THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Lower Kent Ridge Road, Singapore 119260

TRB9/05

Algebra and the Formal Semantics of GLASS

Wei NI and Tok Wang LING

September 2005

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

Algebra and the Formal Semantics of GLASS
(Technique Report)

Wei Ni Tok Wang Ling

Dept. of Computer Science, SoC, National University of Singapore
10, Kent Ridge Crescent, Singapore, 119260
Tel: 65-68742779
{niwei, lingtw}@comp.nus.edu.sg

Abstract. In database world, it is common to translate a query language into an algebra for the purpose of precisely defining the formal semantics of a query language and doing query optimization later. In this paper, we examine the scenario of graphical XML query languages, focus on their expressive power and present the underlying algebra of our graphical XML query language. Compared with various previous works on XML algebra, our algebra supports not only traditional select, project and join operators but also swap and SQL-like group operators. To achieve the exactness in query representation, we use ORA-SS (Object-Relationship-Attribute model for Semi-Structured data), a semantic rich data model for XML including the information such as Key constraints, Functional dependencies and Relationship types which are lacked in DTD. With examples, we show how our graphical language solves the difficult points in representation and how it is translated into our algebra. Based on the translation, we use the algebra to define the formal semantics of GLASS.

1. Introduction

Although XQuery [24] is a powerful functional language and the standard of XML query today, it is more like a programming language than a query language, which is difficult for common users to use. Some previous works such as [2] have shown that graphical XML query languages and graphical user interfaces (GUIs) can provide more intuitive and easier XML query with capturing the essential power of XQuery (such as FLWOR expressions).

So far, there are various graphical XML query languages and GUIs such as XML-GL [5, 6, 10], Equix [9], BBQ [16, 19], XML Ape [18], QURSED/TQL [22] etc. All these mentioned works are built on the base of XQuery/XPath, that is, they all need to be translated into XQuery/XPath for query evaluation. None of them defines its own algebra for query evaluation or optimization.

So, it is necessary to design an algebra for graphical XML query language? In our opinion, the answer is *Yes* and here are the reasons,

(1) *Graphical representation is different from text*. In graphical query, people focus on the selection predicates and the structure of the result. Ideally, people only concern about what they want instead of iteration, variable binding and XQuery syntax. For example, to swap different element types in different hierarchy is

easy to draw but not easy to write in XQuery. Therefore, we need an algebra that can directly grasp the semantics of graphical query rather than translate them into XML Query for evaluation.

(2) *XML Query algebra is really a suggested standard*. The definition of XML Query algebra in [26] is like a programming language also. Although it defines the formal semantics of XQuery, it dose not support query optimization well enough. In fact, most XML database systems today define their own algebra for query evaluation and optimization such as TAX [13] (in TIMBER), XCQL algebra [21], UnQL algebra [4], Lorel algebra [1], and the works in [11], [12], [23], which have widely covered XML querying, data manipulation (e.g. UPDATE) and query optimization (Notice that the above works are all textual XML query languages).

Since the algebra of graphical XML query language is necessary, w*hat Do We Need inside the algebra*.

Table 1 shows the operators we have in our algebra in comparison with [26] and [13]. We select these two for comparison because [26] is the first proposed standard of XML query and [13] is a typical algebra for native XML databases. Table 1 has listed most operators that so far researchers are interested in. Here we should highlight *two* points in this table:

Table 1. Comparison in operator set among XML Query Algebra, TAX and our algebra.

Different Operators		**XML Query Algebra [26]**	**TAX [13]**	**GLASS Algebra (in this paper)**
Traditional Set Operators (e.g. Union)		–	✓	✓
XML View Specification Operations	SPJ	✓	✓	✓
	Group and Aggregation Functions	✓	✓	✓
	Swap	✓	–	✓
Data Manipulation Operators (e.g. Update)		–	✓	✓
Functions		✓	–	–
Misc.	Order	✓	✓	✓
	Rename	–	✓	✓
	Sort	✓	✓	✓

(1) **Swapping** is one of the most useful operators in graphical XML query languages. The semantic meaning of swapping to change the hierarchical position of two element types in the source schema and get a new XML view. In [26], swapping is used as an example of restructuring, but their definition of restructuring is too abstract since any changes in structure from original data can be regarded as a restructuring. In our project, we find it necessary to define a specific restructuring operator – **Swapping,** especially for graphical XML query language. More details about swapping will be discussed later with examples in this paper.

(2) Besides, the operator **Function** in [26] will NOT be treated as an operator in our algebra. Like a programming language, the *function* in [26] is used to make

XQuery more modular. However, the functionality of *function* can be substituted by various methods in graphical query language such as multi-graph queries, condition identifier (i.e. user-defined ID that identifies a particular sub-part of the query graph), and multiple queries (i.e. a complex query can be step-by-step represented). Beyond that, the *function* operator defines a function rather than generates an XML fragment, which is not consistent with other operators in the algebra. Therefore, in this paper, we will not include function as an operator.

The rest of the paper is organized as follows. In Section 2, we give the preliminary information on ORA-SS and GLASS; then we raise a series of query examples, which are difficult to be expressed by other graphical XML query languages, and demonstrate how they are represented in GLASS. The GLASS algebra is defined in Section 3. The formal semantics of GLASS is defined in Section 4 where the translation from GLASS query into GLASS algebra expressions is also discussed. We summarize the paper and highlight the future works in Section 5. The formal syntax of GLASS is listed in EBNF as an appendix.

2. GLASS & Motivating Examples

2.1 ORA-SS: Semantics in Data Are Crucial

ORA-SS (Object-Relationship-Attribute model for Semi-Structured data) [14] is a data model for semi-structured data. Compared with DTD, XML Schema [28], OEM and XML Graph [10], ORA-SS is semantically richer in the following aspects,

(1) object ID indicates different object instances rather than element instances;
(2) *n*-ary relationship types (where n≥2) also their cardinality constraints can be represented;
(3) relationship attributes are distinguished from object attributes.

ORA-SS is *not* ER (Entity-Relationship Data Model) because ER is used for modeling structured data while ORA-SS can model semi-structured data. ORA-SS is better than DTD, OEM or XML Graph because it represents more semantic information in the XML data. The above three advantages of ORA-SS are *important* to define the query meaning clearly and interpret the query correctly.

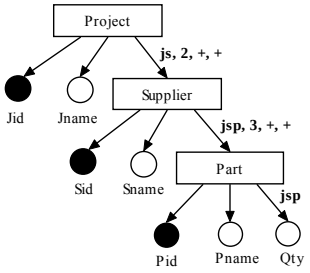


Fig. 1. Example ORA-SS schema diagram

```
Project: [@Jid: J001, Jname: Punch,
    Supplier: [@Sid: S001, @Sname: Jones,
        Part: [@Pid: P002, Pname: Nut, Qty: 500]]
    Supplier: [@Sid: S002, @Sname: Ada,
        Part: [@Pid: P001, Pname: Screw, Qty:700]
        Part: [@Pid: P002, Pname: Nut, Qty: 200]]]
Project: [@Jid: J002, Jname: Tape,
    Supplier: [@Sid: S001, @Sname: Jones,
        Part: [@Pid: P001, Pname: Screw, Qty:400]
        Part: [@Pid: P003, Pname: Nut, Qty: 100]]]
…
```

Fig. 2. Fragment of "project.xml" in compact format, a dataset conforming to the schema in Fig. 1.

