Extracting Object-Oriented Database Schemas from XML DTDs Using Inheritance [⋆]

Tae-Sun Chung, Sangwon Park, Sang-Young Han, and Hyoung-Joo Kim

School of Computer Science and Engineering, Seoul National University
San 56-1, Shillim-dong, Gwanak-gu, Seoul 151-742, KOREA
{tschung,swpark,syhan,hjk}@oopsla.snu.ac.kr

Abstract. As XML has become an emerging standard for information exchange on the World Wide Web, it has gained attention in database communities to extract information from XML seen as a database model. Recently, many researchers have addressed the problem of storing XML data and processing XML queries using traditional database engines. Here, most of them have used relational database systems, while we show in this paper that object-oriented database systems can be another solution. Our technique generates an OODB schema from DTDs and processes XML queries. In particular, we show that the semi-structural part of XML data can be represented by 'inheritance' and that it can be used to improve query processing.

1 Introduction

Recently, as XML[2] has emerged as a standard for information exchange on the World Wide Web, it has gained attention in database communities to extract information from XML seen as a database model. As XML data is self-describing, we can issue queries over XML documents distributed in heterogeneous sources and get the necessary information.

There are two kinds of approaches to query XML documents. One is using special purpose query engines for semistructured data since an XML document can be regarded as an instance of a semistructured data set[3, 8, 10, 13, 15]. The other is using traditional databases such as relational databases or object-oriented databases for storing and querying XML documents[5, 7, 9, 16]. In particular, many approaches using RDBMSs have been proposed. That is, XML data is converted to relational tuples and XML queries are translated to SQL queries. However, to the best of our knowledge, there is no special work on the problem of using OODBMSs to store and query XML data. An exception is work in [5] that processes SGML data using an OODBMS to store and query SGML documents[1].

[⋆] This work was supported by the Brain Korea 21 Project.

[1] Additionally, there is a special purpose XML query processor, Excelon[11] that is based on an OODBMS.

In this paper, we propose a technique that stores and queries XML data using an object-oriented database. Compared with the proposal in [5], our work differs in that we use 'inheritance', a key concept in object-oriented paradigms.

For example, let us assume that the following DTD is given.

```
<!ELEMENT person (name, address, vehicle*,(school|company))>
<!ELEMENT name (firstname?, lastname)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT vehicle (model, company, gear?)>
<!ELEMENT model (#PCDATA)>
<!ELEMENT gear (#PCDATA)>
<!ELEMENT school (name, baseball-team?, person+,url?)>
<!ATTLIST school name CDATA #CDATA REQUIRED>
<!ELEMENT baseball-team (#PCDATA)>
<!ELEMENT url (#PCDATA)>
<!ELEMENT company (name, person+, url?)>
<!ATTLIST company name CDATA #CDATA REQUIRED>
<!ELEMENT alumni (name, year, school)>
<!ATTLIST alumni name CDATA #CDATA REQUIRED>
<!ELEMENT year (#PCDATA)>
```

Fig. 1. An example DTD

Here, the first line says that an element *person* has the *name* and *address* sub-elements, and he or she has zero or more vehicles, and finally, is a student or a company employee. From the DTD declaration for the element *person*, we can classify the element *person* into four groups: 1. ones who have one or more vehicles and work for companies, 2. ones who have no vehicle and work for companies, 3. ones who have one or more vehicles and are students, and 4. ones who have no vehicle and are students. Our technique uses this information in designing object-oriented schema by means of inheritance semantics. In the above example, each group is defined as *Person-1, Person-2, Person-3,* and *Person-4* type classes that inherit the general class *Person*. Here, for example, as *Person-1* is a specialization of *Person*, the inheritance semantics is satisfied.

If we design object-oriented schemas in this way, it can be used for enhancing query evaluation. For example, if a query is related to students having vehicles, a query processor can only traverse extents of *Person-3*.

This paper shows a technique of extracting OODB schemas using inheritance and querying XML documents stored in an OODBMS.

