemas (i.e., satisfying Criterion 1). Criterion 2 is trivial as all the equivalent object classes in equivalent relationship types will be merged. Criterion 3 is the trickiest one to be satisfied and the focus of this section. In Section 2.2, we provide a cost model to estimate the cost of reordering which is used to decide a good hierarchy for the integrated relationship type. We study the search space of our decision process in Section 2.3.

As mentioned, we assume in source XML data, the objects of an object class have been sorted by the key values. Otherwise, adopting variant hierarchies of object classes in an integrated relationship type makes little difference to the cost of data transformation using our method. To simplify the presentation, we use key values to represent objects in relationships, and omit attributes of object classes and attributes of relationship types.

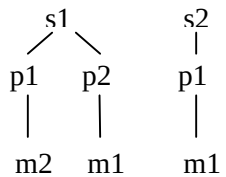
2.1 Storage Model and Reordering Object Classes in a Relationship Type

XML data of source schemas may be in different models, e.g., *Element-Based Clustering* model (or EBC) in which element nodes with the same tag name are clustered and organized as a list [MLLA03], *Object Exchange Model* (or OEM) [MAG+97], relational database (or object-relational database) [GMT02], or native XML documents. No matter how XML data is stored in sources, in order to efficiently reorder object classes in a relationship type, we will first get and store the relationships in a flat table with the fields corresponding to the key attributes of the object classes involved in the relationship type. Then we compute the relationships with a needed hierarchy of objects from the flat table.

Example 2.1: Suppose in Figure 1.1, we want to reorder the hierarchy S/P/M into P/S/M (S, P, M are shorthands for SUPPLIER, PROD and MONTH). We first scan and store the original relationships in a flat table of Figure 2.1 (a). In the table, the values of S, P, M are the key values of the corresponding objects. Note the relationships in the table are already sorted by S, P, M (This could be implemented by a sorting algorithm with a comparison function that, given two relationships, compares the S values, and if there is a tie, compares the P values, and if another tie occurs, compares the M values). Figure 2.1 (b) is the corresponding trees of these relationships.

S	P	M
s1	p1	m2
s1	p2	m1
s2	p1	m1
...	...	...

(a) flat table



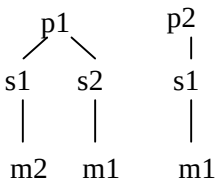
(b) tree structure

Figure 2.1: Relationships in a flat table and tree structure

Now we change the relationship type S/P/M into P/S/M, i.e., swapping SUPPLIER and PROD in the relationship type of Figure 1.1. To accomplish this, we just sort the table by P, S, M. The result is Figure 2.2 (a). To construct the tree structure of the relationships, we scan the table, and merge the same objects of P, S and M in order, and get Figure 2.2 (b). □

S	P	M
s1	p1	m2
s2	p1	m1
s1	p2	m1
...	...	...

(a) flat table



(b) tree structure

Figure 2.2: Reorder S/P/M into P/S/M: first sort the table by P, S, M, then merge the common objects

Another optional implementation of reordering is using XQuery, i.e., defining the reordered relationship type as a query against the original one. For example, to implement the reordering of Example 2.1, people should write a query with three nested "for" loops on P, S and M (Figure 2.3). Intuitively, our method would outperform the XQuery implementation thanks to the low cost of sorting (see Section 2.2 for details).

```
for $P# in distinct /SUPPLIER/PROD/@P#
return <PROD P# = "{$P#}">
        {
          for $S in /SUPPLIER[PROD/@P# = $P#]
          return <SUPPLIER S# = "{$S/@S#}"
                  {
                    for $M in $S/PROD[@P# = $P#]/MONTH
                    return $M
                  }
                  </SUPPLIER>
        }
        </PROD>
```

Figure 2.3: XQuery statement to swap the object classes SUPPLIER and PROD in the relationship type of Figure 1.1

2.2 Cost Model

From Example 2.1, we know the reordering of object classes in a relationship type needs re-sort the table storing the relationships by the object classes in a required order. Besides the cost of sorting, the integration of a set of equivalent relationship types involves some other costs, e.g., reading the relationships in sources into our flat tables, and merging the relationships of several sorted tables to get the integrated relationship type. The costs of reading and merging are constant given the numbers of original relationships in sources. But the sorting cost may be variable if we adopt different hierarchies of object classes in the integrated relationship type. So we only need compare the sorting costs in deciding a good hierarchy by Criterion 3.