Let us show the importance from the data set introduced in Fig. 1 and 2. Fig. 1 is

the ORA-SS schema diagram of an XML document (namely "project.xml") on `project`, `supplier` and `part`. The rectangles indicate that `project`, `supplier` and `part` are object classes. Attributes are represented as circles where the filled circles are object ID. Relationship types are represented as arrows with labels.

Fig. 2 shows a fragment of the "project.xml", a document conforming to the schema diagram in Fig. 1, in a compact format. We present XML data in the compact format in this paper to save the space, where complex element types are expressed as "elementname: […]", simple element types as "elementname: value" and attributes as "@attributename: value".

Here are the important semantics we catch in ORA-SS:

(1) *Object ID of each object class*. In the dataset, `supplier` "S001" appears twice since it supplies `parts` to both `project` "P001" and "P002". In such a case, `Sid` cannot be declared as an ID attribute in DTD. Meanwhile, OEM model will give the two `supplier` elements two different IDs because OEM treats each element instance as a unique object. In contrast, ORA-SS naturally defines `Sid` as an object ID field which means two element instances with the same object ID value are of the same object instance. This is crucial in projection and grouping because the system needs to know which attribute(s) can be used to identify different object instances.

(2) *There are two relationship types specified in Fig. 1*,

"**js, 2, +, +**" means that there is a binary relationship type "js" between `project` and `supplier` where the first "+" means each `project` has one or many `suppliers` and the second "+" means each `supplier` may attends one or many `project`.

"**jsp, 3, +, +**" means that there is a ternary relationship type "jsp" among `project`, `supplier` and `part`, which is defined as a relationship type (jsp) between the relationship type js and `part`. The first "+" means for one `project`, one `supplier` (in js) can provide one or many `parts` to the `project`; and the second "+" means one `part` can be provided in one or many different combinations of `projects` and `suppliers` in js.

This declaration is important because the same DTD structure can have different semantics and cause ambiguities in query representation.

(3) The "jsp" label on the arrow from `part` to `Qty` implies that `Qty` is a *relationship type attribute* of "jsp" rather than the object attribute of `part`. That is, the value of `Qty` is determined by the combination of `Jid`, `Sid` and `Pid` rather than `Pid` only. This information is significant because relationship type attributes should be treated differently in most operators including project, swap, join, etc [7, 8]. One simple example is that, when user wants to drop `supplier` information from the source data, the `Qty` attribute should be applied by some aggregation functions such as SUM so that the result XML view is meaningful in semantics.

2.2 GLASS in a Nutshell

GLASS [20] is a visual XML query language designed on the base of ORA-SS and its graphical representation. GLASS regards each query as a definition of a new XML view (i.e. an XML document). Thus, we extended the notations in ORA-SS to perform the view definition.

A typical GLASS query consists of four parts,

(1) Right Hand Side Graph (RHS Graph) – defines the output structure of the query result. It is a compulsory part for any user queries.
(2) Left Hand Side Graph (LHS Graph) – denotes the basic conditions of a user query. It can contain different structure from the source schema diagram. In such cases, we shall do view validation before evaluating the query.
(3) Link Set – specifies the bindings between the RHS Graph and LHS Graph. When two graph entities (i.e. nodes in query graphs that represent object classes and attributes) are linked, they are visually connected by a line, which means the data type and value of the entity in RHS are exactly the same as the linked entity in LHS.
(4) Condition Logic Window (CLW) – It is an optional part where users write conditions, rules and constructions that are difficult to draw. The CLW includes
   – Logic expressions by using condition identifiers, logic operators including AND, OR and NOT and the quantifiers (EXIST and FORALL);
   – Mathematical expressions and comparison expressions;
   – Reconstruction statements such as the clause:

IF <logic exp. | comparison exp.> THEN EXTRACT …

Most notations in GLASS are borrowed from those in ORA-SS schema diagram. Object classes are represented as rectangles, attributes as circles, relationship types as arrows (with labels) and IDREFs as dashed arrows. New notations such as Box of group entity and Condition Identifier are introduced to represent the query condition and result reconstruction. More detailed information of the language feature and the comparison between GLASS and other graphical query languages can be found in [20].

2.3 Examples, the expressive power of GLASS

In this section, we raise three query examples which can be difficult for other graphical XML query languages to express. All query examples are proposed on the source schema in Fig. 1.

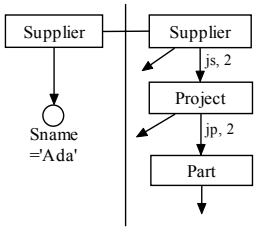


Fig. 3. Query 1, Swapping and result construction.

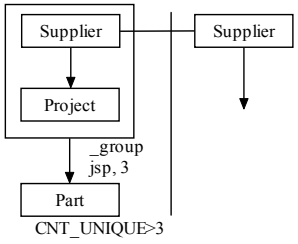


Fig. 4. Query 2, Multi-field grouping and box of group entity

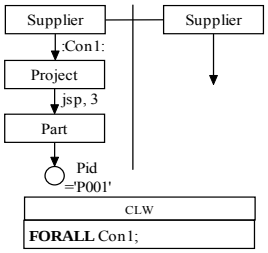


Fig. 5. Query 3, Condition identifier and the use of CLW

(**Query 1**). Find the supplier whose name is "Ada", display all projects which the supplier has supplied parts to, and for each displayed project, list all its different parts supplied by some suppliers. (The hierarchical structure of the result is different from the source schema.)

(**Query 2**). Find the supplier who has provided more than 3 different parts for some

projects. (Contains Multi-field grouping)

(**Query 3**). Find the supplier who provides part "P001" for all projects that it supplies parts to.(Universal Quantifier)

Difficult points and discussion:

(1) *To define the result view precisely*. In Query 1, the hierarchical structure of the result (the RHS graph) is different from the source schema in Fig. 1, i.e., the `supplier` and `project` have been swapped and a new binary relationship type **jp** is derived by projecting `project` and `part` from **jsp**. The declaration of the derived relationship type is crucial. Without it, people will NEVER know whether the `part` is "all different `parts` provided by any `supplier` for this `project`" or "just the `parts` provided by current `supplier` for this `project`". This example shows that without the semantic declaration, the graphical query can be ambiguous.

(2) *To express the (multi-field) aggregation in an easy way*. Query 2 gives an example of multi-field aggregation where the group-by field contains more than one object class and/or attribute. In Fig. 4, a box of group entity is applied, which includes `supplier` and `project`, so that the labels "_group" and "jsp, 3" on the arrow from the box to `part` means "to group `part` under each pair of `supplier` and `project` in the ternary relation **jsp**".

(3) *To express quantifiers and even complex query conditions in graphs*. Query 3 poses the problem of expressing universal quantifier. In this example, there is a universal quantifier holding the constraint that "all `projects` corresponding to the selected `supplier` must use `part` P001 provided by the `supplier`". To express this constraint, a condition identifier "Con1" is introduced as shown in Fig. 5. The label "jsp, 3" in Fig. 5 indicates that the constraint is applied in the ternary relationship jsp. The universal quantifier is declared in CLW with the condition identifier.