2 Deriving an OO schema from a DTD

In previous work of [5], each class is created for each element definition. Here, the choice operator('|') is modeled by a union type, and the occurrence indicators

('+' or '$*$') are represented by lists. Values (e.g. strings) of XML data are represented by O_2 classes of appropriate content types (e.g., Text) using inheritance. Figure 2 shows an object-oriented schema for the DTD in Figure 1.

```
class Person public type tuple(name:Name, address:Address,
       vehicle:list(Vehicle),union(school:School,company:Company))
class Name public type tuple(firstname:Firstname,lastname:Lastname)
class Firstname inherit Text
class Lastname inherit Text
...
```

Fig. 2. An OODB schema

However, the technique has several problems as follows.

- Since each element definition creates one class, several classes are created though the classes can be inlined into one class. For example, in Figure 2, the classes *Name, Firstname,* and *Lastname* can be inlined into the class *Person*.
- The technique does not use inheritance in designing classes. For example, if we design the *Person* class in the real world, we create a class *Person* as a base class and create classes *Student* and *Employee* that are subclasses of the class *Person*. In this case, query processing can be improved. For example, if a query is only targeted to the class *Student*, a query processor can only traverse objects of the class *Student*. It need not traverse all of the *Person* type classes.
- For the occurrence indicator ('$*$') or optional indicator ('?'), if an object has no value at the corresponding field, the field should be set to null. This has drawbacks in memory efficiency.
- In the technique, the choice operator ('|') is modeled by a union type. However, since the ODMG[4] model, which is a standard for object database management systems, does not support union of types, it can not be applied to ODMG-compliant object-oriented databases directly.

So, our approach solves the above problems in the following ways:

- By applying an inlining technique of relational databases, we inline as many descendants of an element as possible into a single class(Section 2.1).
- After classifying DTD elements, we reconstruct classes using inheritance(Section 2.2).

2.1 Class inlining technique

We adopt the inlining technique of relational databases proposed in [16]. The technique in [16] inlines as many descendants of an element as possible to a single relation.

Compared with the work in [16], our technique has two different points. First, as the traditional relational databases do not support set-valued attributes, when an element has a sub-element with '+' or '∗' expression, the sub-element is made into a separate relation and the relationship between the element and the sub-element is represented by introducing a foreign key. For example, in Figure 3, a relation for "vehicle" is created and links from vehicles to persons are created by using foreign keys. In an object-oriented model, as an occurrence indicator('∗' or '+') can be represented by lists, we don't have to introduce a foreign key manually. In the above example, the class *Person* can be represented by having a set-valued attribute *vehicle*.

Second, in relational models, to represent relationships between relations, join attributes should be created manually. However, in object-oriented models, as relationships between classes can be represented by direct pointers, manual join attributes don't have to be created. For instance, in Figure 3, when the relations for "alumni" and "school" are created in relational models, the relation *School* has a foreign key *parentId* that joins schools with alumni. In object-oriented models, the class *Alumni* has an attribute *school* that has object identities of the class *School*.

When given a DTD, the class inlining technique creates an object-oriented schema as follows. First, Figure 3 shows a DTD graph for the DTD in Figure 1. A DTD graph introduced in [16], represents the structure of a DTD.

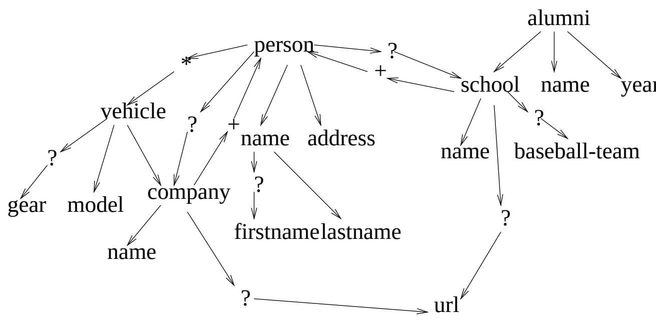


Fig. 3. A DTD graph

Next, we decide what classes to create from the DTD graph by the following rules.