If we adopt the "external merge sort", given a table with n tuples (relationships) or N pages, let B>3 be the buffer pages, the I/O cost of sorting is

$$2*N*\lceil 1+\log_{B-1}\lceil N/B \rceil \rceil \qquad (1)$$

However, the cost may be even reduced if the original relationships are already sorted by the key values of some preceding object classes (this always holds by our assumption), and these

object classes are not involved in the reordering.

In Example 2.1, suppose we want to reorder S/P/M into S/M/P, i.e., the reordering does not involve the root node SUPLLIER. Then in sorting the table of Figure 2.1 (a), we only need sort the tuples for each value of S locally, instead of sorting the whole table globally.

In general, given a relationship type $O_1/O_2/\ldots/O_m$ with n relationships (or N pages for the table storing the relationships) sorted by the key attributes of O_1, O_2, …, O_m, let k_i, i=1, …, m, be the number of the possible distinct values of the key attribute of O_i in the relationship type, and suppose the key values are uniformly distributed. Now we reorder $O_1/O_2/\ldots/O_m$ into $O_1'/O_2'/\ldots/O_m'$ such that $O_1=O_1'$, …, $O_i=O_i'$ but $O_{i+1} \neq O_{i+1}'$ for some $0 < i < m-1$. In this case, we only need sort the relationships for each sequence of the key values of O_1, …, O_i instead of all the relationships in the table. Then the I/O cost of sorting the relationships by O_1', …, O_m' is

$$2*N*\lceil 1+\log_{B-1}\lceil N/(B*k_1*\ldots*k_i)\rceil\rceil \qquad (2)$$

Comparing with formula (1), the I/O cost saved is:

$$2*N*\lceil \log_{B-1}(k_1*\ldots*k_i)\rceil \qquad (3)$$

There're two special cases not included in formula (2) and (3): (Case I) when i=0, i.e., the reordering change the root node of a relationship type, and the sorting have to be performed on all the relationships, then formula (2) is degenerated into (1), and the saved cost of (3) is 0; (Case II) when i=m, i.e., the relationship type keeps the original hierarchy, then formula (2) has a value of 0, and formula (3) becomes (1).

Now the problem can be defined as follows: given a set of equivalent relationship types $\mathcal{R}$ from different sources, find a relationship type R which is equivalent to those of $\mathcal{R}$ and minimizes the total cost of (2) (or maximize the total saved cost of (3)) to reorder the object classes of the relationship types in $\mathcal{R}$ to the hierarchical structure of R.

2.3 Enumerating the Hierarchies of the Object Classes in Equivalent Relationship Types

Given a set of equivalent relationship types $\mathcal{R}$ each element of which involves m object classes, is it necessary to enumerate all the m! permutations of the m object classes in deciding their hierarchy in the integrated relationship type? The answer is negative in most cases. We only need consider all the distinct hierarchies of the object classes in the relationship types of $\mathcal{R}$, as shown in the following proposition:

Proposition 2.1: Given a set of equivalent relationship types $\mathcal{R}$, let R be the integrated relationship type of $\mathcal{R}$ minimizing the reordering cost (formula (2) of Section 2.2), then the hierarchy of the object classes in R is the same as some relationship type in $\mathcal{R}$. □

The proof is easy and omitted. Before ending this section, we give an example of deciding the hierarchy of the object classes in equivalent relationship types.

Example 2.2: Given two equivalent relationship types from two sources: (R1) S/P/M and (R2) S/M/P, suppose for R1 (R2), the page number of the table storing the relationships is 1000 (3200), and the distinct values of the key attributes of S, P and M are 50 (10), 40 (60) and 12 (12) respectively. Let the number of buffer pages B = 5. Applying formula (2) of Section 2.2, we can compute the reordering cost if we adopt R1 or R2 as the integrated relationship type:

R1 as the integrated relationship type: $0 + 2*3200*\lceil 1+\log_{5-1}\lceil 3200/(5*10)\rceil\rceil = 25600$;

R2 as the integrated relationship type: $2*1000*\lceil 1+\log_{5-1}\lceil 1000/(5*50)\rceil\rceil + 0 = 4000$.

Consequently, the integrated relationship type should have the same hierarchical structure as R2.□

3 Merging Relationship Types

In general, several relationship types from source schemas may have some common object classes constituting of equivalent sub relationship types. In this and the next section, we propose, respectively, a basic and an improved algorithm to merge those common object classes in relationship types.