The above three query examples demonstrate the different traits of GLASS from other graphical XML query languages. The most important different point is that GLASS is not a path-based graphical query language, i.e. a path in GLASS query graph is NOT an XPath equivalent. The LHS graph can be regarded as a temporary view with constraints (also the constraints in CLW); the RHS graph is a result view defined by user; and the links indicate whether instances in RHS come from the temporary view of LHS or the source. Compared with XML-GL and TQL, GLASS

(1) explicitly defines the relationship constraints in both LHS and RHS graph;
(2) explicitly defines the bindings between LHS graph and RHS graph;
(3) explicitly defines group-by field(s) in LHS graph;
(4) flexibly uses different hierarchical structure from source schema in both LHS and RHS graphs;
(5) separately expresses complex query logic such as quantifier, negation and IF-THEN construction in CLW.

3. GLASS Algebra

In this section, we discuss the underlying algebra of GLASS query, namely GLASS

algebra, by presenting a list of operators on XML views (Section 2.2) including traditional set operators, SPJ operators, swapping and grouping. Each operator takes one or more XML views as its operand(s) and produces a new XML view as its result.

3.1 Traditional Set Operators

To apply the traditional set operators Union, Intersection and Difference, two operand views must be union-compatible. In the scope of XML view, we define two XML view union-compatible if they conform to the same ORA-SS schema. And two instances are identical if they match in type, structure and equal in value with the consideration of relationship types and order-sensitiveness in ORA-SS schema.

3.2 XML View Specified Operations

In [7], 4 view constructing operators have been proposed as select, swap, drop and join. In our algebra, we reuse the *select* and *swap*, complement the *drop*, extend the *join* in [7] and explicitly define a SQL-like *grouping*.

3.2.1 Selection

The selected result is an XML View that conforms to the ORA-SS schema of the XML view and satisfies the selection predicates. A selection operator is in the form of $\sigma_C(V)$ where C is the selection predicate and V is an XML view. For example, to select all contents that the `project` has some `supplier` providing `part` 'P001', we would write:

$$\sigma_{@Pid = \text{'P001'}}(\text{project.xml})$$

and the result of selection is:

```
Project: [@Jid: J001, Jname: Punch,
 Supplier: [@Sid: S002, @Sname: Ada,
    Part: [@Pid: P001, Pname: Screw, Qty:700]]]
Project: [@Jid: J002, Jname: Tape,
 Supplier: [@Sid: S001, @Sname: Jones,
    Part: [@Pid: P001, Pname: Screw, Qty:400]]]
…
```

The selection predicate C can contain

(1) comparison operator =, ≠, <, ≤, ≥ and >;
(2) XPath expressions and functions;
(3) existential quantifier EXIST (Like SQL, the universal quantifier is represented by using negated existential in GLASS algebra.)[1];
(4) logic operator ∧ (and), ∨ (or) and NOT.

For example, $\sigma_{\text{Part}[@Pid = \text{'P001'} \wedge \text{Qty} > 500]}(\text{project.xml})$ will select all contents that the `project` has some `supplier` providing more than 500 pieces of `part` 'P001'.

Comparison: the Selection in GLASS algebra is similar to that in XML Query Algebra [26] except that the selection predicate in GLASS algebra supports negation (NOT), existential quantifier and logic operators, which are usually represented by quantification and function operators in [26]. In GLASS algebra, the functionality of

[1] The universal quantifier can be rewritten by using negated-existential quantifier (i.e. combine NOT and EXIST) so that we do not use universal quantifier directly in selection predicates.

quantification is merged into predicate fields of each operation.

3.2.2 Projection and X-projection

Projection is applied to get a collection of one specified element or attribute type from the original XML view. The projection operator is denoted as $\Pi_P(V)$ where P is the project component from the XML view V. Notice that P is similar to an XPath. For example, to project all `parts`' name (Pname), we will write,

$$\Pi_{//\text{Pname}}(\text{project.xml})$$

and the result will be

Pname: Nut, Pname: Screw, Pname: Nut, Pname: Screw, ……

which is a list of `Pname` elements possibly with duplicates. To eliminate the duplicates, we use key word UNIQUE such as

$$\Pi_{\text{UNIQUE}(//\text{Pname})}(\text{project.xml}).$$

To project one or more object classes with their attributes according to the ORA-SS schema of V, we define the extended-projection (or **x-projection**) operator. X-projection is denoted as $x\Pi_T(V)$ where T is the expression that explicitly indicates the wanted object classes together with their attributes in the form of:

obj[attr, attr, …], obj[attr, attr, …], …

where *obj* is the object class name in ORA-SS, *attr* is either the object attribute name or relationship attribute name in ORA-SS. Notice that, the result should be kept in the same hierarchical structure as the original schema. To secure that the result view is meaningful, we suggest that T should satisfy that,

(1) There is one relationship type R in ORA-SS schema that covers all object classes in T;
(2) For each attribute type a in T, if a is an attribute of an object class A, then A must be in T; and if a is an attribute of an n-ary relationship type R consists of object classes $A_1, A_2, \ldots, A_n$, then $A_1, A_2, \ldots, A_n$ must be in T.

For example, to obtain an XML view that contains `project` (`Jid` and `Jname`) and `part` (`Pid` and `Pname`) without `suppliers` from the original document, we would write

$$x\Pi_{\text{Project[@Jid, Jname], UNIQUE(Part[@Pid, Pname])}}(\text{project.xml})$$

and the result will be

Project: [@Jid: J001, Jname: Punch,
  Part: [@Pid: P002, Pname: Nut]
  Part: [@Pid: P001, Pname: Screw]]
Project: [@Jid: J002, Jname: Tape,
  Part: [@Pid: P001, Pname: Screw]
  Part: [@Pid: P003, Pname: Nut]]
…

Notice that, the duplicates in `part` elements are eliminated in the result by UNIQUE key word. Since the result is kept in the same hierarchical structure of the source schema in Fig. 1, the ancestor-descendant relation between each pair of `project` and `part` instances are also kept. Therefore, the key word UNIQUE eliminates duplicate `parts` under each `project`.

Counterexample: In contrast, the following expression

$$x\Pi_{\text{Project[@Jid, Jname], UNIQUE(Part[@Pid, Pname, Qty])}}(\text{project.xml})$$

is legal but **NOT** meaningful because it violates the second constraint of x-projection where `Qty` is an attribute of the ternary relationship type **jsp** consists of `project`,

`supplier` and `part`, but `supplier` is not in the project field. As we mentioned in Section 2.3, there should be some kind of aggregation applied on `Qty` to keep the result meaningful. In GLASS, meaningful checking is automatically done by the system and improvement choices will be given to the user when a violation is detected.