1. Classes are created for element nodes that have an in-degree of zero. Otherwise, the element can not be reached. For example, the class for "alumni" is created, because the element node *alumni* has an in-degree of zero.
2. Elements below a '∗' or a '+' node are made into separate classes. This is necessary for classes that have set-valued attributes. For instance, the element node *vehicle* that is below a '∗' node is made into a separate relation. The class *Person* will have a set-valued attribute *vehicle*.
3. Nodes with an in-degree of one are inlined. For example, in Figure 3, nodes *gear* or *model* are inlined as they have an in-degree of one.
4. Among mutually recursive elements all having in-degree one, one of them is made into a separate relation.

5. Element nodes having an in-degree greater than one are made into separate relations.

If we apply these five rules to the DTD graph in Figure 3, the classes *Person, Vehicle, Company, School, Alumni*, and *Url* are created. Once we decide which classes are created, we construct an object-oriented schema. In the DTD graph, if X is an element node that is made into a separate class, it inlines all the nodes Y that are reachable from it such that there is no node that is made into a separate class in the path from X to Y. An object-oriented schema is created for the DTD graph in Figure 3 as follows.

```
class Person public type tuple(name.firstname:string,name.lastname:string,
       address:string,vehicle:list(Vehicle),school:School,company:Company)
class School public type tuple(name:string,baseball-team:string,
       person:list(Person),url:Url)
class Alumni public type tuple(name:string, year:String,school:School)
class Company public type tuple(name:string,person:list(Person),url:Url)
class Url inherit Text
class Vehicle public type tuple(model:string,company:Company,gear:string)
```

Fig. 4. An OODB schema

2.2 Designing a schema using inheritance

XML data having irregular schema can be represented by inheritance, because if we design the structural part of an element as a superclass and the semi-structural part of it as subclasses, the generalization relationship between the superclass and the subclasses is satisfied.

DTD Automata First, for each element that is made into a separate class, we abstract a DTD as a set of $(n : P)$ pairs. Here, let N be a set of element names, $n \in N$, and P is either a regular expression over N or PCDATA which denotes a character string.

For an element e and the corresponding DTD declaration $(n : P)$, the regular expression P can be divided into five categories as follows. If r, r_1, and r_2 are regular expressions that DTDs represent, $L(r)$, $L(r_1)$, and $L(r_2)$ are the languages that can be described by the regular expressions.

1. case $r = r_1, r_2$: The languages that r denotes are the concatenation of $L(r_1)$ and $L(r_2)$.
2. case $r = r_1 | r_2$: $L(r)$ is the union of $L(r_1)$ and $L(r_2)$.
3. case $r = r_1+$: This represents more than one repetition of the same structure.
4. case $r = r_1*$: This is the same as case 3 except that it permits zero repetitions of the same structure.
5. case $r = r_1?$: This represents zero or one occurrence of the same structure.

Among these five categories, cases 2,4, and 5 result in subclasses, while case 1 and 3 don't result in subclasses. This is because in cases 1 and 3, XML data that conforms to the DTD has attributes of the same form. The reason case 4 becomes information is because whether or not an element has an attribute can be represented by specialization. On the other hand, whether an element has one attribute or more than one can not be explained by specialization.

So, we define the following relaxed regular expression to extract only the necessary information in classifying elements.

Definition 1 (Relaxed Regular Expression). *A relaxed regular expression is constructed from a given regular expression as follows.*

1. $r_1, r_2 \Rightarrow \ r_1, r_2$
2. $r_1|r_2 \Rightarrow \ r_1|r_2$
3. $r+ \Rightarrow \ r$
4. $r* \Rightarrow \ r + |\perp \Rightarrow r|\perp$ *(by rule 3)*
5. $r? \Rightarrow \ r|\perp$

Example 1. In Figure 1, the DTD declaration for the element *person* is abstracted to (person: (name, address, vehicle*, (school|company))), and we get $(person : \ (name, address, (vehicle|\perp), \ (school|company)))$ after applying the relaxed regular expression.