For example, given two relationship types (R1) /S/P/M and (R2) /S/P among SUPPLIER, PROD and MONTH, suppose the sub relationship type /S/P of R1 models the same association as R2 (e.g., the supply association between suppliers and products). Then we should merge R1 and R2 into /S/P/M in which some objects of P may not have any objects of M as their children.

In Section 3.1, we first give an example of merging relationship types, and define some concepts for further use. Then in Section 3.2, we propose a greedy approach to merge relationship types. For easy of presentation, we assume the equivalent object classes in two relationship types always constitute equivalent sub relationship types (otherwise, the object classes cannot be merged).

3.1 Definition

In what follows, we first give an example of merging relationship types, then define some concepts for further use.

Example 3.1: Suppose we want to merge three relationship types A/B/C/D, A/B/D and B/A/C. One of the merging strategies is presented in Figure 3.1. That is, we first merge the 1st and 2nd relationship types on A, B and D (C and D should be swapped in the 1st relationship type). Then we merge the intermediate result with the 3rd relationship type on A and B (B and A should be swapped in the 3rd relationship type), and get the final integrated schema.□

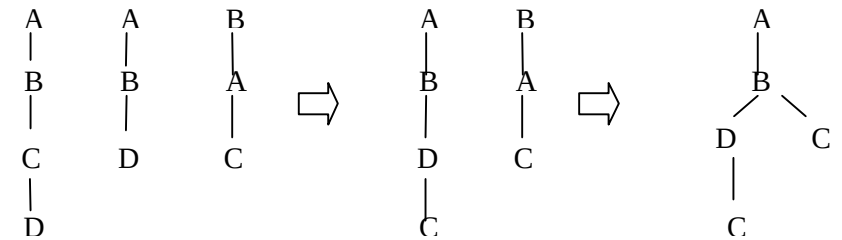


Figure 3.1: First merge A/B/C/D and A/B/D, then B/A/C

In an integrated schema, we call a path a ***separate path*** if all the object classes within the path are from the same set of source relationship types. A separate path is ***non-trivial*** if it contains more than one object class.

In the integrated schema of Figure 3.1 (c), there are 4 separate paths: A/B merged from all

the three source relationship types, D merged from the 1st and 2nd relationship types, the left C from the 1st relationship type, and the right C from the 3rd relationship type. The swap of the object classes in a separate path will not break relationship types. On the other hand, the swap of the object classes from two separate paths will break relationship types, and therefore is not allowed (e.g., in Figure 3.1 (c), the swap of B and D breaks A/B/C).

We call a set of (at least two) ORASS schemas $\mathcal{T}$ are ***(k-)merge-able*** on a set of object classes {O1, O2, …, Ok}, provided each tree of $\mathcal{T}$ has O1, O2, …, Ok in the separate path from the root node. And O1, O2, …, Ok are the ***merge-able object classes*** among the trees of $\mathcal{T}$. Given a set of schemas $\mathcal{T}$ and a set of object classes OBJ, we denote $\mathcal{T}_{\mathbf{OBJ}}$ to be the maximum set of trees merge-able on OBJ in $\mathcal{T}$. Given a set of relationship types $\mathcal{R}$ and an integrated schema (or a set of schemas) S, we denote the ***reordering cost of S*** as the total cost needed to transform the relationship types of $\mathcal{R}$ into the corresponding ones in S.

3.2 Algorithm

Recall the three criteria introduced at the beginning of Section 1, i.e., semantics preserving, conciseness and minimum cost of data transformation. For Criterion 1, the reordering of object classes in relationship types preserves the semantics of relationship types, attributes of object classes and attributes of relationship types in ORASS schemas. If all the relationship types are equivalent to each other (as the case discussed in Section 2), Criterion 2 becomes trivial, and only Criterion 3 needs to be considered. In a general case, all the 3 criteria should be considered. Unfortunately, Criterion 2 and 3 may not be achieved at the same time, and Criterion 2 has higher priority than Criterion 3 usually.

In general, given a set of relationship types $\mathcal{R}$, we do the merging in 3 steps. Step 1 is for Criterion 2, the conciseness of an integrated schema. In this step, we try to merge equivalent object classes of equivalent (sub) relationship types as many as possible. It produces a set of intermediate integrated schemas which will be further multiplied and justified in Step 2 and 3 by Criterion 3 of cost consideration. We briefly describe the 3 steps as follows:

Step 1: In this step, we merge the relationship types of $\mathcal{R}$ in a loop. In each iteration of the loop, let $\mathcal{T}$ be the set of trees obtained from the last iteration (initially $\mathcal{T}$ is equal to $\mathcal{R}$), and k be the maximum number of merge-able object classes between any two trees of $\mathcal{T}$. Let OBJ be a set of merge-able object classes with the cardinality k. Let $\mathcal{T}_{\text{OBJ}}$ be the maximum set of trees merge-able on OBJ in $\mathcal{T}$.

