IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661,p-ISSN: 2278-8727, Volume 18, Issue 5, Ver. I (Sept - Oct. 2016), PP 40-52
www.iosrjournals.org

Transforming XML into Object-Relational Schema

Mustapha Machkour[1], Karim Afdel[2]
[1,2]*(Department of Computer Sciences, Faculty of Sciences/ Agadir, Morocco)*

Abstract: *Recently, there is a vast increase in the use of XML for describing and exchanging data. To manipulate efficiently these data, it would be wise to use database systems which represent an appropriate tool to store and manage data. To have this purpose, we need to transform XML schema into database models such as relational and Object-Relational (OR). The aim of this work is to present a methodology that transforms an XML schema into the OR model. To be automatic, the steps of this transformation are formalized by a mapping function defined from XML into the object-relational model. Among the advantages of this transformation is that it preserves the structure and constraints defined in XML schema which enables to retrieve the initial XML structure from its converted, OR schema.*

Keywords: *Database model, Mapping, Object-Relational model, Transformation, XML.*

I. Introduction

Extensible Markup Language (XML) [4] has emerged as the standard for representing and exchanging data. Many applications adopt XML for exchanging or publishing information. As examples, we cite:

- Web sites started publishing information in the form of XML feeds e.g. resource description framework (RDF);
- XML Web services became an integral part of enterprise applications;
- A large number of applications are being written that make use of XML web services such as Google APIs, Amazon Web Services, etc.
- Many web sites are turning to AJAX (Asynchronous Java Script and XML) programming, where data is exchanged in XML format.

To enable relational database systems [6,7] to access these large amounts of XML data, conversion methodologies [5,10-12] from XML into relational schema have been developed. But, due to the limits of the relational systems, the object-relational (OR) model [11, 12] has been introduced as new database model. It combines relational and object-oriented capabilities [11, 12]. These concepts include class, object, object identifier, inheritance, etc. More and more database systems started support this model such as Oracle database [14,15], PostGres [16], and Informix Dynamic Server(IDS) [1]. They became object-relational database management systems.

To pursue these changes, we need methodologies for transforming XML data into the object-relational model. The aim of this paper is to present a methodology that transforms an XML data based on a document type definition (DTD) into an OR data. This translation is done with preservation of data structure, without any alteration such as simplification, grouping, and flattening, which allows constructing the XML schema from the OR schema.

II. Related Work

There is a lot of research related to the transformation between computer models. These transformations imply different levels of abstraction such as conceptual and logic. At the conceptual level, there are as examples, the transformation between Entity-Relationship and Object-Oriented (OO) models [14], and between UML and XML models [15-18]. At the logical level, we consider the translation between Network and Relational models [11, 12]. There are also translations that involve more than one level such as transformation from UML class into object-relational schemas.

We are interested in this article to XML transformation. Many papers talk about transformations between XML and relational model [5]. However, the relational model does not have enough capability to model recent real-world applications (scientific, web and multimedia) and needs to be enriched with OO concepts. This has led to a new era of the object-relational model.

This model adds the powerful concepts of OO model, such as complex data, methods, and inheritance, to the simplicity of the relational model and the great experience it has gained since the 70s. It offers a better representation of the real world objects than the tabular form used in the relational model.

To be aligned with these extensions, we present in this work, a methodology that translates an XML-based DTD into an OR schema based on the structured query language (SQL):2003 standards, named also SQL4

[6-9,13]. This methodology is based on a mapping function defined between XML and OR models. This function allows formalization of the translation and makes it automatic.

The remainder of this paper is organized as follows. In section 3 we present concepts and notations used in this paper. Section 4 describes a mapping function that makes formal the correspondences between XML and OR concepts. Section 5 details the steps of the algorithm of conversion. Section 6 presents an example of mapping. Finally, in section 6, we conclude the paper. I also added in the appendix the scripts that represent the physical OR schema.

III. Concepts And Notations

This section introduces the concepts and notations used throughout the paper. These notations help us to formalize the steps of the transformation and make it automatic. The concepts of the OR model include [8,9,13]:

User defined type (UDT): a fundamental concept in the object-relational model. it's used to create complex-structured objects;

- reference type: defines the object identifier (OID);
- row type: to create a structured attribute without using UDT;
- array type (AT) : to define limited collection;
- multiset type (MT): to define unlimited collection;
- object table: used to create table based on UDT and stores objects;
- inheritance, for making inheritance relation between types.

Let be (Attri), $1\leq i \leq n$, the attributes of a UDT E. The definition of Attri comprises its name, its type, and its constraints which represent a list of constraints on its permissible values and include null, not null, unique, check and foreign key constraint. This definition is represented by the following notation:

Fig.1. <Attri; Type; Constraints>

Fig.2. Definition of the attribute Attri

For a UDT E, regarded as a list of its attributes, we use the below notation:

E ((<Attri; Type; Constraints>)$_{1\leq i \leq n}$)

Fig.3. Definition of the UDT E

The rest of this section explains the notations used for the XML schema. We employ the below DTD to illustrate our notations.

