International Journal of Multidisciplinary and Current Research ISSN: 2321-3124
Research Article Available at: http://ijmcr.com

A Translation from XSD into ORM using Mappings

Mustapha Machkour[#,!*], Karim Afdel[#,!], Said Aminzou[$], El Hassan Megeder[*]

[#]Department of Mathematics and Computer Sciences, Faculty of Sciences, Agadir, Morocco
[!]Laboratory of the Computing Systems and Vision, Faculty of Sciences, Agadir, Morocco
[$]Laboratory of Energy Engineering, Materials and Systems, ENSA, Agadir, Morocco
[*]Laboratory of Image and Form Recognition - Intelligent and Communicative Systems Faculty of Sciences, Agadir, Morocco

Accepted 01 Jan 2017, Available online 07 Jan 2017, **Vol.5 (Jan/Feb 2017 issue)**

Abstract

The XML Schema Definition (XSD) describes with high precision, better than the Document Type Definition (DTD), the structure and semantics of XML data. To translate the XSD schemas into database schemas as has been done for the DTD, the translation methodologies are needed. The object of this article is to present a method that makes it possible the translation of an XSD schema into an object-relational database schema (ORS). To preserve the integrity constraints defined in XSD such as type constraints, value constraints, and structure constraints during the process of translation, the extended DTD (XDTD) schema is defined. The XDTD helps to represent the XML element in DTD with XSD constraints. In this method, we introduce new specifications for XML, ORM and define the mapping from XSD into XDTD and XDTD into ORS. These mappings allow an automated translation without human intervention.

Keywords: *XML, XSD, XDTD, Integrity Constraint, Object-Relational schema, Translation, Mapping.*

Introduction

XSD [28] is used to describe XML [6] data in an improved way which is adequate for many areas such as platform configurations, communication protocols, programming environments, and data exchange. This improvement concerns mainly the structure and semantics of data which are, in general, expressed via the attributes and the facets [29].

The database users especially those using the relational model [10,27] and the object-relational model (ORM) [8,9,12,26,30] are faced with the problem of manipulating these XML data. One solution to this issue is to translate XSD data into database models.

This paper proposes a methodology to convert an XSD data into the ORM. One increasingly uses this latter due to its many advantages. On the one hand, it preserves the qualities of the relational model and introduces concepts used in the object model [10,16,27] such as user-defined type (UDT) or object type, collections (limited and unlimited), inheritance, On the other hand, it reduces even eliminates the impedance mismatch between object-oriented languages and relational databases [10, 11].

To have our aim, we extend the DTD to XDTD, define two mappings and a composition between them. The first mapping relates XSD to XDTD, and the second is between XDTD and ORM. The extension is needed to preserve constraints defined in XSD, and the mappings allow possible an automated translation.

*Corresponding author's ORCID ID: 0000-0000-0000-0000
DOI: https://doi.org/10.14741/ijmcr/v.5.1.4

Although our methodology focuses on XSD, it applies to any XML schema based on elements composed of attributes and other nested subelements. To implement and test our method, we have used the Oracle Database which supports many features of SQL-2003, especially those, which employ in this paper.

This article is structured as follows. In next section, we cite some related works on model translation. In section 3, we present the terminology used in the paper. Section 4 describes the extension from XSD into XDTD. Section 5 details the mapping between XDTD and ORM. Sections 6 and 7 give the algorithms that create the ORM schema. Section 8 provides an example of translations and Section 9 presents conclusions.

Related works

Many works have dealt with the conversions between data models. For instance, we cite the conversion between relational and object-relational models [3,13,14], between ER and OO models [5], and between UML and XML models [2,4,18].

To access XML data with database systems, one designed and developed schema translation methods [7,19-21]. Nowadays, despite the support of databases for XML, the need to translate XML into a database model continues to impose itself. That is because there are few databases supporting XML[17]. Moreover, there is no standard for accessing XML stored in databases.

Some of these methods translate XML documents that conform to DTD into corresponding relational data [9]. Algorithms have been designed and implemented in this direction. On the other side, have been developed translation methods from a relational schema into its corresponding XML[15,19,20].