Comparison: the Projection in GLASS algebra plays both the role of "projection" and "iteration" in XML Query Algebra [26]. It should be emphasized that, the duplicate elimination is optional by using the key word UNIQUE. The **x-projection** can project multiple object classes and attributes in ORA-SS (or different element and attribute types in XML) at a time. Notice that, the projection in [26] cannot play the same role as x-projection does. Of course, by using XPath, the projection in [26] can also project multiple element/attribute types in XML. However, without the two constraints of x-projection, the result will not be secured to be meaningful such as the counterexample of x-projection. Notice that, such semantics in the constraints cannot be found in DTD, XML Schema or their equivalents but available in ORA-SS.

It should be mentioned that, our projection and x-projection is a complement to the drop operator in [7]. Drop operator removes those object classes or attributes that are not needed in the result, which is very useful when only a few object classes or attributes are removed.

It should be emphasized that, x-projection can not replace projection in functionality. Projection is used to get a list of one specified attributes (attribute types or simple element types with PCDATA in XML). In contrast, x-projection obtains multiple object classes and their attributes and keeps the original hierarchical structure.

3.2.3 Swapping

Swapping operator is one of the most significant operators in [7] and our algebra with the help of ORA-SS. In graphical query language, it is very common and easy to express the swapping that changes the hierarchical position of two object classes in their hierarchical path. This is useful in both specifying query conditions and defining result structure.

Swapping operation obtains a new XML view by changing the position of two object classes in one path, i.e. it changes the ancestor-descendant relationship between two object classes in ORA-SS schema. The basic idea of swapping is "object attribute follows object class and relationship type attribute follows relationship type" to generate valid XML view, which requires the semantic information in ORA-SS. The swap operator is defined in the form of

$$\mathrm{Swap}_{A,\,B}(V)$$

where V is the XML view, A and B are two object classes, which reads "Swap A and B in V".

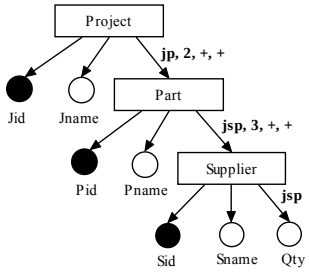


Fig. 6. The ORA-SS schema diagram after Part and Supplier is swapped in Fig. 1.

Notice that, the swapping operation is not always as simple as the restructuring in [26]. For example, if a user wants to swap the hierarchical position between Part and Supplier in "project.xml" based on the ORA-SS schema diagram in Fig. 1, he can write $Swap_{Part, Supplier}$(project.xml). The ORA-SS schema diagram of the result will be like Fig. 6. According to the rules in [7],

(1) A new binary relationship type jp between `project` and `part` is derived from the original jsp
(2) The object attributes are move up/down with their corresponding object classes; and
(3) The relationship type attributes stay with the lowest object class of the relationship type in the hierarchical structure.

To do such a swapping in XQuery can be very complex and the writer should know well about the semantics of the data. Unfortunately, such semantics as object ID, relationship types and relationship type attributes cannot be found in DTD. Without knowing the object ID, the part elements will not be able to be merged; without knowing the relationship types, if there is a change in semantics from the original data, it cannot be detected; and without knowing the affiliation between attributes and relationship types (or Functional Dependencies [14]), whether an attribute should follow its parent object class or stay with a relationship type cannot be easily decided. Therefore, we need ORA-SS to secure that the swapping operation works correctly and generates meaningful XML views.

Comment: Although the semantic of swapping operation is simple, it can be very complex when it is handled in XQuery because it depends on how well the user knows about the data and its semantics. Therefore, a rich semantic data model is helpful in interpreting the query meaning and checking whether the result is meaningful.

In GLASS, swapping operation is very common. However, to swap two object classes in the same relationship type without changing other relationship types in an ORA-SS schema is "safe" because no semantic changes will be caused by the operation. Therefore, when a GLASS query is translated into algebraic expressions, we need decompose the query graphs (both LHS and RHS) into small unit where each unit is a sub-graph of the original one that contains only one relationship types.

3.2.4 Join and Outer-join

Join is defined in [7] as "to join two object classes and their attributes together by foreign key to key reference". In our algebra, beside the above definition, we also use join operator to join two object classes (with their attributes) by *non-key attribute values*.
The joined fields (i.e. the object classes or attributes) can be either in one XML document or from different XML documents.

For example, if a user wants to change the ternary relationship type jsp in the original schema diagram into two binary relationship types, `project-supplier` (js) and `supplier-part` (sp), where js is the same as the original one and sp is derived from original jsp by projecting `supplier` and `part`, he can write the following expressions to get view V1 and V2. (V1 contains the original binary relationship type js which contains all projects and, under each project, there are suppliers that have supplied parts to the project. V2 contains the derived binary relationship type sp which contains all suppliers and, under each supplier, there are all parts that have been supplied by the supplier to some projects.)

$V1=x\Pi_{Project[@Jid, Jname], Supplier[@Sid, Sname]}$ (project.xml)

$V2=x\Pi_{UNIQUE(Supplier[@Sid, Sname]), UNIQUE(Part[@Pid, Pname])}$(project.xml)

then the join operation

$$V1 \bowtie_{Supplier[@Sid]} V2$$

means to join the views V1 and V2 on `supplier`'s `Sid`. The schema diagram of the result is shown in Fig. 7.

Notice that, the join operator still needs the semantic information in ORA-SS. Consider the schema diagram in Fig. 7 and the RHS graph of Query 1 in Fig. 3, when the join field is an object class (e.g. Supplier in this example), we need key attributes (object IDs) to know whether two supplier instances are identical.

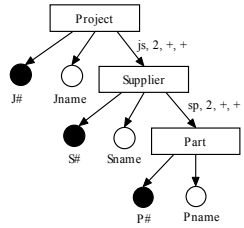


Fig. 7. The ORA-SS schema diagram of the join result**.**

Based on the join operator, we also define **outer-join** in GLASS algebra. The definition of outer-join is similar to join except that a special element (attribute) type with NULL will be introduced in the result when one join field has no counterpart in the other join field of two object classes (or two attributes).[2]

For example, suppose we have two XML views as follows,

```
V3:
    Project[@Jid: J001,
            Supplier[@Sid: S001]
            Supplier[@Sid: S002]
    ]
V4:
    Supplier[@Sid: S002,
            Part[@Pid: P001]]
    Supplier[@Sid: S003,
       Part[@Pid: P001]]
```

then the expression

V3 **OUTERJOIN** V4 **ON** Supplier[@Sid]

will get the following result

```
Project[@Jid: J001,
        Supplier[@Sid: S001]
        Supplier[@Sid: S002, Part[@Pid: P001]]
Project_NULL[
        Supplier[@Sid: S003, Part[@Pid: P001]]]
```

where the element type Project_NULL contains all `orphan-supplier` elements.

It should be emphasized that, since XML document can be order-sensitive, the order between the two join/outer-join operands are important. In other words, in XML, we do not have "V1⋈ V2 equal to V2⋈V1".

Comparison: the Join operator in GLASS algebra is a natural join defined as an extension of the join in [7]. Similar to swapping, we need a set of rules to guarantee meaningful join result. Here we only use simple examples to show the idea of join and outer-join. More complex cases and detailed rules can be found in [7].

[2] Here, the join field can be one of the three: foreign key, key reference, value of attributes (including non-key attributes).