<!ELEMENT volume (issue+)>
<!ELEMENT issue (paper+)>
<!ELEMENT paper (title, author, cite?)>
<!ATTLIST paper id ID #REQUIRED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (fn, ln)>
<!ELEMENT fn (#PCDATA)>
<!ELEMENT ln (#PCDATA)>
<!ELEMENT cite (paper)>*

Listing.1. Example of DTD

In general, if A is an XML object such as an attribute, a list of attributes, an element or its content model [4] ..., $\underline{A}$ (underlined A) gives its definition. Let be Attr an attribute of an XML element E. Its definition provides the following things:

1) Attr: the name of the attribute,
2) type: the data type of the attribute, or a list of its possible values, as defined in the XML model [4] and
3) Description: the value of this component, as defined in XML model [4], is specified in the following figure, using a context-free grammar notation [2]:

Description::= #REQUIRED | #IMPLIED | #FIXED value | value.

Fig.4. Definition of an attribute Description

We group these things using the below notation (here, the symbol::= denotes a definition):

$\underline{\text{Attr}}$::=<Attr; type; Description>

Fig.5. Definition of an attribute of an XML element

For example, in **Error! Reference source not found.**, the attribute "id" has the definition: id=<id; ID; #REQUIRED>.

Now, let be E an XML element, with D its content model [4] and Attrs its list of attributes [4], i.e. Attrs=(Attri)$_{1\leq i \leq n}$ (we suppose that E has n attributes). The definition of E is provided by:

E::=<E; Attrs; D>

Fig.6. Definition of the XML element E

In this notation, the definition of the attribute list, Attrs (underlined Attrs), is a list of attribute definitions (see 0). This list is resulting by the distribution of '_' between attributes Attri ($1\leq i\leq n$). Therefore, we have the expression: Attrs=$(\text{Attri})_{1\leq i \leq n}$.

As example, in listing 1, the definition of paper is given by

paper=<paper; id; title, author>

IV. A Mapping Between XML And Object-Relational Models

Now, we define the mapping function φ which formalizes the conversion from XML model into the Object-relational model. Each element E in XML model maps to a UDT, named also E, representing its image in the Object-relational model. So we have

φ : XML model → OR model, and

E→ φ(E) with φ(E) is a UDT.

The expression of the XML element E, as seen in 0, is

E=<E; Attrs; D>. If we apply φ, in this formula, on both sides of the equality, we get

φ(E)=φ(<E; Attrs; D>).

By definition, we have

φ(<E; Attrs; D>) = E (φ(Attrs) U φ(D)),

and, by transitivity, we get the formula:

φ(E) = E (φ(Attrs) U φ(D))

Fig.7. Definition of a UDT E

This means that the list of attributes of the UDT E is obtained by the union of the image (by φ) of Attrs and D. So, to find the structure of the UDT E, we need to calculate φ(Attrs) and φ(D). First, we look for the expression of φ(Attrs).

4.1. Calculation of □(Attrs)

φ(Attrs) is a list of attribute definitions of a UDT. This list is obtained by the following algorithm:

```
Algorithm ListAttributes(Attrs) return φ(Attrs);
begin
if Attrs=empty then /*the XML element has no attributes*/
        return emptyString;
elseif Attrs= (Attr1, Attr2,..., Attrn) then
            return (φ (Attr1), φ(Attr2) ...φ(Attrn));
end if;
end;
```

Listing.2. Calculation of φ(Attrs)

This algorithm needs the valuation of φ(Attri) ($1\leq i \leq n$). If we replace Attri by its value defined in 0, we get

φ(Attri)= φ(<Attri; typeOrValues; Description>).

The value of φ(<Attri; typeOrValues; Description>) is defined by:

<Attri; φ(typeOrValues) MINUS Constraints; φ(Description) PLUS Constraints>.

This expression has three parts: the first one represents the name of the attribute of UDT; the second defines its type and the third represents its constraints.

Finally, the value of φ(<Attri>) is given by

φ(<Attri>)=<Attri; φ(typeOrValues) MINUS Constraints; φ(Description) PLUS Constraints> .

Fig.8. Definition of the UDT attribute Attri

This requires the calculation of φ(typeOrValues), φ(Description) and "Constraints".

4.1.1. Calculation of □(typeOrValues)

The value of the expression φ(typeOrValues) defines the type and constraints of the attribute Attri, in object-relational model. This value is given using the Table 1**Error! Reference source not found.**. Constraints are defined using regular expressions [3].

Table 1. Calculation of φ(typeOrValues)

typeOrValues (in XML)	φ (type)(in OR model)	
	Type	Constraints
ID	varchar(n)	(Letter|_)(Letter|_|Digit|:|.|-)*, UC: Unique Constraint
CDATA	varchar(n)	No constraint
IDREF	varchar(n)	(Letter|_)(Letter|_|Digit|:|.|-)*, FKC:Foreign Key Constraint
IDREFS	array(p) or multiset of varchar(n)	(Letter|_)(Letter|_|Digit|:|.|-)*, FKC: Foreign Key Constraint

NMTOKEN	varchar(n)	(Letter|_)(Letter|_|Digit|:|.|-)*
NMTOKENS	array(p) or multiset of varchar(n)	(Letter|_)(Letter|_|Digit|:|.|-)*
Enumerated Attribute list	varchar(n)	(Letter|_)(Letter|_|Digit|:|.|-)*, ELC: Enumerated List Constraint

Below, we give the meaning of the columns Type and Constraints. First, in the column Type we have:
1) varchar(n): represents string type in database systems. n is the size of type;
2) array(p): is an array type. p is the size of the collection;
3) multiset: is a multiset type .