But, due to its shortcoming, the relational model is increasingly replaced by the object model or extended to the object-relational model. Consequently, current relational database systems have included concepts of object-oriented paradigm to allow data modeling and their relationships in a high level of abstraction and thus became object-relational database systems.

Meanwhile, to better describe XML data, several schema languages have been developed. We will be interested in the XSD language.

Terminologies

In this section, we give the vocabularies that we use throughout the paper. These cover the DTD, XSD, XDTD and object-relational model and are based on those employed in works previously published [22-25].

Specification for DTD

The notations that we use in this subsection relates to attribute and element in DTD. An attribute has a description and a type. The description is specified by

description ::= #REQUIRED | #IMPLIED | #FIXED value | value

Fig.1 Specification of description

and the type is defined by
type ::= ID|CDATA|IDREF |IDREFS |NMTOKEN |NMTOKENS |Enumerated Attribute list
Then, if attri is an XML attribute, its specification denoted by attri (attri underscored) is given by

attri ::= <attri; type; description>

Fig.2 Specification of the attribute attr according to its DTD

Let E be an element in DTD, attrs=$(attri)_{1\le i\le n}$ its attributes (we suppose that E has n attributes), and D its model of content [6]. The specification of E denoted by E (E underscored) is given by:

E ::= <E; attrs; D>

Fig.3 Specification of the element E according to its DTD

Items of this specification comprise the name of the element E, the specification of its list of attributes (attrs) and the specification of its content model (D). The value of attrs is given by a list of attribute specifications (see Fig.**2**), i.e. we can write the following expression:

attrs :: = $(attri)_{1\le i\le n}$.

Fig.4 Specification of the attribute list in DTD

Therefore, the specification denoted by "_" can be regarded as a map[1] function applied to each attribute in the list.
Second, the content model D of an element E in DTD can be:

- List of symbols between "<! ELEMENT ElementName (" and ")>",
- EMPTY,
- ANY.

Elements of D are connected by sequence, alternative, Kleene closure, transitive closure, and an optional value. The following grammar that we call G describes these connections:

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 (replaces *),
f) E :: = {E},E for the transitive closure (replaces +),
g) E :: = [E] for an optional value(replaces ?),
h) E :: = #PCDATA for a simple type.

Fig.5 Elements of the G Grammar

To have the specification of each item in the grammar G, we associate to it the following grammar that uses the symbol "_" (underscore). We call this grammar G (G underlined).

Recall that E (E highlighted, see Fig.**3**) gives the specification of the element E. The productions order in the two grammars, G and G, is preserved.

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

Fig.6 Elements of the Grammar G

Here, the symbol of specification "_" is also a map function applied to each element or value to the left of the symbol "::=".

Examples

Before proceeding further, it will be convenient to give some examples of the specifications. These examples repose on the following listing.

[1] A function that transforms a list by applying another function to each of its elements.

```
<!ELEMENT journal  (volume+)>
<!ATTLIST journal id ID #REQUIRED category CDATA #IMPLIED >
<!ELEMENT volume (issue+)>
<!ELEMENT issue (paper+)>
<!ELEMENT paper (title, author)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
```

Listing 1. Example of DTD

As the value of D for element journal is volume+ and its attributes are id and category, then its specification is given by:
journal ::=<journal; id, category; volume+>.
Likewise, since the element volume has the issue+ as the content model and has no attributes, its specification is:
volume :: = <volume; ; issue+>
and the element paper has the following specification:
paper::=<paper; ; title, author>, since the value of its content model D is "title, author."
Since the elements title and author have no attributes, their specifications are:
title =<title;_; #PCDATA> and author::= <author;_; #PCDATA>, and hence, the value of D for both elements title and author is #PCDATA.

Specification for XSD

The specification of an element in XSD reposes on its type. The following regular expression shows a simplified To of this:

type::= simpleType | complexType

Fig.7 Simplified type in XSD

Then, if E is an element of type "type", its specification is given by E (E double underlined):

E:: = <E; type>.