3.2.5 Grouping and Aggregation Function

The Grouping operator creates a new XML view from the source data by grouping an object class or an attribute under other object classes (and their attributes) with or without applying certain aggregation functions. The general form of grouping is

$$\text{GROUP}_A^B(V)$$

where *B* is the group-by fields, *A* is the grouped fields (with or without aggregation functions), *V* is an XML view, which reads "group *A* under *B* from *V*".

For example, to obtain a view of `project` and `part`, for each `part` we need the total `Qty` of the `part` for each `project`, we would write

$$\text{GROUP}_{\text{SUM(Part[Qty])}}^{\text{Project[@Jid], Part[@Pid]}}(\text{project.xml})$$

which means to group Qty under each combination of `project` and `Part` (by using their object IDs) in the hierarchical structure of XML document "project.xml" and get the SUM of `Qty` attributes under `part` for each group, and the result will be,

```
Project: [@Jid: J001,
        Part: [@Pid: 002, SUM_Qty: 700]
        Part: [@Pid: 001, SUM_Qty: 700]]
Project: [@Jid: J002,
        Part: [@Pid: 001, SUM_Qty: 400]
        Part: [@Pid: 003, SUM_Qty: 100]]
...
```

where SUM_Qty is the default name of the derived attribute of the binary relationship type between project and part. The derived attribute is displayed as simple element types (i.e. element types with PCDATA only) in the result XML view.

The allowed aggregation functions include CNT, SUM, AVG, MAX, MIN and the keyword UNIQUE for eliminating duplicates.

Comment: In comparison with XML Query Algebra [26], we define a SQL-like grouping operator that can easily express multi-field aggregation (i.e. the group-by field contains more than one object class or attribute). Compared with TAX [13], our grouping operation keeps the original hierarchical relation within the group-by fields. It seems that the group-by fields must come from the ancestor element nodes (in XML) of those in grouped field. However, combined with Swapping operation, our grouping operator can also handle the case when we should group ancestor nodes by their descendants.

4. The Formal Semantic of GLASS

In this section, we define the formal semantic of GLASS by using the operators defined in Section 3.

4.1 LHS Graph and Logic Expressions in CLW

Definition 4.1 (General-connected) Two nodes n and m are general-connected if
(1) there is an arrow or dashed arrow[3] connecting n and m; or
(2) there is an arrow or dashed arrow connecting n and a group box that contains m

[3] As defined in [12], arrow is relationship type (basically the parent-child relationship between two nodes) and dashed arrow is the key reference (including the ID reference in XML).

Definition 4.2 (General-connected Graph) A graph <N, E> is a General-connected graph if
(1) N contains only one node; or
(2) $\forall n \in N, \exists m \in N$ that $n \neq m$, n and m are general-connected.

Definition 4.3 (Single-Object-Class Query Graph and Single-Relationship-Type Query Graph) We decompose the LHS/RHS Graph as follows,

(**Type 1**) the general-connected sub-graph formed by an object node with its object attribute nodes is a Single-Object-Class query graph or *SOG*; (e.g. the LHS graph of Fig. 3)

(**Type 2**) if there is a relationship type in ORA-SS schema, which contains all object nodes and relationship type attributes nodes in the general-connected sub-graph, then it forms a Single-Relationship-Type query graph or *SRG*; (e.g. the LHS graph of Fig. 4 and 5)

(**Type 3**) if there is a derived relationship type (defined by user on the base of the relationship types in ORA-SS schema) that contains all object nodes and attribute nodes in the general-connected sub-graph, then it also forms a *SRG*.

The algorithm that decomposes the LHS/RHS graph into a set of SOG and SRG is to

Step 1. Traverse all the object classes and relationship types in the ORA-SS schema, then we can generate all SOG (Type 1) and SRG (of Type 2). We mark the corresponding nodes and edges in the LHS/RHS graph and

Step 2. Each un-marked edge in the LHS/RHS with its nodes forms the SRG of derived relationship types (Type 3).

Notice that, the decomposition of LHS and RHS are processed separately.

When a node appears in two SOGs or SRGs, they will be marked to be the same so that the original LHS/RHS graph can be restored[4]. Remarkably, the LHS graph of Fig. 3, 4 and 5 are SOG/SRG already.

The SOG and SRG are the basic units in our query graph interpretation. Since each SOG/SRG has one relationship type covering the whole graph (each SOG is covered by its corresponding object class type), the interpretation of each SOG/SRG is secured to be meaningful. To interpret the SOG/SRG, we will use the operators defined in Section 3 as the examples demonstrated in Fig.7 and 8. The interpretation is almost straightforward.

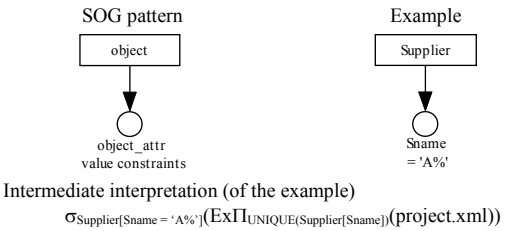
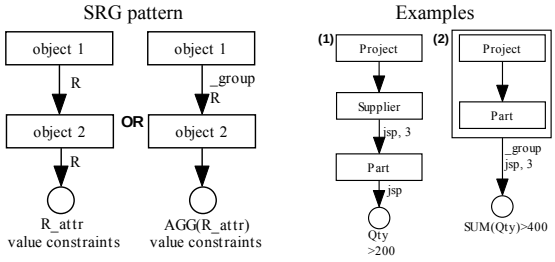


Fig. 3. Interpretation of SOG (Type 1)

[4] There are many ways to make the mark, such as using the same node ID, etc.



Intermediate interpretation (of the examples)

(1) $\sigma_{Qty>200}$(project.xml)

(2) $\sigma_{SUM(Part[Qty])>400}$($GROUP^{Project[@Jid],\ Part[@Pid]}_{SUM(Part[Qty])}$ (project.xml))

Fig. 4. Interpretation of SRG (the two patterns can be found in both Type 2 and 3)

The selection predicates of value comparison can be directly found in each SOG/SRGs; and the logic operator AND and OR can be found in CLW. The interpretation on quantifier (EXIST and FORALL) will be discussed below.

In CLW, the syntax of logic expressions can be described as

Expression := [**FORALL | NOT EXIST**] Condition Identifier';'
Expression := Expression [**AND | OR**] Expression';'
Expression := **NOT** Expression';'

Definition 4.4 (Active range of Condition Identifier) Given a condition identifier *Cid*, the active range of *Cid*, denoted as T_{Cid}, is a sub-graph of the LHS graph that

(1) If an arrow (or dashed arrow) E_{AB}, pointing from node A to B, is assigned by *Cid*, then $\{E_{AB}, A, B\} \subseteq T_{Cid}$.
(2) If two arrows (or dashed arrows) E_{AB}, E_{BC}, making a directed path from node A to C via B, and $E_{AB} \in T_{Cid}$, then $\{E_{BC}, C\} \subseteq T_{Cid}$.

Intuitively, if a condition identifier *Cid* is assigned on the arrow from node *A* to node *B*, then, following the direction of arrows (and dashed arrows), the active range of *Cid* covers node A, B, the arrow from A to B and all nodes and edges below node B.