Second, in the constraints column: we have patterns that values of the attribute in OR model, must check in order to keep the semantic aspect of the XML attribute. These patterns are based on the regular definitions [2]: Letter and Digit. Letter represents the expression [A..Za..z] and Digit represents the expression [0..9]. In the constraints column, we find also:
1) Foreign key Constraint (FKC): a constraint that represents the usual referential integrity;
2) Unique Constraint (UC): a constraint to have distinct values for the attribute and
3) Enumerated List Constraint (ELC): a constraint with a list of values corresponding to the enumerated values that specifies the content model of XML elements attributes. We use LC to denote the constraint defined by the regular expression: (Letter|_)(Letter|_|Digit|:|.|-)*.

4.1.2. Calculation of □(Description)

The value of φ(<u>Attrs</u>) requires also the value of φ(Description) (see 0). The value of φ(Description) is a usual database constraints. It is obtained by an algorithm based on the following table:

Table 2. Calculation of φ(Description)

Description	φ(Description)
#REQUIRED	not null
#IMPLIED	null
#FIXED Value	not null, default Value
Value	default Value

To understand how φ maps attributes, we consider the example in Listing.1. The element paper has one attributes id. If we apply φ to this element we get:
φ(<u>paper</u>)=paper(φ (<u>id</u>),…).

where, "paper" (at the right of "=" symbol) is a UDT with φ(<u>id</u>)=φ(<id; ID; REQUIRED>) as attribute.

Calculus of φ(<u>id</u>) :
According to formula in 0, we have
φ (<u>id</u>)=<id; φ(ID) – (LC+UC); φ(#REQUIRED) + (LC+UC)>.

Since φ(ID)=varchar + (LC+UC) and
φ(#REQUIRED)=not null, then φ(<u>id</u>) becomes

φ(<u>id</u>)= <id; varchar; not null+ LC+UC)>.

This means that φ(<u>id</u>) is an attribute of the UDT "paper" with these characteristics :
1) id: is its name;
2) varchar is its type; and
3) not null, LC and UC are its constraints.

We can recapitulate these steps in the following algorithm:

```
Algorithm UDT_attribute_from_XML_attribute;
Input Attri: an XML attribute;
Output φ(Attri) : a UDT attribute;
begin
Calculate φ(type);
Calculate φ(Description);
Return <Attri; φ(type) minus Constraints; φ(Description) plus  Constraints>;
end;
```

Listing.3. Algorithm for obtaining a UDT attribute from an XML attribute.

4.2. Calculating φ(<u>D</u>)

The expression of φ(<u>E</u>) (see 0) is given by:

φ(E)=E (φ(Attrs) U φ(D)).
φ(Attrs) is already treated and now we treat φ(D).
We have in 0, E=<E; Attrs; D> where D is the content model of the XML element E. D can be *EMPTY*, *ANY* or List of symbols between *"<! ELEMENT ElementName(" and ")>"*.
To see examples of "D", consider, again, the example in Listing.1. The value of D for "issue" is "paper+". So we can write:

issue::=<issue; _; paper+>.

The value of D for the element "paper" is "title, author". This allows us to write:
paper::=<paper; id ; title, author>.
Also, we can write:
title::=<title;_;#PCDATA>, and
author::=<author;_;#PCDATA>, since the value of D for title and author is #PCDATA.
To render formal the valuation of φ(D), we describe the content model by the below grammar which we name G (E is an element of D).

a) E::=ANY,
b) E::=EMPTY,
c) E :: = E, E for the sequence,
d) E :: = E + E for the alternative,
e) E :: = {E} for the Kleene closure (replace *),
f) E :: = {E}, E for the transitive closure (replace +),
g) E :: = [E] for an optional value(replace ?),
h) E :: = #PCDATA for a simple type.

Fig.9. Grammar G : Description of the content model

To this grammar, we associate the following grammar which we note G (G underscored). Recall that E as defined above (see 0) gives the definition of the element E. This grammar is defined as follows:

a) E::=ANY
b) E::=EMPTY
c) E::=E, E
d) E::= E1+E2
e) E::= {E}
f) E::= {E}, E
g) E::= [4]
h) E::=#PCDATA

Fig.10. Elements of the Grammar G.