Fig.8 Specification of the element E in XSD

Specification for XDTD

To preserve the integrity constraints not supported by DTD, during translation from XSD into DTD, we define the extended DTD (XDTD for short) and name these constraints XSD constraints (XSDC for short).
In XDTD, we represent an attribute attri by $attri_{extended}$ which we specify in the following figure.

$attri_{extended}$ ::= <attri; type; description; XSDC>,

Fig.9 Specification of the attribute attr in XDTD

which we also note by

$attri_{extended}$ ::= <attr; XSDC>,
where attri (attr underscored) represents the definition of attr in DTD (see
Fig.**2**).

Like the specification, denoted by "_", the extension denoted by "extended" is considered as a map function. That enables us to write the following equalities:

$attrs_{\mathbf{extended}}=((attri)_{1\le i\le n})_{\mathbf{extended}}$
$=((attri)_{1\le i\le n})_{\mathbf{extended}}$
$=(attri_{\mathbf{extended}})_{1\le i\le n}$

So, we have the following relations

$attrs_{extended}=((attri)_{1\le i\le n})_{extended} = (attri_{extended})_{1\le i\le n}$

Fig.10 Specification of the list attrs in XDTD

Similarly, the specification of an element E in XDTD is given by

$E_{extended}$ :: = <E; $attrs_{extended}$; $D_{extended}$; XSDC>

Fig.11 Specification of the element E in XDTD

The expression of $D_{extended}$ deduced from the grammar **G** (see
Fig.**6**) is defined by the following grammar which we name $\mathbf{G}_{extended}$:

a) $E_{extended}$ ::= ANY
b) $E_{extended}$ ::= EMPTY
c) $E_{extended}$ ::= $E_{extended}$, $E_{extended}$
d) $E_{extended}$ ::= $E1_{extended}$ + $E2_{extended}$
e) $E_{extended}$ ::= {$E_{extended}$}
f) $E_{extended}$ ::= {$E_{extended}$}, $E_{extended}$
g) $E_{extended}$ ::= [$E_{extended}$]
h) $E_{extended}$ ::= #PCDATA

Fig.12 Elements of the Grammar $G_{extended}$

Specification for the object-relational model

The concepts of the OR model include [11,12,26]:

- user defined type (UDT)[2]: 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[3] (AT) : to define limited collection;
- multiset type[4] (MT): to specify unlimited collection;
- object table: used to create a table based on UDT and stores objects;

[2] We use the terms UDT, type and user type interchangeably.
[3]An array type is a limited and ordered collection of any element of any type admissible.
[4]A multiset type is an unlimited and unordered collection of any element of any type admissible.

- inheritance, for making inheritance relation between types.

Let attri, 1≤ i ≤n, be the attributes of a UDT E. The specification of attri comprises its name, its type, and its constraints which represent a list of constraints on its permissible values. We represent this specification by:

<attri; type; constraints>

Fig.13 Specification of the attribute attri in ORM

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.14 Specification of the UDT E

The next sections describe the translation from XSD into ORS. This translation is done by composition in two phases. The first one is from XSD into XDTD and the second is from XDTD into ORM.

Translation from XSD into XDTD

In this section, we detail the steps that translate a schema from XSD into XDTD.

Mapping from XSD into XDTD

In this subsection, we show how translating XSD into XDTD. There is much software that converts XSD into DTD but does not consider the XSD constraints. Below we detail how these constraints can be preserved using the XDTD and functions that compute these constraints.