Definition 4.4 finds that, in the active range of each condition identifier, the root node *has one and only one* child node, and the arrow from root node to its single child node is the place where *Cid* is assigned. Moreover, we will have $T_{Cid1} \supseteq T_{Cid2}$, if *Cid2* is assigned on an arrow in T_{Cid1}.

Definition 4.5 (Satisfaction of condition identifier) Given a condition identifier *Cid*, T_{Cid} is its active range,

(1) If T_{Cid} does not contains any other condition identifiers, then *Cid* is satisfied if there is a sub-document-tree in source that matches the query conditions in T_{Cid}.
(2) If T_{Cid} contains other condition identifiers, then *Cid* is satisfied if there is a sub-document-tree in source that matches the query condition in T_{Cid} and the logic expressions (in CLW) that consist of the condition identifiers in T_{Cid}.

Then, in CLW, the atomic expression with one single condition identifier is TRUE if

the condition identifier is satisfied. Besides, we find the expression useless when there is a binary logic operator AND or OR connecting two different condition identifiers *Cid1* and *Cid2* where $T_{Cid1} \supseteq T_{Cid2}$ because in such a case, the satisfaction of *Cid1* implies the satisfaction of *Cid2*.

Now we can describe the meaning of the expressions with quantifiers.

Definition 4.6 (The semantic of universal-quantified expressions on condition identifier in CLW) Given a condition identifier *Cid*, assigned on an arrow (or dashed arrow) from node A to node B,

CASE 1. A is an object class and B is A's attribute. Then "**FORALL Cid**" is TRUE when all values of B (especially multi-valued attribute B) of current object instance of A make *Cid* satisfied.

CASE 2. A is an object class and B is an attribute of relationship type *R*, *R* is an n-ary relationship type consists of *n* object classes A_1, A_2, …, A_{n-1} and A. Then "**FORALL Cid**" is TRUE when all values of B corresponding to current hierarchical path $obj_1/\ldots/obj_{n-1}/A/B$ in *R* make *Cid* satisfied, where obj_k (*k* from 1 to n-1) is the instance of object class A_k.

CASE 3. A and B are both object classes in relationship type *R*, R is an n-ary relationship type consists of *n* object classes A_1, A_2, …, A_{n-2}, A and B Then "**FORALL Cid"** is TRUE when all instances of B (with their object attributes) corresponding to current hierarchical path $obj_1/\ldots/obj_{n-2}/A/B$ in *R* make *Cid* satisfied, where obj_k (*k* from 1 to n-2) is the instance of object class A_k.

The semantic of existential-quantified expression can be defined similarly.

Based on the above definitions, we define the evaluation of LHS with logic expression in CLW as follows.

Definition 4.7 (*Companionate* and *Independent* SOG/SRGs) Given two SOG/SRGs[5] G_1 and G_2,

(1) G_1 and G_2 are *companionate* (denoted as $G_1 \infty G_2$) if G_1 and G_2 have some common nodes.

(2) G_1 and G_2 are *independent* iff G_1 and G_2 do not have any common nodes.

When we do the evaluation, we should combine all the intermediate interpretations of each SOG/SRGs according to the logic expressions in CLW.

Definition 4.8 (Evaluation) The evaluation of two SOG/SRGs is a temporary XML view defined as follows,

Given two SOG/SRGs G_1 and G_2, $G_1 \infty G_2$, and their intermediate interpretation $Int(G_1)$ and $Int(G_2)$, If there are two condition identifier Cid_1 in G_1 and Cid_2 in G_2, and there is a logic expression in CLW in the form of "Cid_1 *Op* Cid_2", where *Op* is the logic operator AND or OR.

CASE 1. If the logic operator is AND, then the evaluation of G_1 and G_2 is a natural join between $Int(G_1)$ and $Int(G_2)$ on *N* where *N* is the common node (or the common path);

[5] In this paper, the expression "two SOG/SRGs" means the cases of two SOGs, or two SRGs, or one SOG and one SRG.

CASE 2. If the logic operator is OR, then the evaluation of G_1 and G_2 is a full outer-join between Int(G_1) and Int(G_2) on N where N is the common node (or the common path).

The evaluation of LHS graph and CLW is the result after all SOG/SRGs of the LHS are evaluated; the evaluation result is still an XML view.

4.2 RHS Graph and Result View Definition

The RHS graph in GLASS is a schema definition of the result XML view.

Definition 4.9 (Construction) The construction is a mapping set M from the source data X_S (with its ORA-SS schema) and the evaluation result of the LHS (and CLW) E to the XML view V_R (i.e. the RHS graph) that,

$M = M_{Snode} \cup M_{Srel} \cup M_{Enode} \cup M_{Erel}$

where

M_{Snode} is the mapping from nodes in X_S to nodes without links in V_R;
M_{Srel} is the mapping from relationship types in X_S to relationship types in V_R;
M_{Enode} is the mapping from linked nodes in E to nodes with links in V_R;
M_{Erel} is the mapping from relationship types in E to relationship types in V_R.
(M_{Erel} is available iff all object classes participating the relationship type are linked.)

We denote the construction from X_S and E to V_R on the link set L as $M_L((X_S, E) \rightarrow V_R)$.

In the above definition, the result construction is considered as a set of mapping rules. The mapping rules are all GLASS algebra operations, which are obtained similarly to the evaluation of LHS graph. The RHS graph are decomposed into SOG/SRGs like the LHS graph (as mentioned in Definition 4.3), interpreted (also similar to the intermediate interpretation of the SOG/SRGs in LHS graphs) and combined together via a series of *join* operations.

For example, consider Query 1 (in Fig. 3),
(**Query 1**) Find the supplier whose name is "Ada", display all projects which the supplier has supplied parts to, and for each displayed project, list all its different parts supplied by some suppliers.

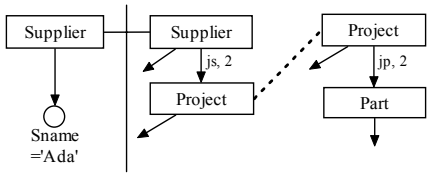


Fig. 9. Decomposed Query 1.

To explain the situation when one object class (or attribute) node appears in two or more SOG/SRGs, we just copy the corresponding node twice or more times. As an instance, the `project` node in the decomposed RHS of Query 1 appears in both SRGs of the RHS. We use a dotted line connecting the two `project` nodes in Fig. 9, which indicates that they are the copies of the same project node in the original query graph. (The dotted line is just for illustration.)

According to Fig. 8,
The LHS graph of Query 1 is translated as (V1 is the Evaluation of LHS.)