In the trees of $\mathcal{T}_{\text{OBJ}}$, we push the object classes of OBJ to the tops of these trees, and permute these object classes in the same hierarchy which is decided in random. Then we merge these trees on OBJ. Note the set OBJ is not unique, and different choices of OBJ lead to different intermediate integrated schemas which will be further multiplied and justified in Step 2 and 3.

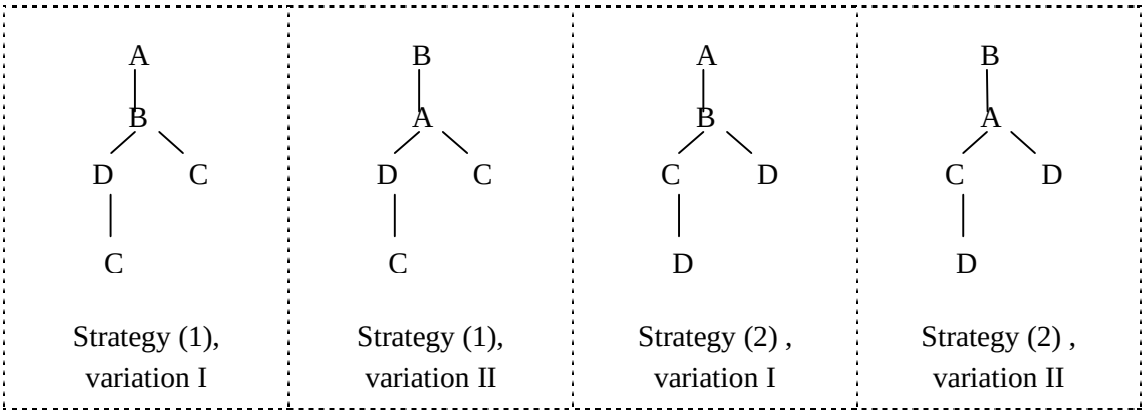
The loop will not terminate until no more trees can be merged in $\mathcal{T}$.

Step 2: For each intermediate integrated schema S obtained in the last step, we produce a set of variation schemas with different hierarchies of the object classes in the non-trivial separate paths of S.

Step 3: For each schema S' produced from Step 2, we compute the reordering cost needed to transform the original relationship types of $\mathcal{R}$ into the corresponding ones of S'. The schema with the minimum reordering cost is selected as the final integrated schema. □

This is a greedy algorithm, as in each iteration of Step 1, we choose a set of trees with the maximum number of merge-able object classes to be merged. It cannot be ensured that the final integrated schema has the minimum number of object classes (Criterion 2).[3] The formal algorithm is omitted. We hereby explain the 3 steps by merging the three relationship types of Example 3.1.

Example 3.2: Given the three relationship types A/B/C/D, A/B/D and B/A/C, we have two strategies to merge them (corresponding to the different choices of the set of merge-able object classes): (1) first merge the 1st relationship type with the 2nd one on {A, B, D}, then with the 3rd one on {A, B}, or (2) first merge the 1st one with the 3rd one on {A, B, C}, then with the 2nd one on {A, B}. Considering the permutations of the object classes (A and B) in the non-trivial separate path, we get four candidate integrated schemas after Step 1 and 2:



Then, in Step 3, we compute the reordering cost for each of the 4 schemas. For the 1st schema, the cost comprises the reordering cost to transform A/B/C/D into A/B/D/C and the cost to transform B/A/C into A/B/C. Note the relationship type A/B/D preserves its hierarchy in the integrated schema, and has no reordering cost. Similarly, we compute the reordering costs of the other 3 schemas, and choose the one with the minimum cost as the final integrated schema. □

4 An Improved Algorithm to Merge Relationship Types

In the algorithm of Section 3.2, the search space may become big because of two reasons: (a) in Step 1, different choices of the merge-able object classes OBJ, and (b) in Step 2, different hierarchies of object classes in non-trivial separate paths. In this section, we design an algorithm to reduce the search space caused by the second reason. Roughly, for an intermediate integrated schema S produced by Step 1, our algorithm decides a hierarchy for each non-trivial separate path in S, and get a schema S' minimizing the reordering costs of all the variations of S (the variation schemas are those with different hierarchies of the object classes in the non-trivial separate paths of S). We call S' a ***minimum cost tree*** *of S.* After that, we compare the reordering costs of all the minimum cost trees of the schemas produced by Step 1, and select the tree with the minimum cost as the final integrated schema. As the improved algorithm takes intermediate integrated schemas as input, it is independent of the