Let E be an element in XSD. Its specification is given by (see
Fig.**8**):
E:: = <E; Type>
Consider the mapping Ψ defined from XSD to XDTD. This function takes an XSD element E (resp. an XSD attribute attri) and reformulates it in an XDTD $E_{extended}$ (resp. an XDTD attribute $attri_{extended}$)):

Ψ : {XSD} →{XDTD}
E → Ψ(E) = $E_{extended}$
attri → Ψ(attri)= $attri_{extended}$

Fig.15 Definition of the function Ψ from XSD to XDTD

We recall that $E_{extended}$=<E; $attrs_{extended}$; $D_{extended}$> and $attr_{extended}$=<attr; XSDC>.
We use the term XSDElement to refer to an element in XSD, and the term XSDAttribute to refer to an attribute in XSD.
To obtain the value of XSDC, we define the three followings functions:

- VC (XSDElement E) return string; //Returns a list of value constraints of the element E,
- TC (XSDElement E) return string; // Returns a list of type constraints of the element E

and

- SC (XSDElement E) return string;// Returns a list of structure constraints of the element E.

These functions allow writing the following regular expression:
XSDC ::= (VC | TC | SC)*
The functions VC and TC are also applied, using the overloading, for an attribute parameter. That means we can write VC (attribute) and TC (attribute).
Below, we give the definition of each function.
The definition of the function VC is:

```
VC (XSDElement E) returns list of constraints;
/*function called for the element E with simple type*/
Cs: string; /* variable to concatenate all constraints of value*/
begin
set Cs=""; //initialization of the Cs.
for each constraint c in (Bounds, Pattern, Enumerated values, default, fixed) of E loop
create a logic constraint using c; /*let LC be the name of this constraint*/
set Cs=Cs +","+ LC; /* here the symbol "+" denotes the concatenation*/
end loop;
return Cs;
end;
```

Listing 2. Definition of the function VC

The definition of the function TC is:

```
TC (XSDElement E) returns list of constraints;
/*function called for element E with simple type*/
Cs : string; /* variable for grouping all constraints of type*/
begin
Cs="";
for each constraint c in (type, base, Length, Precision) of E loop
create a logic constraint using c; /*let LC be the name of this constraint*/
set Cs=Cs + ","+ LC;
end loop;
return Cs;
end;
```

Listing 3. Definition of the function TC

The Definition of the function SC is:

```
SC (XSDElement E) returns string;
/*function called for the element E with different default value for minOccurs and/or maxOccurs*/
```

```
v_minOccurs, v_maxOccurs : string;
begin
Let v_minOccurs be the value of minOccurs associated to element E;
Let v_maxOccurs be the value of maxOccurs associated to element E;
return v_minOccurs+ "," + v_maxOccurs;
end;
```
Listing 4. Definition of the function SC

Before proceeding further, it would be useful to deal with some translation examples from XSD to XDTD.
Consider, for instance, the following XSD which describes the XML element "nb_pages":

```
<xsd:element name="nb_pages">
<xsd:simpleType>
<xsd:restriction base="xsd:integer">
<xsd: minInclusive value="4"/>
<xsd:maxInclusive value="10"/>
</xsd:restriction>
</xsd:simpleType>"
</xsd:element>
```
Listing 5. XSD of the element nb_pages

Let's look for the specification of nb_pages in XDTD, i.e. $nb\_pages_{extended}$ or nb_pages and XSD constraints.
The type of nb_pages is delimited by "<xsd:simpleType>" and "</xsd:simpleType>" :

```
type="<xsd:simpleType>
<xsd:restriction base="xsd:integer">
<xsd: minInclusive value="4"/>
<xsd:maxInclusive value="10"/>
</xsd:restriction>
</xsd:simpleType>"
```
Listing 6. The type of the element "nb_pages"

After that, the representation of nb_pages using our specification is
nb_pages =<nb_pages; Type>.
So the image of nb_pages by Ψ (see Fig.**15**) is
$\Psi(nb\_pages) = nb\_pages_{extended}$
Since the corresponding type for integer in DTD is #PCDATA, then
nb_pages=<nb_pages; ; #PCDATA>.
we compute the XSD constraints as follows:

- For type constraint: this constraint, returned by the algorithm TC (0), is related to the type of nb_pages which is "integer" obtained from the attribute "base". This constraint is expressed using the regular expression [0..9]+);
- For value constraint: the value of this constraint, returned by the algorithm VC (see VC (XSDElement E) returns list of constraints;

```
/*function called for the element E with simple type*/
Cs: string; /* variable to concatenate all constraints of value*/
begin
set Cs=""; //initialization of the Cs.
for each constraint c in (Bounds, Pattern, Enumerated values, default, fixed) of E loop
create a logic constraint using c; /*let LC be the name of this constraint*/
set Cs=Cs +","+ LC; /* here the symbol "+" denotes the concatenation*/
end loop;
return Cs;
end;
```

- **Listing 2**), is "4, 5" obtained from constraining facets (maxInclusive, minInclusive).