Let us now define the value of φ for each item of the grammar G. We define a grammar that we call φ(G):

a) φ (E)= φ (ANY)::=generic type such as anydata or anytype.
b) φ (E)= φ (EMPTY)::=Empty string
c) φ (E)= φ (E1, E2)::= φ (E1), φ(E2) E1and E2 are used to distinguish between E at left and right of '='.
d) φ (E)::= (+,φ(E1),φ(E2)). Here, we define a generic type that can hold the object types φ(E1) and φ(E2).
e) φ (E) ::={φ (E)},list of φ(E) with null constraint;
f) φ (E) ::={φ (E)}list of φ(E) with not null constraint;
g) φ(E) ::= [φ (E)], φ (E) with null constraint ;
h) φ (E) ::= φ (#PCDATA).

Fig.11. Elements of the Grammar φ(G)

The value of φ(#PCDATA) is given by :
φ(#PCDATA)::=<value; varchar; _>
Fig.12. Value of φ (# PCDATA)

where "value" is an attribute of the UDT that contains the value of the XML element, varchar represents its type and the "_" symbol means that there is nothing. To understand how the function φ functions, we propose some schema translations from XML into object-relational model.

4.2.1. Examples of conversion

a) Example 1

From Listing.1, the element fn has the definition: *fn =<fn;_;#PCDATA>*,

If we apply the function φ to this element, we obtain
φ(<u>fn</u>)=φ(<fn;;<u>#PCDATA</u>>)
=fn(φ(<u>Attrs</u>) U φ (<u>#PCDATA</u>))
=fn(φ(<u>#PCDATA</u>)), since φ(<u>Attrs</u>) is empty, i.e. there is no attribute for the element fn.
If we replace φ(<u>#PCDATA</u>) with its value using the grammar φ(<u>G</u>), we get for the φ(<u>fn</u>) the expression:
φ(<u>fn</u>)=fn(<value; varchar; _>).
So, "fn" is a UDT with an attribute named value which type is varchar with no constraint. We can do the same for the elements ln and title, and we obtain the expressions:
φ(<u>ln</u>)=ln(<value; varchar; _>) and
φ(<u>title</u>)=title(<value; varchar; _>).

b) Example 2
For a complex example that shows how *φ* works, we take the author element defined at **Error! Reference source not found.**Listing.1. We have the expression
<u>author</u>=<author; <u>_</u>; <u>fn, ln</u>>.
If we apply φ to this formula, we get
φ(<u>author</u>)=author(φ(<u>fn, ln</u>))
=author(φ(<u>fn</u>), φ(<u>ln</u>)).
If we replace φ(<u>fn</u>) and φ(<u>ln</u>) by their values, we obtain:
φ(<u>author</u>)=author(fn(<value; varchar;_>), ln(<value; varchar; _>)).

Therefore, "author" is a UDT with two attributes: fn and ln. Each of them is a UDT with an attribute named value. In general the calculation of φ(<u>D</u>) is defined in the following listing:

```
Algorithm UDT_attribute_from_Content_model;
Input: D, a model of content of an XML element E;
Output: φ(D), list of UDT attributes;
begin
loop
        select an arbitrary φ (v) in φ (D) with v different to E;
        If (φ(v) is not in v (to avoid recursion)) then
                Calculate φ(v) using the φ(G) and algorithm in Listing.3;
    end if;
        If (there is no φ(v) in φ (D)) or (each φ(v) in φ (D) is in v
/*case of recursion*/  or φ (v)=φ (E))          then
                        Exit; /*to leave loop*/
        end if;
end loop;
end;
```

Listing.4. Algorithm that calculates UDT attributes from Content model

c) Advanced example of the calculus of the φ(<u>D</u>) (case of recursion)
To illustrate the calculus of the *φ(<u>D</u>)* with recursion, we consider the element "paper" defined at Listing.1.
We have <u>paper</u> ::= <paper; <u>_</u>; <u>D</u>>   where the value of D is *(title, author, cite?).*
Therefore
φ (<u>paper</u>) = paper(φ(<u>Attrs</u>) U *φ(<u>D</u>))* = paper(φ(<u>D</u>)), because φ(<u>Attrs</u>) is empty.

If we replace φ(<u>D</u>) by its value, φ(<u>title, author, [cite]</u>), in last formula ,we obtain
φ(<u>paper</u>) = paper(φ(<u>title, author, [cite]</u>)) = paper(φ(<u>title</u>, <u>author</u>, [<u>cite</u>])),
and then

φ(<u>paper</u>)=paper(φ(<u>title</u>),φ(<u>author</u>),[φ(<u>cite</u>)]).
Fig.13. Intermediate value of φ(paper).
We have already calculated φ(<u>title</u>) and φ(<u>author</u>). Let us find φ(<u>cite</u>).
From the expression <!ELEMENT cite (paper*)>, we can write : <u>cite</u>::= <cite;<u>_</u>;{<u>paper</u>}>
If we apply the function φ to the element <u>cite</u>, we get
φ(<u>cite</u>)= φ(<cite;;{<u>paper</u>}>) =cite(φ ({<u>paper</u>}))= cite({φ (<u>paper</u>)}).

If we replace φ(<u>title</u>), φ(<u>author</u>) and φ(<u>cite</u>), at 0, by their values, we obtain

φ(paper)=paper(title(<value; varchar; _ >),
author (fn(<value; varchar; _>),
 ln(<value; varchar; _>)),
[cite({φ (paper)})]).