$V1 = \sigma_{Supplier[Sname='Ada']}(x\Pi_{UNIQUE(Supplier[Sname])}(project.xml))$

The decomposed RHS graph is translated as

$V2 = Swap_{Project,\ Supplier}($
$x\Pi_{Project[@Jid,\ Jname],\ Supplier[@Sid,\ Sname]}(project.xml))$

$V3 = x\Pi_{Project[@Jid,\ Jname],\ UNIQUE(Part[@Pid,\ Pname])}(project.xml)$

Then the mapping rules with join would be

(V4 is the reconstruction of the RHS graph)

$V4 = V2 \rtimes_{Project[@Jid]} V3$

(V_R is the result, which is rendered by the link between the LHS and RHS)

$V_R = V4 \rtimes_{Supplier[Sname]} V1$

Thus, we get the final translation result of Query 1 as follows,

$V1 = \sigma_{Supplier[Sname='Ada']}(x\Pi_{UNIQUE(Supplier[Sname])}(project.xml))$

$V2 = Swap_{Project,\ Supplier}($
$x\Pi_{Project[@Jid,\ Jname],\ Supplier[@Sid,\ Sname]}(project.xml))$

$V3 = x\Pi_{Project[@Jid,\ Jname],\ UNIQUE(Part[@Pid,\ Pname])}(project.xml)$

$V4 = V2 \rtimes_{Project[@Jid]} V3$

$V_R = V4 \rtimes_{Supplier[Sname]} V1$

Finally, we should mention the IF-THEN clause in CLW. IF-THEN clause indicates that a part of the mapping rules in *M* are only effective when the logic expression after IF is TRUE. The IF-THEN clause is in the form of

IF (*exp*) THEN EXTRACT (node identifier)

which means when the logic expression *exp* is true, we should do the mapping of the specified node and the corresponding relationship types. The *exp* in the IF-THEN clause is independent of the other logic expressions in the CLW so that we ignore the IF-THEN clauses when evaluating the LHS and CLW. However, we should consider them in result construction.

Definition 4.10 (Result of GLASS query) Given an XML source data X_S (with its ORA-SS schema) modeled by ORA-SS and a GLASS query graph Q_G, then the result of Q_G on X_S is given by $M_L((X_S, E) \rightarrow V_R)$, where L is the link set in Q_G, E is the evaluation result of the LHS (and CLW) and V_R is the result view defined in RHS.

Similarly, we can also get the algebraic expressions of Query 2 and 3.

(**Query 2,** in **Fig. 4.**) Find the supplier who has provided more than 3 different parts for some projects.

$V1 = \sigma_{CNT(Part[@Pid])>3}(GROUP^{Supplier[@Sid],Project[@Jid]}_{Part[@Pid],CNT(Part[@Pid])}(project.xml))$

$V2 = x\Pi_{UNIQUE(Supplier[@Sid,\ Sname])}(project.xml)$

$V_R = x\Pi_{UNIQUE(Supplier[@Sid,\ Sname])}(V2 \rtimes_{Supplier[@Sid]} V1)$

In this example, the meaning of "GROUP BY…HAVING…" in traditional SQL/OQL/XQuery is represented by the combination of grouping and selection (V1). To get the result, we need a join between V1 and V2 because V1 does not contain the non-key attribute (such as Sname) of the group-by fields.

(**Query 3,** in **Fig. 5.**) Find the supplier who provides part "P001" for all projects that it supplies parts to.

$V1 = x\Pi_{UNIQUE(Supplier[@Sid, Sname])}(project.xml)$

$V2 = x\Pi_{UNIQUE(Supplier[@Sid, Sname])}($
$\sigma_{NOT\ EXIST(Part[@Pid='P001'])}(project.xml))$

$V_R = V1-V2$

In this example, the universal quantifier FORALL has been transformed into negated-existential quantifier since we do not use universal quantifier directly (like SQL). V1 gets all `suppliers` (with `Sid` and `Sname`) from the source document. Then, V2 gets all `suppliers` of those `projects` that never use `Part` 'P001'. Finally, we use the set operator difference to get the result by eliminating the `suppliers` in V2 from all `suppliers` in V1.

5. Summary and Future Works

In this paper, we focus on the underlying algebra of our graphical XML query language (GLASS). We have shown that the semantic information in ORA-SS is very important to defining the meaning of graphical XML query clearly and generating meaningful query result. Without the semantic information such as object ID, relationship type and attribute affiliations, the correctness of algebra operators (swapping, join and x-projection) cannot be achieved. By translating GLASS query graph into algebraic expressions, we are now ready to use the algebraic query plan to do query evaluation and optimization.

Our future works include three major aspects: first, we shall find the property of the algebra and optimization rules; second, we will the translate GLASS to XQuery based on the algebra and ORA-SS so that our graphical language can be used in native XML environment and third, we need to enhance our current case tools of GLASS.

References

[1] Abiteboul, S., Quass, D., McHugh, J., Widon, J., and Wiener, J. The Lorel query language for semistructured data. Journal on Digital Libraries, 1(1), 1996.

[2] Bonifati, A., Ceri, S. Comparative Analysis of Five XML Query Languages. SIGMOD Record, Vol. 29, No. 1, Mar, 2000.

[3] Buneman, P., Davidson, S., Fan, W. F., Hara, C., and Tan, W. C. Keys for XML. In Proceedings of the WWW10, Hong Kong, China, May, 2001.

[4] Buneman, P., Fernandez, M., Suciu, D.: UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. VLDB Journal 9 (2000) 76--110

[5] Ceri, S., Comai, S., Damiani, E., Fraternali, P., Paraboschi, S., and Tanca, L. XML-GL: a graphical language of querying and restructuring XML documents. In Proc. WWW8, Toronto, Canada, May 1999.

[6] Ceri, S., Comai, S., Damiani, E., Fraternali, P., and Tanca, L.. Complex Queries in XML-GL. SAC(2) 2000:888-893.

[7] Chen, Y. B., Ling, T. W., Lee, M. L. Designing Valid XML Views. ER2002, Oct 7-11, 2002, Tampere, Finland.

[8] Chen, Y. B., Ling, T. W., Lee, M. L. Automatic Generation of XQuery View Definitions from ORA-SS views. (ER'2003), 13-16 October 2003, Chicago, Illinois, USA.

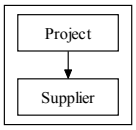
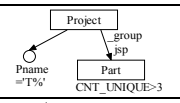
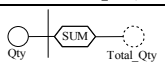
[9] Cohen, S., Kanza, Y., Kogan, Y., Nutt, W., Sagiv, Y., and Serebrenik, A. Equix – Easy Querying in XML Databases. In Proc. of Webdb'98 – The Web and Database Workshop, 1998.

[10] Comai, S., Damiani, E., and Fraternali, P. Computing Graphical Queries over XML Data. ACM Transactions on Information Systems, Vol. 19, No. 4, October 2001, Pages 371-430.