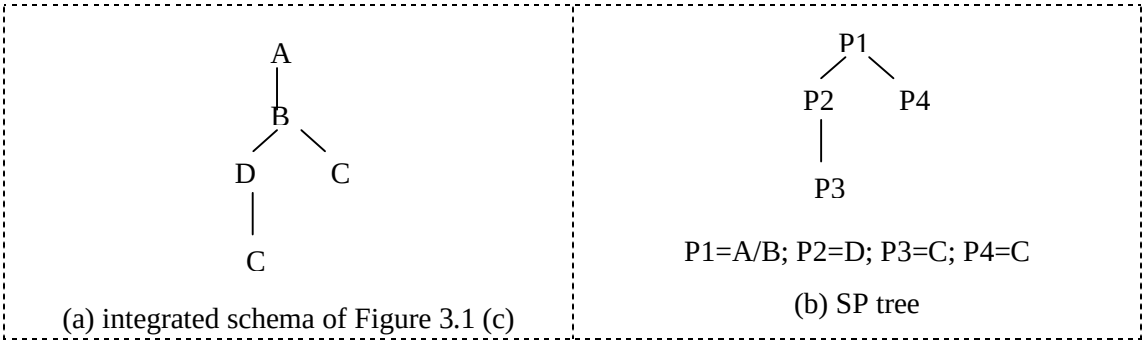
[3] It seems an optimal algorithm would have too large search space to be applicable. And it is even harder to minimize the redundancy of a whole integrated schema.

merging method of Step 1. That is, if we adopt another method (instead of the greedy method of Step 1 in Section 3.2) to merge original relationship types, the algorithm also works. In what follows, we first define some useful concepts, then present the algorithm.

4.1 Definition

Given an intermediate integrated schema S produced by Step 1 (see Section 3.2), we introduce a ***separate path tree*** (or ***SP tree***) as a tree obtained by using a node to represent each separate path in S.

For example, we show the integrated schema of Figure 3.1 (c) and its SP tree below.



(a) integrated schema of Figure 3.1 (c)

(b) SP tree

Figure 4.1: An intermediate integrated schema (of A/B/C/D, A/B/D and B/A/C) and the corresponding SP tree

A SP tree provides an overview of an integrated schema, treating separate paths as units to be handled in our algorithm. Each path in a SP tree corresponds to a path in the integrated schema. For example in the above figure, the path P1/P4 of the SP tree corresponds to the path A/B/C in the integrated schema.

We call a ***permutation*** of a node P in a SP tree as a hierarchy of the object classes in the separate path of the integrated schema. We call a permutation of P is ***beneficial***, if the path from the root to P in the SP tree corresponds to a path from the root in a source relationship type.

For example, in Figure 4.1 (b), P1 = A/B is a beneficial permutation corresponding to the paths from the roots in the relationship types A/B/C/D and A/B/D. Similarly, P2 = D is a beneficial permutation, as P1/P2 = A/B/D corresponds to the source relationship type.

An observation is that adopting a beneficial permutation of a separate path P (if any) in an integrated schema saves the reordering cost to transform some source relationship types to the integrated schema. On the other hand, if P does not have any beneficial permutation (e.g., P3 and P4 in Figure 4.1 (b) have no beneficial permutation), then the permutations of P and the permutations of the descendants of P in the SP tree make no difference to the reordering cost, and therefore can be decided in random.

4.2 Algorithm

Given a set of source relationship types $\mathcal{R}$, an intermediate integrated schema S of $\mathcal{R}$ obtained from Step 1 (see Section 3.2), and the corresponding SP tree SPT0 rooted at a node P, our algorithm MCT(P, c, SPT) below computes a SP tree SPT corresponding to the minimum cost tree of S with the minimum reordering cost c.