Fig.14 Value of φ (paper).

The process stops here because there is no more φ(v) in φ(D) with φ(v) different to φ(paper). In the next section, we present the algorithm that takes the values created by φ and generate an object-relational schema.

V. Algorithm of Translation

The value of φ has the form attr(listOfItems) where attr is an abbreviation of "attribute" and represents any word before the left parenthesis in this value. For example in 0, paper is attr and, title, author and [cite({φ(paper)})] are items. Similarly, in the expression title(<value; varchar; _ >), title is attr and <value; varchar; _ > is an item, and in the expression cite({φ(paper)}), cite is attr and {φ(paper)} is an item. In general, items in listOfItems are of type:

- (…)
- <…>
- […]
- {…}

Our translation algorithm is based on the function createUDTAttribute defined in Listing.5. This function takes values created by φ function and returns UDT attributes. It calls a set of procedures and functions that deal with the different types of items. These procedures will be explained as their presentations. Now, we give the body of createUDTattribute.

```
Algorithm createUDTAttribute (attr(listOfItems)) return UDT Attribute ;
begin
1)  closure_1(attr(listOfItems));//handles items of type {x(...)}
2)  closure_2(attr(listOfItems));//handles items of type {φ (x)}
3)  optional(attr(listOfItems)); //handles items of type [...]
4)  Loop
5)  simpleItems(attr(listOfItems));//handles items of type  < ..>
6)  //case of alternative with named element
7)        namedAlternative (attr(listOfItems));
8)  //case of alternative with unnamed element
9)  unnamedAlternative(attr(listOfItems));
10) recursion (attr(listOfItems)); // handles items of type (...)
11) end loop;
end; /*end of createObjectAttribute*/
```

Listing.5. CreateUDTAttribute Function

In the following subsections, we present the definitions of the procedures and functions called by the function CreateUDTAttribute.

5.1. Processing of items of type "<…>": simple items

The processing of items of type "<…>", i.e. simple items, or items without alternative, recursion…, is done by the function simpleItems described in the following listing.

```
Algorithm simpleItems (attr(listOfItems)) return UDT Attribute;
begin
1)  if each item of listOfItems matches "<...>" then
2)    if the attr type is not yet created then
3)      Create a type named attr, where each of its
        attributes corresponds to each item of listOfItems;
4)    end if;
5)    return the UDT attribute :
      <"attr"; attr; list_of_item_constraint>;
6)  end if;
end; // end of function
```

Listing.6. simpleItems Function

For an example of application of this function, consider the expression title(<value; varchar; _>) (see 0).

Since title contains only items that match " <…>", the call of simpleItems (title(<value; varchar;…>)) creates a UDT named title with one attribute named value and returns a UDT attribute defined by : <"title"; title; _>.

5.2. Processing of items of type e(…) and recursion

Here, we show the definition of the procedure "recursion" that deals with items of type e(…), direct recursion and mutual recursion. The following listing describes this definition.

```
Algorithm recursion (attr(listOfItems))
begin
1)  for each item e(…) in listOfItems loop
2)  if e(…) doesn't contain directly any φ then
3)    replace in attr, e(..) by CreateUDTAttribute(e(..));
4)  elseif e(..) matches e(φ(x)) then
/* case of recursive element */
5)    replace in attr, e(..) by <"e"; ref x;>;
6)  elseif e(..) matches e(…,φ(x),…) then
/* case of elements mutually recursive */
7)      if the UDT x is not yet created then
        create the UDT x as incomplete type;
8)      end if;
9)      replace φ(x) by <"x"; ref x;>;
10)  end if;
11) end loop;
end; // end of recursion
```

Listing.7. recursion procedure

As an application of this procedure, let's find CreateUDTAttribute (author(…)) .
We have
φ(author)=author(*fn(<value;varchar;_ >),*
ln(<value; varchar; _ >)).
In this expression, author has items (fn and ln) that match "e(…)".
In this case, to have CreateUDTAttribute (author(…)), we use recursion(author(…)) and we get
<"fn";fn; _> (obtained by CreateObjectAttribute(fn(…)))
and
<"ln"; ln; _> (obtained by CreateObjectAttribute(fn(…))).
After this substitution, author becomes
author(<"fn";fn; _>, <"ln";ln; _>).
We can now apply simpleItems (which handles <…> items) to
author(<"fn";fn;_>,<"ln";ln;_>) and we get an attribute defined by <"author", author, _>.

5.3. Processing of closure : items of type {x(...)}

The body of the procedure that deals with closure {x(..)}, is given below. It deletes all closure of this type.

```
Algorithm closure_1(attr(listOfItems))
y : UDT Attribute; //y is a variable to store a UDT attribute;
begin
1)  for each {x(..)} in listOfItems loop //x is an XML element
2)  y = createUDTAttribute (x(...)) ;
3)  create a row type named xs (name of x concatenated to 's') based on object type y;
4)  replace {x(..)} in attr by <"xs";xs; constraints_on_x>;
5)  end loop;
end;// end of procedure
```

Listing.8. closure_1 procedure

Consider the element listAuthors defined by
<!ELEMENT listAuthors (author+)>
<!ELEMENT author (#PCDATA)>
We have the expression
φ (listAuthors)=listAuthors({φ (author)})
= listAuthors ({author(φ (#PCDATA))})
and then, listAuthors ({author(<value; varchar ;_>}).
If we apply closure_1 to ListAuthors, we obtain the expression:

listAuthors (<"authors"; authors; _>) where "authors" is an attribute of type authors. The latter is row type created by the lines 2 and 3 in Listing.8.