Then, if we extend the expression of "nb_pages" by adding these constraints, we obtain the specification of nb_pages in XDTD. This is given in the following figure:
$nb\_pages_{extended}$=<nb_pages; ;#PCDATA ;TC(nb_pages), VC(nb_pages)>

Fig.16 Specification of nb_pages in XDTD

In the previous example, there is no structural constraint. For this type of constraint, we consider, for instance, the following XSD

```
<xsd:element name="authors">
<xsd:complexType>
<xsd:sequence minOccurs="1" maxOccurs="5">
<xsd:element name="author">
<xsd:simpleType>
<xsd:restriction base="xsd:string">
<xsd:minLength value="4"/>
<xsd:maxLength value="50" />
</xsd:restriction>
</xsd:simpleType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
```
Listing 7. The XSD of the element authors

Here, the specification of the element authors is
authors=<authors; type>,
where "type" is the string between "<xsd:complexType>" and "</xsd:complexType>. The expression of authors in DTD is given by.

```
<!ELEMENT authors (author)+>
<!ELEMENT author (#PCDATA)>
```
Listing 8. DTD of the element authors

First, if we look for the image of the element authors by Ψ (see Fig.**3** and Fig.**15**), we get
$\Psi(authors)=authors_{extended}$ with
authors=<authors;;(author)+>.

In this case, the value of the structure constraint returned by "SC(authors)" (see SC (XSDElement E) returns string;
/*function called for the element E with different default value for minOccurs and/or maxOccurs*/
v_minOccurs, v_maxOccurs : string;
begin
Let v_minOccurs be the value of minOccurs associated to element E;
Let v_maxOccurs be the value of maxOccurs associated to element E;
return v_minOccurs+ "," + v_maxOccurs;
end;
Listing 4) is "1,5", where "1" is the value of minOccurs and "5" is the value of maxOccurs. The specification of the element "authors" in XDTD is then

$authors_{extended}$ =<authors; $(author_{extended})$+; SC(authors)>,

Fig.17 Specification of the element authors in XDTD

where $author_{extended}$ is the image of the element author by Ψ given by (VC(author) returns constraints defined by minLength and maxLength facets (see Listing 7))

Ψ(author)=$author_{extended}$ =<author;;#PCDATA;VC(author))

Fig.18 Specification of the element author in XDTD

That's all for the translation of an element. We can similarly do for an attribute. To see that, we consider the following XSD that describes the element "journal" with two attributes "id" and "issn".

```
<xsd:element name="journal">
<xsd:complexType>
<xsd:attribute name="id" type="xsd:ID" use="required"/>
<xsd:attribute name="issn">
<xsd:simpleType>
<xsd:restriction base="xsd:string">
<xsd:length value="9"/>
<xsd:pattern value="\d{4}\-\d{3}[/dX]"/>
</xsd:restriction>
</xsd:simpleType>
</xsd:attribute>
</xsd:complexType>
</xsd:element>
```

Listing 9. XSD with attributes

Let us find the specification of the attributes id and issn in XDTD.
The specification of the element journal in DTD is:

```
<!ELEMENT journal EMPTY>
<!ATTLIST journal id ID #REQUIRED issn CDATA #IMPLIED>
```

Listing 10. DTD of the element journal

Thus, its image by Ψ, is given in the below expression:

Ψ(journal)=$journal_{extended}$ =<journal; $id_{extended}$, $issn_{extended}$; XSDC>.
So, since there is no constraint for element journal in XSD (i.e., XSDC is empty) its specification in XDTD is given by $journal_{extended}$=<journal; $id_{extended}$, $issn_{extended}$; _>.
For the attribute issn, we have
Ψ(issn)=$issn_{extended}$ = <issn; CDATA; #IMPLIED; XSDC>,
where XSDC is the constraint obtained by VC(issn) and TC(issn): VC(issn) returns the value of the attribute "pattern", and TC(issn) returns the value of the attribute "length".
Whence the specification of issn in XDTD is

Ψ(issn)=$issn_{extended}$ =<issn; CDATA; #IMPLIED; VC(issn), TC(issn >,

Fig.19 The specification of issn in XDTD

With the same manner, we calculate the specification of the attribute id in XDTD. We have
Ψ (id)=$id_{extended}$ with id=<id; ID; #REQUIRED> (see Listing 10).
Due to the absence of value constraint and type constraint, the specification of the attribute id is

$id_{extended}$=<id; ID; #REQUIRED; _>

Fig.20 Specification of id in XDTD

So far, we have shown that elements and attributes of XSD can be specified in XDTD using DTD and constraints defined in XSD. To end the translation, we shall translate the XDTD into ORS in the next section.

Translating XDTD into Object-Relational schema

In this paragraph, we present a mapping that allows a formal translation from XDTD into the object-relational schema. This translation based on the notations used in the paper cited at [23] uses extended notations (see Fig.**9** and Fig.**11**) to include constraints defined in XSD that we have named XSDC. We add these constraints to constraints computed in this paragraph for both attribute and element.

Mapping between XDTD and Object-relational model

To formalize the translation between XDTD and Object-relational model (ORM), we define a polymorphic[5] function ϕ which associates to an element E in the XDTD a UDT E (we suppose that E has n attributes) in the ORM

ϕ : XDTD → ORM
ϕ : $E_{extended}$ → ϕ($E_{extended}$) =E((<attri;type; constraints>)$_{1\leq i \leq n}$)

Fig.21 Specification of UDT ϕ($E_{extended}$)

[5] Function that accepts an arbitrary number of different types arguments.

We will show, throughout this section, how are calculated the attributes of the UDT "E" using the function ϕ.
We know that $E_{extended}$ =<E; $attrs_{extended}$; $D_{extended}$; XSDC>, (see
Fig.**11**).
If we apply ϕ on both sides of the equality above, we obtain
$\phi(E_{extended}) = \phi$ (<E; $attrs_{extended}$; $D_{extended}$; XSDC>) which is equivalent by definition to

$\phi(E_{extended}) = E(\phi(attrs_{extended}) \cup \phi(D_{extended}), XSDC)$,

Fig.22 Definition of a UDT E image by ϕ of the element E where the symbol 'U' denotes the union operator. We have then the relation

If we combine the relations in 0
Fig.21 and
Fig.22, we obtain
$E((<attri; type; constraints>)_{1\le i \le n}) = E(\phi(attrs_{extended}) \cup \phi(D_{extended}), XSDC)$.
From this equality, we can say that the list of the attributes of the UDT E is the union of the images, obtained by ϕ, of the attribute specifications of the XML element E, i.e. $\phi(attrs_{extended})$) and its content model, i.e. $\phi(D_{extended})$. The calculation of these images is the purpose of the next subsections.

Calculation of φ($attrs_{extended}$)

$\phi(attrs_{extended})$ is a list of attribute specifications for a UDT. We obtain them by the following algorithm:

Algorithm listAttributes;
Input $attrs_{extended}$: list of attributes in XDTD;
Output $\phi(attrs_{extended})$: list of attribute specifications in ORM;
begin
if $attrs_{extended}$ = empty then /* There is no attributes for the XML element. */
  $\phi(attrs_{extended})$ = empty string;
elseif $attrs_{extended} = (attri_{extended})_{1\le i\le n}$ then
  $\phi(attrs_{extended}) = (\phi(attri_{extended}))_{1\le i\le n}$ //ϕ works as map function.
end if;
return ϕ ($attrs_{extended}$);
end;
Listing 11. Calculation of $\phi(attrs_{extended})$