DTD automata are constructed in the following ways. Let $(n_i : \ P_i^{'})$ be an expression which is obtained by applying relaxed regular expressions to each DTD declaration $(n_i : \ P_i)$. We construct automation A_i by Algorithm 1 with a new regular expression $n_i P_i^{'}$[2].

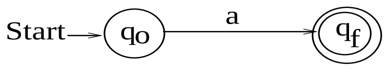


Fig. 5. $r = a$

Theorem 1. *There always exists an automaton M constructed by Algorithm 1 for the input regular expression r, and if $L(M)$ is the language accepted by M, and $L(r)$ is the language which is describable by the regular expression r , then $L(M) = L(r)$.*

We omit the proof for lack of space.

Example 2. Figure 6 shows an automaton constructed by Algorithm 1 for the element *person* after applying the relaxed regular expression, i.e. $(person : \ (name, address, (vehicle|\perp), (school|company)))$.

[2] In this paper, we occasionally omit the concatenation operator, that is, $n_i P_i^{'} = n_i, P_i^{'}$.

Algorithm 1 The construction of DTD automata

```
1: **Input:** A relaxed regular expression $r$
2: **Output:** An automaton $M$
3: **procedure** Make_DTD_Automata(regular expression r)
4: **if** $r = a$ $(a \in \sum)$ **then**
5:    Construct an automaton $M$ as shown in Figure 5;
6:    return $M$;
7: **else if** $r = r_1|r_2$ **then**
8:    $M_1 = (Q_1, \sum_1, \delta_1, q_1, F_1) \leftarrow$ Make_DTD_Automata($r_1$);
9:    $M_2 = (Q_2, \sum_2, \delta_2, q_2, F_2) \leftarrow$ Make_DTD_Automata($r_2$);
10:   Construct the new automaton $M = (Q_1 - \{q_1\} \cup Q_2 - \{q_2\}, \sum_1 \cup \sum_2, \delta, [q_1, q_2], F_1 \cup F_2)$ from the automata $M_1$ and $M_2$, where $\delta$ is defined by

      1. $\delta(q,a) = \delta_1(q,a)$ for $q \in Q_1 - \{q_1\}$ and $a \in \sum_1$,
      2. $\delta(q,a) = \delta_2(q,a)$ for $q \in Q_2 - \{q_2\}$ and $a \in \sum_2$,
      3. $\delta([q_1,q_2],a) = \delta_1(q_1,a)$ where $a \in \sum_1$,
      4. $\delta([q_1,q_2],a) = \delta_2(q_2,a)$ where $a \in \sum_2$;

11: **else** { $r = r_1, r_2$ }
12:   $M_1 = (Q_1, \sum_1, \delta_1, q_1, F_1) \leftarrow$ Make_DTD_Automata($r_1$);
13:   $M_2 = (Q_2, \sum_2, \delta_2, q_2, F_2) \leftarrow$ Make_DTD_Automata($r_2$);
14:   Let the final states $F_1$ of $M_1$ be states $f_1, f_2, ..., f_m$ $(m \geq 1)$. Construct the new automaton $M = (Q_1 - F_1 \cup Q_2 - \{q_2\} \cup \{[f_1,q_2],[f_2,q_2],...,[f_m,q_2]\}, \sum_1 \cup \sum_2, \delta, q_1, F_2)$ from the automata $M_1$ and $M_2$, where $\delta$ is defined by

      1. $\delta(q,a) = \delta_1(q,a)$ for $q \in Q_1 - F_1$, $\delta_1(q,a) \neq f_k$ (where $1 \leq k \leq m$), and $a \in \sum_1$,
      2. $\delta(q,a) = \delta_2(q,a)$ for $q \in Q_2 - q_2$ and $a \in \sum_2$,
      3. $\delta([f_k,q_2],a) = \delta_2(q_2,a)$ for all k(where $k = 1, 2, ..., m$ ) and $a \in \sum_2$,
      4. $\delta(q_f,a) = [f_k,q_2]$ for all $q_f$ which satisfies $\delta_1(q_f,a) = f_k$(where $1 \leq k \leq m$) and $a \in \sum_1$;

15: **end if**
16: return $M$
```