5.4. Processing of closure:items of type {(x)}

The goal of this processing is to delete the symbols "{" , "}" and φ. The body of the procedure that does this work is given below.

```
Algorithm closure_2(attr(listOfItems))
begin
1)  for each {φ(x)} in attr loop
2)  if type x is not yet created then
3)  create the UDT x as incomplete;
/* necessary to have recursion*/
4)  end if;
5)  create a type of nested table xs (name of x concatenated to 's') based on the reference of the object type x
    "ref x";
6)  replace in attr {φ (x)} by <"xs";xs;_>;
7)  end loop;
end; //end of procedure
```

Listing.9. closure_2 procedure

As an example of transformation by this procedure, we propose the following expression:

[cite({φ (paper)})] (see 0).

To transform [cite({φ (paper)})], CreateUDTAttribute calls the "closure_2" procedure (see Listing.9) to delete symbols "{" , "}" and φ. Thus this procedure does the following operations:

1) it creates an incomplete UDT named paper;
2) it creates a row type based on "ref paper" named papers; and
3)it replaces {φ (paper)} by <"papers"; papers; _>.

After that, we get the expression:

[cite(<"papers"; papers; _>)].

So, we have eliminated symbols "{" , "}" and φ;

5.5. Processing of optional element:items of type […]

The purpose of the processing of optional element is to eliminate the expression of type [x(…)]. This is done by the following procedure.

```
Algorithm optional(attr(listOfItems))
y : UDT Attribute; //y is a variable to store a UDT attribute;
begin
1)      for each [x(..)] in attr loop
2)  y = CreateUDTAttribute(x(...)) ;
3)  add to y a null constraint;
4)  replace  [x(...)] in attr by y;
5)  end loop;
end; //end of procedure
```

Listing.10. optional procedure

As an example of application of this procedure, we consider the expression:

[cite(<"papers"; papers; _>)]

To transform [cite(<"papers"; papers; _>)] which results from the previous procedure (closure_2), createUDTAttribute calls the "optional" procedure (see Listing.10) to eliminate symbols '[' and ']'. Then, "optional", executes its operations and returns the expression :

<"cite"; cite; null_constraint>.

5.6. Processing of named alternative

The namedAlternative function is created to process the named alternative, e.g. <!ELEMENT a (b|c)>). The following listing shows its definition.

```
function namedAlternative (attr(listOfItems)) return UDT Attribute;
begin
1)     if each item of listOfItems matches "<…>"
except one item that matches "+" then
2)       if the UDT attr is not yet created then
3)         for each item <x …> in listOfItems loop
4)           create an UDT named "x" if it's not created;
5)         end loop;
6)         create an UDT named attr that has an attribute
   named 'value' with a generic type(e.g., ANYDATA);
7)         add to attr a constraint that limits values of the attributes
   'value' to objects that are instances of types 'x' created
    by 'for each' above at line 3 to 5;
        // we call this constraint : c_attr
8)       end if;
9)     return <"attr"; attr; c_attr>;
10)   end if;
end; // end of function
```

Listing.11. namedAlternative function

As an example of application of this function, we assume that the element author contains the element address defined by:

<!ELEMENT address (email, phone)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT phone (#PCDATA)>

We have

φ(address)=address(φ (email | phone)) then

φ(address)=address(+, φ (email), φ (phone)) (see 0).

We have also

φ(email)=email (<value; varchar;_>) and φ(phone)=phone (<value; varchar;_>) .

then, we obtain the expression

address(+,<"email";email;_>,<"phone";phone;_>)

With lines between 6 and 9 in Listing.11, we obtain the expression

<"address"; address; c_atrr> where address is UDT with one attribute named value.

5.7. Processing of unnamed alternative

The unnamedAlternative procedure is created to treat the unnamed alternative, e.g. <!ELEMENT a (d, (b|c))>). The following listing presents its definition.

```
procedure unnamedAlternative (attr(listOfItems)) ;
i integer; /* variable for counting the number of attributes that are added in the case of the alternative which is
not surrounded by an element (for example <!ELEMENT a  (b|c), d>).*/
begin
1)         i=0;
2)         for each item (+,…) in listOfItems loop
3)           i←i+1;
4)           Replace, in attr, (+,…) by   createUDTAttribute(_attr_i(+,…)); // _attr_i is created for alternative
5)         end loop;
end; // end of procedure
```

Listing.12. unnamedAlternative procedure

As an example of application of this procedure, we assume that the element author is defined by

<!ELEMENT author (fn, ln, (email | phone)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT phone (#PCDATA)>

We have

φ(author)=author(φ(fn), φ(ln), φ(email | phone))

then

φ(author)=author(φ(fn), φ(ln), (+, φ(email),(phone)))(see 0).