The algorithm is recursive. In each recursion MCT(P', c', SPT') for P' a non-leaf node in SPT0, we divide two cases:

Case 1: P' has beneficial permutations. For each beneficial permutation, we recursively call MCT(P'.child[i], ci, SPTi) (i=1, …, n for n the number of the children nodes of P' in SPT0) to compute the SP trees SPTi's rooted at the children of P' in SPT0, such that each SPTi plus the path from the root of SPT0 to the root of SPTi (i.e., P'.child[i]) corresponds to a minimum cost tree. Then from all the beneficial permutations of P', we select the one minimizing c'=c1+…+cn, and construct SPT' as a SP tree that has the root P' and n sub-trees SPT1, …, SPTn under P'.

Case 2: P' has no beneficial permutation. In this case, we randomly decide permutations for P' and the descendants of P' in SPT0, as the permutations have nothing to do with the reordering cost.

The complete algorithm is given below.

Algorithm MCT(P, c, SPT)

Input: A set of source relationship types $\mathcal{R}$, an intermediate integrated schema S of $\mathcal{R}$ obtained from Step 1 of the algorithm in Section 3.2, and the corresponding SP tree SPT0 rooted at P.
Output: A SP tree SPT corresponding to the minimum cost tree of S with the reordering cost c.

Case 1: P has beneficial permutations.
  Case 1.1: P is an intermediate node in SPT0.
    For each beneficial permutation of P, **do**
      For each child node P.child[i] of P, i=1, …, n **do**
        MCT(P.child[i], ci, SPTi).
    Let optimPerm be a permutation of P minimizing c = c1+c2+…+cn, and let SPTi, i=1,…,n be the returned SP trees for optimPerm.
    Order the object classes of P according to optimPerm.
    Construct the SP tree SPT that has the root P and n sub-trees SPT1, …, SPTn under P.
  Case 1.2: P is a leaf node.
    For each beneficial permutation of P, **do**
      Let P_S be the path in S corresponding to the path from the root to P in SPT0.
      Compute the reordering cost c1 of P_S.
    Let optimPerm be a permutation of P minimizing c = c1.
    Order the object classes of P according to optimPerm.
    Construct the SP tree SPT that has the only node P.
Case 2: P has no beneficial permutation.
  Order the object classes of P and the object classes of the descendants of P at random.
  For each leaf node LPi (i=1, …, n) which is a descendant of P, **do**
    Let P_S be the path in S corresponding to the path from the root to LPi in SPT0.
    Compute the reordering cost ci of P_S.
  c = c1+…+cn.
  Construct the SP tree SPT as the same tree rooted at P in SPT0. □

We give an example below to explain the algorithm.

Example 4.1: In Example 3.2, given the three source relationship types A/B/C/D, A/B/D and B/A/C, we have two strategies to merge them in Step 1. For Strategy (1), i.e., first merging A/B/C/D and A/B/D, then B/A/C, we suppose the intermediate integrated schema (say S) and

the SP tree (say SPT0) are shown in Figure 4.1. Calling the algorithm MCT(P1, c1, SPT1), we can compute a SP tree SPT1 corresponding to the minimum cost tree of S. The execution of MCT(P1, c1, SPT1) is explained below:

P1 has two beneficial permutations A/B and B/A. For the permutation A/B, we recursively call MCT(P2, c2, SPT2) and MCT(P4, c4, SPT4) (Line 5). In executing MCT(P2, c2, SPT2), we find D is a beneficial permutation of P2, as P1/P2 corresponds to the source relationship type A/B/D. Then we recursively call MCT(P3, c3, SPT3). P3 has an only permutation, i.e., the object class C which is not beneficial (Line 16). Now we have fixed the permutations of P1, P2 and P3 in the SP tree, corresponding to a path A/B/D/C in S. We compute the reordering cost c3 to transform the source relationship types A/B/C/D and A/B/D to the integrated relationship types A/B/D/C and A/B/D, and construct SPT3 = P3 = C (Line 18~22). Then after MCT(P3, c3, SPT3) returns, we compute c2 = c3, and SPT2 = P2/P3 = D/C in MCT(P2, C2, SPT2). Similarly, we can compute c4 and SPT4 in MCT(P4, c4, SPT4).

Then in MCT(P1, c1, SPT1), we compute c1 = c2+c4 and SPT1 the SP tree of Figure 4.1 (b). Similarly, for the other permutation B/A of P1, we can compute another c1 and SPT1, say c1' and SPT1'. Then we compare c1 and c1', and choose the SP tree with the minimum cost as the minimum cost tree of the integrated schema of Strategy (1). Similarly, we compute the minimum cost tree of the integrated schema of Strategy (2) (see Example 3.2 for the 2 strategies). From the two minimum cost trees, we select one with the less reordering cost as the final integrated schema. □

The proof is omitted for lack of space. The time and space complexities of the algorithm are, respectively, $O(n^2)$ and $O(n)$ for n the number of original relationship types (note a relationship type usually will not involve many object classes the number of which is treated as a constant in the analysis). Intuitively, the algorithm MCT(P, c, SPT) is efficient because:

(1) If P is a non-leaf node and has beneficial permutations, then the recursive calls to the children of P are independent of each other. That is, given a SP tree rooted at a child of P, we can decide the permutations of its nodes locally.