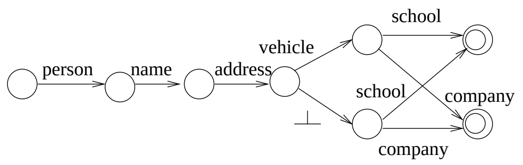
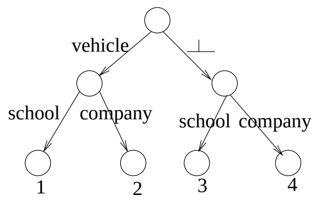


Fig. 6. A DTD automaton

Classification of DTD elements using DTD automata As the DTD automata are constructed from relaxed regular expressions, they contain information only about concatenations and unions. Here, the diverging points in automata become those of classifying DTD elements. So, by recording the labels at diverging points we can classify the DTD elements.

Figure 7 shows a classification tree from the DTD automaton of the element *person* in Figure 6, and the corresponding classification table. Here, the DTD element *person* is divided into 4 groups according to its label sets, namely {*vehicle, school*}, {*vehicle, company*}, {*school*}, and {*company*}.



1	{vehicle, school}
2	{vehicle, company}
3	{school}
4	{company}

Fig. 7. A classification tree and a classification table

The fact that each element is classified into several groups stems from the flexibility of XML data, and it becomes a hint to a query processor. This is because if a query is targeted to certain groups only, a query processor can only traverse the groups targeted.

We created a superclass from the structural part of an element and several subclasses that inherit the class. The subclasses are created from the semi-structural part of the element. For example, for the element *person*, the class *Person* has attributes *name* and *address*. Further, subclasses are created that inherit it and has attributes {*vehicle, school*}, {*vehicle, company*}, {*school*}, and {*company*}. In this way, an object-oriented schema is created as follows.

```
class Person public type tuple(name.firstname:string,name.lastname:string,
        address:string)
class Person1 inherit Person type tuple(vehicle:list(Vehicle),
        school:School)
class Person2 inherit Person type tuple(vehicle:list(Vehicle),
        company:Company)
class Person3 inherit Person type tuple(school:School)
class Person4 inherit Person type tuple(company:Company)
...
```

Fig. 8. An OODB schema

3 Query Language

Several query languages including XML-QL[6], UnQL[3], Lorel[1], and XQL(from Microsoft) have been proposed to query semistructured data. The semistructured query languages are based on the following regular path expression that expands the path expression[12] of object-oriented database query languages.

Definition 2. *(Regular Path Expression) A regular path expression is in the form of H.P, where*

1. *H is an object name or a variable denoting an object,*
2. *P is a regular expression over labels in an DTD, i.e.* $P = label|(P|P)|(P.P)|P*$.

We should solve two problems to convert semistructured query languages based on regular path expressions to object-oriented query languages. First, as object-oriented query languages do not support processing the alternation operator('|') that stems from $(P|P)$ in the regular path expression, we should make a routine that processes it. We adapt the alternation elimination technique proposed in [14]. For example, the regular path expression $person.(school|company).name$ is converted to $(person.school.name) \cup (person.company.name)$.

Second, we should process arbitrary complex queries, i.e. nested recursive queries that stems from $(P*)$ in the regular path expression. In [14], the authors suggest a technique that replaces all regular expression operators with possible path instantiations using DataGuides[10] that are structural summaries of databases. We suggest a similar technique using DTDs. First, we define a simple regular path expression as follows.

Definition 3. *(Simple Regular Path Expression) A simple regular path expression is a sequence* $H.p_1.p_2.....p_n$ *where*

1. *H is an object name or a variable denoting an object,*
2. p_i *(where* $1 \leq i \leq n$*) is a label in an DTD or wild-card "$*$" which denotes any sequence of labels.*

Compared to the regular path expression, it is simple, but can process almost all XML queries. We show that queries that have simple regular path expressions can be converted to object-oriented queries. Our technique can be generalized to regular path expressions.