Transformation of the expression (+, φ(email),(phone)), using lines between 2 and 5 in Listing.12, generates _author_1 (+, <"email";email;_>,<"phone";phone;_>)),

and then the expression
<"_author_ "; _author_ ; c__author_1> where _author_ is UDT with one attribute name value.

5.8. Logical mapping: creation of the UDT associated to XML schema

Now we consider the function *CreateUDT* shown in Listing.13. It makes a logical translation from an XML schema into object-relational schema. It takes a UDT obtained by applying the function φ to the root element of XML schema and returns a UDT with constraints. The body of this function is as follows:

```
Function CreateUDT(root(listOfItems)) return UDT with constraints ;
y UDT Attribute ; /*y is an UDT attribute variable.*/
begin
y=CreateUDTAttribute (root(listOfItems)); /*y has the form <"UDT"; UDT; Constraints>.*/
return <UDT, Constraints>; //a UDT with constraints;
end;
```

Listing.13. CreateUDT function.
If we apply CreateUDT to paper defined by:
<"paper"; paper; constraint_on(cite)> we obtain the UDT: <paper, constraint_on(cite)>.

5.9. Mapping algorithm

To finish the transformation, we have to create the structure that stores the instances of the UDT created at Listing.13. This structure is defined by an object table. The type of this table is the root element of the XML document. Steps that we have detailed to transform an XML schema into an OR schema are summarized in the following algorithm.

```
Algorithm transformation;
Input: a valid XML document with its DTD; Let be E its root;
Output: an object-relational schema;
begin
1)   Calculate φ(E) using the rules presented above in 0.
2)   Let be "E(listOfItems)" this value;
3)   Let be <E, Constraints> the UDT obtained by CreateUDT (E(listOfItems));/*algorithm at Listing.13 */
4)   Create an object table named "E_Table" with UDT E as type and constraints defined by E;
/*"E_Table" is created to store the instance of UDT E.*/
end; /*end of mapping*/
```

Listing.14. Algorithm of transformation.
Conceptually, the process of transformation can be schematized as follows

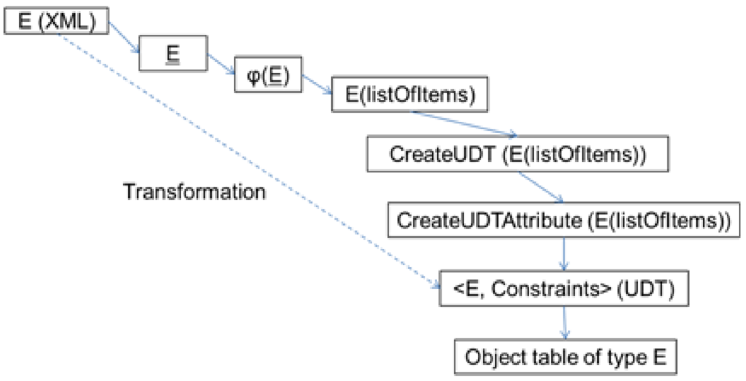

Fig. 1. Process of transformation.

VI. Example of The Transformation from XML Schema into or Model

Consider the following XML schema.

```
< ELEMENT paper (title, authors, cite?)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT authors (author+)>
<!ELEMENT author (fn, ln)>
<!ELEMENT fn (#PCDATA)>
<!ELEMENT ln (#PCDATA)>
<!ELEMENT cite (paper*)>
```

Listing.15. XML schema for paper

Applying this algorithm to the example in Listing.15, we create an object table named "Paper_Table" based on object type paper. The table has constraints defined by constraint_on (cite). The structure of this table is as follows. A detailed description of this schema is provided below.

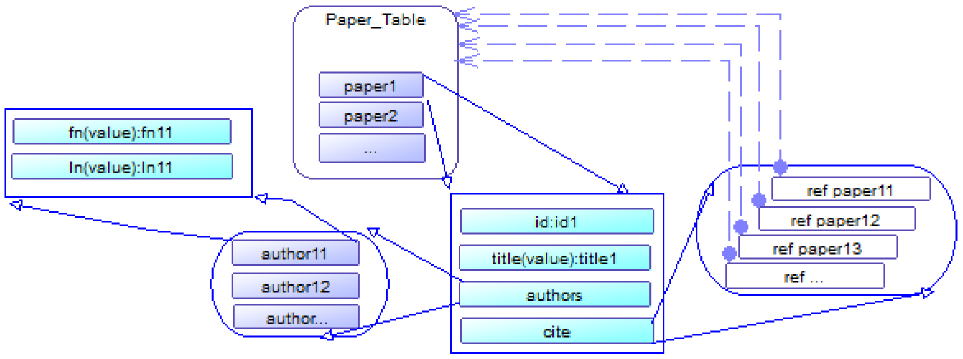

Fig.2 Structure of Paper table.