(2) If P has no beneficial permutation, then we can randomly decide the permutations for P and the descendants of P in the SP tree.

5 Integration of ORASS Schemas

In the integration of a set of ORASS schemas, besides the three criteria introduced at the beginning of Section 1, people usually require an integrated schema to be in the same model as source schemas. In our case, it is required that:

Criterion 4: An integrated schema should be an ORASS schema also.

This criterion is not trivial. Some work may integrate a set of XML schemas in DTD or some semantic model into one not conforming to the original model. We will see such examples in Section 6 below. Unfortunately, it is hard to meet all the 4 criteria (some may even contradict each other); tradeoffs are therefore needed. For example, for the conciseness and understandability of an integrated schema, some constraints may be loosed. On the other hand, for the semantic preserving issue, some redundancy may be allowed in the integrated schema.

Before giving the integration algorithm, we first analyze some characteristics of ORASS schema, and propose corresponding strategies in schema integration.

First, two relationship types may join on some object classes. For example, suppose a path (P) DEPT/EMP/PROJ (abbreviated D/E/J), represents two binary relationship types: D/E between departments and employees, and E/J between employees and projects. The hierarchical structure of the path indicates an *inclusion dependency*: an employee in a relationship of E/J must participate in D/E. To preserve such constraints, we will adopt a top-down integration approach. That is, to integrate P with other schemas, we first merge D/E with other relationship types at the same level, then E/J.

Second, several relationship types may share some preceding object classes, i.e., branches in the tree structure of an ORASS schema. In this case, we "split" and handle those relationship types respectively.

In general, given a set of source ORASS schemas, we integrate them into ORASS schemas (the result may be a forest instead of a tree) in two steps:

Step 1: Integrate the relationship types rooted at the root nodes.

Step 1.1: Merge the relationship types rooted at the root nodes using the algorithms proposed in Section 3 and 4.

Step 1.2: Repeat Step 1.1 till the reordering of object classes in the relationship types does not generate new roots.

Step 2: For each schema obtained from Step 1, we perform a depth first search on it; for each object class O (except the root) scanned, do

Merge the relationship types rooted at O using the algorithms of Section 3 and 4. □

Note in this algorithm, the operations on root (Step 1) and intermediate object classes (Step 2) are a little different. That is, for an intermediate object class O:

- The reordering of object classes in relationship types would not change the beginning position of O, for O is the common root of all the relationship types considered in that step.
- The merging is restricted in the sub-tree rooted at O instead of a set of trees.

6 Related Work

[YLL03] resolved structural conflicts in the integration of ORASS schemas. In that work, they assigned each source schema a weight of importance, and tried to keep the characteristics of source schemas with larger weights in the integrated schema. However, our purposes are to preserve semantics, minimize the redundancy of integrated schemas, and reduce the cost of data transformation. However, that work has some problems in comparison with ours:

(1) Some relationship types may be lost in the integration. For example, given two paths /S/P/M and /M/P/S representing the same ternary relationship type among SUPPLIER, PROD and MONTH, their algorithm would integrate them into a schema consisting of two binary relationship types /P/S and /P/M, loosing the original ternary one. Using our approach, we reorder the object classes in one relationship type and merge it with the other one, preserving the ternary relationship type in the integrated schema.

(2) It may introduce some error constraints. For example, given two relationship types /M/S/P and /S/P among MONTH, SUPPLIER and PROD, their algorithm would integrate them into /M/S/P which means each relationship of /S/P must occur in a relationship of /M/S/P. But this may not be the case for the source data. A remedy is to

allow the existence of /S/P relationships without a preceding M object in the integrated schema. Special measures are then needed to handle the "null" values on M. Using our approach, we will reorder /M/S/P into /S/P/M and merge it with /S/P (note in the integrated relationship type /S/P/M, some products may not have months as their children).