3.1 Translating simple regular path expressions without the '$*$' operator

In this section, we deal with simple regular path expressions without the '$*$' operator. In this case, we can convert it to object-oriented database query languages easily. For example, consider the following Lorel-like semistructured query.

```
select X.name.firstname, X.name.lastname
from person X, X.vehicle Y
where X.address = "Seoul", Y.model = "EF-Sonata", Y.gear="auto"
```

The query asks for the first and last name of the person who has a vehicle "EF-Sonata" with an automatic transmission. The query is converted to the following object-oriented database query languages.

```
select tuple(f:p."name.firstname",l:p."name.lastname")
from p in Person,y in p.Vehicle
where p.address = "Seoul", y.model = "EF-Sonata", y.gear="auto"
```

When the query is processed, the number of target classes can be reduced. That is, in the database schema in Figure 8, as the variable p bound to the class *Person* has an attribute *vehicle*, only the instances of the class *Person1, Person2* are traversed.

3.2 Converting simple regular path expressions with the '$*$' operator

Queries having the expression '$*$' that denotes any sequence of paths are frequently used in XML queries by those who do not know database schema. For example, consider the following query.

```
select u
from  person.*.url u
```

The query requires all urls that are reachable by the paths that first, have the edge *person* followed by any sequence of arbitrary edges, and next, have the edge *url*. As object-oriented database query languages do not support this kind of queries directly, we convert the path expression with the '$*$' operator to possible path instantiations using the DTD graph in Section 2.1. Here, operators in the DTD graph are excluded.

The '$*$' expression in the above query is replaced with the paths *school, company,* and *vehicle.company*. Thus, the query is converted to the following Lorel-like query.

```
select u
from  (person.school.url|person.company.url|
       person.vehicle.company.url) u
```

Next, The query is converted to the following object-oriented query.

```
select u
from  p in Person,s in p.school,c in p.company,v in p.vehicle
      v2 in v.company, u in (s.url,c.url,v2.url)
```

4 Conclusion

In this paper, we showed that object-oriented databases can be another solution for storing and querying XML data. We propose a technique that creates object-oriented schemas from DTDs. In particular, we solve the problem of impedance mismatch that stems from the flexibility of XML data by using inheritance. That is, by representing the semi-structural part of XML data using inheritance, our technique solves the null value problem and enhances query processing.

We showed that XML queries composed of simple regular path expressions are converted to object-oriented database query languages and the results of queries to XML data. Here, we suggest a technique that removes the '$*$' expression by using DTDs.

References

1. S. Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet Wiener. The lorel query language for semistructured data. *International Journal on Digital Libraries*, 1996.
2. T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible markup language (XML) 1.0. Technical report, W3C Recommendation, 1998.
3. Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 1996.
4. R.G.G. Cattell. *The object database standard: ODMG-93.* Morgan Kaufmann Publishers, 1994.
5. V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From Structured Documents to Novel Query Facilities. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 1994.
6. A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. Query language for XML. In *Proceedings of Eighth International World Wide Web Conference*, 1999.
7. Alin Deutsch, Mari Fernandez, and Dan Suciu. Storing semistructed data with STORED. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 1999.
8. Mary Fernandez and Dan Suciu. Optimizing regular path expressions using graph schemas. In *IEEE International Conference on Data Engineering*, 1998.
9. Daniela Florescu and Donald Kossmann. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 1999.
10. Roy Goldman and Jennifer Widom. DataGuides: enabling query formulation and optimization in semistructured databases. In *Proceedings of the Conference on Very Large Data Bases*, 1997.
11. http://www.odi.com/excelon. 2000.
12. M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 1992.
13. J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database management system for semistructured data. *SIGMOD Record*, 1997.
14. J. McHugh and J. Widom. Compile-Time Path Expansion in Lore. In *Proceedings the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, 1999.
15. Tova Milo and Dan Suciu. Index structures for path expressions. In *Proceedings of the International Conference on Database Theory*, 1999.
16. Jayavel Shanmugasundaram, H. Gang, Kristin Tufte, Chun Zhang, David DeWitt, and Jeffrey F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of the Conference on Very Large Data Bases*, 1999.