[11] Frasincar, F., Houben, G., and Pau, C. XAL: an Algebra for XML Query Optimization. 13th Australasian Database Conference (ADC2002), Melbourne, Australia.
[12] Hosoya, H. and Pierce, B. C. XDuce: A Typed XML Processing Language. In Proc. Int. Workshop on Web and Databases, May 2000.
[13] Jagadish, H. V., Lakshmanan, L. V. S., Srivastava, D., and Thompson, K. TAX: A Tree Algebra for XML, In: Proceedings of 8th International Workshop on Databases and Programming Languages, Rome, Italy, September 2001.
[14] Lee, M. L., Ling, T. W., and Low, W. L. Designing functional dependencies for xml. In Proceedings of EDBT 2002.
[15] Ling, T. W., Lee, M. L., Dobbie. G. Semistructured Database Design. Springer Science+Business media, Inc. 2005
[16] Ludaescher, B., Papakonstantinou, Y., and Velikhov, P. Navigation-driven evaluation of virtual mediated views. In Proceedings of EDBT 2000 (Konstanz, Germany, March).
[17] Luo, D. F., Chen, T., Ling, T. W. and Meng, X. F. On view transformation support for a native XML database. DASFAA 2004, 226-231, Korea, March 17-19, 2004.
[18] Mark, L., etc. XMLApe. College of Computing, Georgia Institue of Technology. http://www.cc.gatech.edu/projects/XMLApe/
[19] Munroe, K. D., Ludaescher, B., and Papakonstantinou, Y. Blended Browsing and Querying of XML in Lazy Mediator System. Konstanz, Germany, March 2000.
[20] Ni, W., and Ling, T. W. GLASS: A Graphical Query Language for Semi-Structured Data. In Proc. DASFAA 2003, Kyoto, Japan.
[21] Papakonstaninou, Y., Borkar, V., Orgiyan, M., Stathatos, K., Suta, L., Vassalos, V., and Velikhov, P. XML queries and algebra in the Enosys integration platform. Data & Knowledge Engineering, Vol. 44, Issue 3 (March 2003), Page 299-322.
[22] Papakonstantinou, Y., Petropoulos, M., and Vassalos, V. QURSED: Querying and Reporting Semistructured Data. ACM SIGMOD 2002, Jun 4-6, Madison, Wisconsin, USA.
[23] Tatarinov, I., Ives, Z. G., Halevy, A. Y., and Weld, D. S. Updating XML. In Proc. SIGMOD, 2001
[24] XQuery 1.0: An XML Query Language. W3C Working Draft 4th April 2005 http://www.w3.org/TR/xquery/
[25] XML Path Language (XPath) 2.0. W3C Working Draft 4th April 2005 http://www.w3.org/TR/xpath20/
[26] XML Query Algebra. W3C Working Draft 04 December 2000 http://www.w3.org/TR/2000/WD-query-algebra-20001204
[27] XML Query Use Cases. http://www.w3.org/TR/xquery-use-cases/
[28] XML Schema. http://www.w3.org/XML/Schema

APPENDIX: The Syntax of GLASS

Table 2. Textual equivalents to GLASS query graph

	GLASS notations	Textual equivalents
1		$Obj_Project1:= Project, URL, <;
2		$Attr_Pname1:= Pname;
3		$Box_1 := ($Obj_Project1, $Obj_Supplier1); $Obj_Project1 → $Obj_Supplier1;
4		$Obj_Project1 → $Pname1; $Obj_Project1 → $Obj_Part1 [_group; ⟨jsp⟩]; {$Pname *like* 'T%'} {CNT_UNIQUE($Obj_Part1)>3}
5		LHS.$Attr_Qty1 — RHS.$Attr_Total_Qty1[SUM];

Explanation:

1. The notation is the full-expanded form of an object class node with its URL and order mark "<". The order mark means that the order of subelement types of Project is important. As to the textual equivalents, all node identifiers, given by machine or user, begin with "$", containing a type prefix and a indexing suffix.
5. The notation is a link between two nodes with aggregation function, which means "the Total_Qty is defined by user and its value is SUM(Qty) under some grouping field".

The EBNF syntax of GLASS (textual equivalents)

The formal GLASS syntax is listed below in style of EBNF. In the syntax, "{…}*" means 0 or more repetitions, "{…}+" means 1 or more repetitions and "[…]" means optional. The character terminals are quoted by a pair of " and ". The category names are highlighted in italic font.

GLASS_query ::= [*Graph_L*] *Graph_R* [*Link*] [*CLW*]

Graph_L ::= "LHS:" {*Node*}* {*Edge*}* "{"{*Predicate*}*"}"

Graph_R ::= "RHS:" {*Node*}* {*Edge*|*Abbr_edge*}*

Link ::= "LINK:" {*Link_edge*}*

CLW ::= "CLW:" {(*Logic_exp* ";")
|(*Math_exp* ";")|*If_clause*}*

Node ::= *Obj_class* | *Attribute* | *Box_group*

Edge ::= *Arrow* | *Dashed_Arrow*

Obj_class ::= *O_node_ID* ":=" (*object_name* | "ANY")
[","*url*] [","*Order_mark*] ";"

Attribute ::= *A_node_ID* ":=" (*attr_name* | "ANY" |
XML_type) ";"

Box_group ::= *B_node_ID* ":=" "("
(*O_node_ID* | *A_node_ID* | *B_node_ID*) {","
(*O_node_ID* | *A_node_ID* | *B_node_ID*)}* ");"

Arrow ::= (*O_node_ID* | *B_node_ID*) " → "
(*O_node_ID* | *A_node_ID*) [*Edge_label*]
[":" *condition_id* ":"] ";"

Dashed_arrow ::= *A_node_ID* " ⇢ " (*A_node_ID* | *O_node_ID*)
[*Edge_label*] [":" *condition_id* ":"] ";"

Abbr_edge ::= *O_node_ID* " → " [*O_node_ID* | *A_node_ID*]
[*Wild_card*] ";"

Link_edge ::= ("LHS." *O_node_ID* " — " "RHS."
O_node_ID [*Agg_function*]) ";"
| ("LHS." *O_node_ID* " — " "RHS."
A_node_ID [*Agg_function*]) ";"
| ("LHS." *A_node_ID* " — " "RHS."
A_node_ID [*Agg_function*]) ";"

Edge_label ::= "[" ["_group;"]["ASC"|"DESC"]
{*Relationship_exp*}* "]"

Relationship_exp ::= "⟨" [*relationship_type_name* ","] *degree*
[","*child_participation*
","*parent_participation*] [","*Order_mark*] "⟩"
| "⟨"*relationship_type_name* "[" *object_name*
{"," *object_name*}* "]" "⟩"

Logic_exp ::= ["FORALL"|"EXIST"] *condition_id*
| "NOT" *Logic_exp*
| *Logic_exp* ("AND"|"OR") *Logic_exp*

If_clause ::= "{IF(" *Logic_exp* ")THEN EXTRACT"
(*O_node_ID* | *A_node_ID*)

Agg_function ::= ["AVG" | "CNT" | "SUM" | "MIN" | "MAX"]
["UNIQUE"]

Wild_card ::= "+" | "*"

XML_type ::= ("element" | "attribute")

Predicate ::= (*O_node_ID* | *A_node_ID*) *Comp_op Constant*";"
|*Agg_function* "(" (*O_node_ID* | *A_node_ID*)
")" *Comp_op Constant*";"

Comp_op ::= "<"|"<="|"="|">="|">"|"<>"|"like"

Order_mark ::= "<"

Constant ::= nil
| integer_literal
| real_literal
| string_literal
| *Boolean_constant*

Bool_constant ::= "TRUE" | "FALSE"

To save the space, we will not list the mathematic expression (*Math_exp*); and the following categories,

object_name	*relationship_type_name*
attr_name	*degree*
O_node_ID	*child_participation*
A_node_ID	*parent_participation*
B_node_ID	*condition_id*