(3) As the integration is not done in a top-down way, some merging would be unnecessary. For example, given two paths (relationship types) /DEPT/EMP and /PROJ/EMP, their algorithm would merge the two EMP object classes (as they are equivalent), and split them again later on (as they have different parent nodes). Using our approach, we will not merge the two EMP object classes at all.

Unlike our semantic approach, some work on XML schema integration is based on the grammar model of DTD. For example, [JH01] applied a tree grammar inference technique to generate a set of tree grammar rules from source DTDs. For the inadequacies of DTDs (Section 1.1), their approach may lose semantics and the integrated schema may be redundant [YLL03]. [RM01] devised a semantic approach to integrate a set of DTDs into a common conceptual schema. Their work has some deficiencies compared with ours: (1) they handled only binary relationship types between XML elements; (2) they did not integrate equivalent relationship types in different hierarchical orders; (3) the integrated schema may have some unnecessary circles for redundant relationship types.

In some XML data integration systems, e.g., Xyleme [DRR03], Nimble [DHW01], LoPiX [May01] and YAT [CCS00, CDSS98], the developers either provide an XML query language for users to write mediated schemas by hand, or assume a mediated schema and the mapping from source schemas to the mediated schema have been given already. They focused on query processing through a mediated schema instead of schema integration. Our work can be regarded as a preceding component of theirs, which generates a mediated schema and the mappings (semi-)automatically.

Some work on schema matching [RB01] focused on the detection of equivalent elements and structures of source schemas using machine learning or IR techniques. Schema integration typically follows schema matching to reduce human participation. It seems existing work on schema matching was inadequate to detect equivalent relationship types with different hierarchical structures, although it is common in XML data.

7 Conclusion and Future Work

In this paper, we designed a semantic approach to integrate XML schemas. We adopted ORASS model to represent XML schemas, which provides rich semantics for effective schema integration. We identified four criteria of schema integration in XML data integration systems for different purposes, i.e., semantic preserving, conciseness of integrated schemas, low cost of data transformation, and consistency of integrated schemas with original schemas. We focused on the challenges caused by the hierarchical structure of XML. Guided by the criteria, we gave algorithms to merge equivalent (sub) relationship types, and a top-down approach to integrate ORASS schemas.

Reference

[BLN86] C. Batini, M. Lenzerini, S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4), 1986

[CCS00] V. Christophides, S. Cluet, J. Siméon. On wrapping query languages and efficient XML integration. *SIGMOD*, 141-152, 2000

[CDSS98] Sophie Cluet, Claude Delobel, Jérôme Siméon, Katarzyna Smaga. Your mediators need data conversion! *SIGMOD*, 177-188, 1998

[DHW01] D. Draper, A. Y. Halevy, D. S. Weld: The Nimble XML data integration system. *ICDE*, 155-160, 2001

[DRR03] C. Delobel, C. Reynaoud, M. –C. Rousset, J. –P Sirot, D. Vodislav. Semantic integration in Xyleme: a uniform tree-based approach. *Data & Knowledge Engineering 44*, 2003

[GMT02] G. Gardarin, A. Mensch, A. Tomasic. An introduction to the e-XML data integration suite. *EDBT*, 2002

[HL04] Qi He, Tok Wang Ling. Resolving schematic discrepancies in the integration of entity-relationship schemas. *ER*, 2004

[JH01] E. Jeong, C. -N. Hsu. Induction of integrated view for XML data with heterogeneous DTDs. *CIKM*, 2001

[MAG+97] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajaraman. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3), 54-66, 1997

[Matt03] N. M. Mattos. Integrating information for on demand computing. *VLDB*, 8-14, 2003

[May01] W. May. LoPiX: A system for XML data integration and manipulation. *VLDB*, 2001

[MLLA03] Xiaofeng Meng, Daofeng Luo, Mong Li Lee, Jing An. OrientStore: A schema based native XML storage system. *VLDB*, 2003

[Plot01] David Plotkin. Building the XML repository (presentation slides). Available at: http://www.intelligenteai.com/XMLRepository/, 2001

[RB01] E. Rahm, P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4): 334-350, 2001

[RM01] P Rodriguez-Gianolli and J. Mylopoulos. A semantic approach to XML-based data integration. *ER*, 2001

[WLL01] Xiaoying Wu, Tok Wang Ling, Mong Li Lee, and Gillian Dobbie. Designing Semistructured Databases Using ORA-SS Model. *WISE*, 2001

[YLL03] Xia Yang, Mong Li Lee, Tok Wang Ling, Resolving structural conflicts in the integration of XML schemas: a semantic approach. *ER*, 2003