- Paper_table is an object table that stores the instances of type "paper". Each instance of paper has the following attributes:
- id : a scalar attribute;
- title: an object with an attribute named "value";
- authors: collections in which each element is an object with fn and ln attributes. Each of these latter has an attribute named "value"; and
- cite: collection of references to instances of type "paper".

In the appendix, we describe the OR presented in 0, using the SQL Oracle Database.

VII. Conclusion

In this paper, we have defined a methodology to transform an XML schema into OR model. This transformation is based on a mapping function which formalizes the steps involved in this methodology and can be used to develop a tool that allows an automatic generation of an OR schema. Due to the preservation of structural and semantic aspects in this conversion, we can construct the initial XML schema from its OR converted. Compared to other methods, our methodology integrates XML elements in a small number of object tables and is independent of any specific DBMS. As future work, we think to apply this methodology to make another transformation especially between XML data described in XML Schema languages and object-relational model.

References

[1] Informix, Http://Www-3.Ibm.Com/Software/Data/Informix/, 2003. .

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and J. D. Ullman, *Compilers Principles, Techniques, & Tools*, (in, 2nd ed, England: Pearson Education, 2007) pp. 42-50, 197-199, 204-205.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and J. D. Ullman, *Compilers Principles, Techniques, & Tools*, (in, 2nd ed, England: Pearson Education, 2007) pp. 116-122,159-163.

[4] T. Bray, Paoli, J., Sperberg-Mcqueen, and a. M. C. M., E., Extensible Markup Language (Xml) 1.0 (Second Edition), *W3C Recommendation. http://www.w3.orglTR2OOOlREC- XML-20001006l*, 2000/10.

[5] E. Castro, D. Cuadra, and M. Velasco, From Xml to Relational Models, *Informatica*, vol. *21(4)*, pp. 505-519, 2010/12.

[6] T. M. Connolly and C. E. Begg, *Database Systems: A Practical Approach to Design, Implementation, and Management*, 6 ed. (USA: Pearson Education, 2015).

[7] C. Coronel, S. Morris, and P. Rob, *Database Systems: Design, Implementation, and Management*, 10 ed. (Boston, USA: Joe Sabatino, 2012).

[8] A. Eisenberg and J. Melton, Sql:1999, Formerly Known as Sql3, *SIGMOD Record*, vol. *28(1)*, March 1999.

[9] A. Eisenberg, J. Melton, K. G. Kulkarni, J.-E. Michels, and F. Zemke, Sql: 2003, *SIGMOD Record*, vol. *33(1)*, pp. 119-126, 2004.

[10] S. Kanagaraj and D. S. Abburu, Converting Relational Database into Xml Document *IJCSI International Journal of Computer Science Issues*, vol. *9(2)*, pp. 127-131, 2012/3.

[11] J. Kim, D. Jeong, and D.-K. Baik, A Translation Algorithm for Effective Rdb-to-Xml Schema Conversion Considering Referential Integrity Information, *Journal of Information Science and Engineering*, vol. *25*, pp. 137-166, 2009/1.

[12] D. Lee, M. Mani, and W. W. Chu, Solving Schema Conversion Problem between Xml and Relational Models: Semantic Approach, *ResearchGate*, 2003/7.

[13] J. Melton, *Advanced Sql:1999: Understanding Object-Relational and Other Advanced Features (the Morgan Kaufmann Series in Data Management Systems)*, 2003).

[14] S. Navathe and R. Elmasri, *Fundamentals of Database Systems*, (in Addison-Wesley, Ed., ed, 2011.

[15] J. Price, *Oracle Database 11g Sql*. (USA: McGraw -Hill, 2008).

[16] M. Stonebraker, L. A. Rowe, and M. Hirohama, The Implementation of Postgres, *IEEE on Knowledge and Data Engineering*, vol. *2*, pp. 125-142, Mars 1990.

Appnedix

In this appendix, we put the scripts that implement the object-relational schema shown in 0. They are written in accordance with the standard SQL3 and respect the object-relational DBMS Oracle schema.

a) Creation of simple object types:

```
create type title as object(value varchar(45));/
create type ln as object(value varchar(45));/
create type fn as object(value varchar(45));/
create type author as object("ln" ln, "fn" fn);/
```

b) Creation of nested table type authors:

```
create type authors is table of author;/
```

c) Creation of incomplete object type paper in order to allow recursion:

```
create or replace type paper ;/   --incomplete object type
```

d) Creation of collection type of references: papers, and cite object type with an attribute containing these references

```
create type papers is table of ref paper; /
create type cite as object("papers" papers);/
```

e) Now, we complete the object type paper:

```
create or replace type paper as object
 (id varchar(45),  "title" title, "authors" authors, "cite" cite);/
```

f) Creation of the object table that will contain the XML schema:

```
create table paper_table of paper
(constraint  NAC_id check (regexp_like(id,'^([a-z]|\.|-|_|:)([a-z]|[0-9]|\.|-|_|:)*$')),
Constraint UC_id unique(id),
constraint REQUIRED_id id not null)
nested table "cite"."papers" store as extern_papers
nested table "authors" store as extern_authors;/
```