We have in
Fig.**9**
$attri_{extended}$=<attri; type; description; XSDC>.
If we introduce ϕ, in this last formula, we obtain
$\phi(attri_{extended}) = \phi$(<attri; type; description; XSDC>).
By definition, the value of ϕ(<attri; type; description; XSDC>) is obtained by the following expression
ϕ(<attri; type; description; XSDC>)::=<attri; ϕ(type) minus Constraints; ϕ(description) plus Constraints plus XSDC>
and, by transitivity, we get the expression
$\phi(attri_{extended})$=<attri;ϕ(type) minus Constraints; ϕ(description) plus Constraints plus XSDC>
Fig.23 Specification of the UDT attribute attri

So, to have the value of $\phi(attri_{extended})$ requires ϕ(type), ϕ(description), Constraints and XSDC. The next subsections deal with these computations.

Calculation of φ(type)

ϕ(type) returns an expression that defines the type and constraints in ORM of the attribute attri. The following table shows this expression. The constraints are defined using the regular expressions [1].

Table 1 Calculation of ϕ(type)

type in XDTD	ϕ (type) in ORM	
	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

In what follows, we explain the two right columns in this table.
In the column Type:
- varchar (n) is a standard type of strings used in database systems. n is the size of type.
- array (p) is a data type representing a collection of values in object-relational databases. p is the size of the collection.
- Multiset is a data type used in object-relational databases. It represents an unlimited collection.

In the column constraints, we have patterns that values of an attribute must respect to preserve the semantic values of XML elements attribute. Those patterns are similar for all constraints. We can use an applicative constraint to maintain this type of constraint (for example check constraint with the operator like).

These patterns use:
- Letter: regular definition [1] that denotes the expression [A..Za..z] and
- Digit: regular definition that denotes the expression: [0..9].

There are also, in the column Constraints:

Foreign key Constraint (FKC): refers to referential integrity constraint which is usual in the database literature.
Unique Constraint (UC): checks the unicity of the attribute values.

Enumerated List Constraint (ELC): Constraint with a list of values corresponding to the enumerated value list that specifies the content model of XML elements attributes. We can use check constraint with the operator like to maintain the ELC constraint.
For convenience, we use the term LAC (contraction for Lexical Attribute Constraint) to denote the constraint based on the regular expression: (Letter|_)(Letter|_|Digit|:|.|-)*

Calculation of φ(description)

The computation of ϕ(attrs $_{extended}$) also requires the calculation of ϕ(description). The value of ϕ(description) is a list of usual constraints in databases system. This value is obtained using 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 show how the function ϕ operates, we consider the example below:

<!ELEMENT journal EMPTY>
<!ATTLIST journal id ID #REQUIRED>
<!ATTLIST journal issn CDATA #IMPLIED>
Listing 12. Element journal.

We recall that this element journal results from the XSD schema in Listing 9.
Calculation of ϕ (journal$_{extended}$)
This element journal has two attributes id and issn.
If we apply ϕ to the element journal we obtain:
ϕ(journal$_{extended}$)=journal(ϕ (id$_{extended}$), ϕ (issn$_{extended}$)).
Then "journal" on right of the symbol "=", is a UDT with two attributes ϕ(id$_{extended}$), ϕ(issn$_{extended}$) to compute.
In order to have ϕ(journal$_{extended}$) we must calculate and ϕ(issn$_{extended}$)= ϕ (<issn;CDATA;#IMPLIED>).
Computation of ϕ(id$_{extended}$)
The value of ϕ(id$_{extended}$) is given by
ϕ(id$_{extended}$)= ϕ (<id; ID; REQUIRED>)
According to formula in 00 and 0, we have
ϕ (id$_{extended}$) =<id; ϕ(ID) - (LAC+UC); ϕ(#REQUIRED) + (UC+LAC)+ XSDC>.
Since
ϕ(ID)=varchar + (LAC+UC), ϕ (#REQUIRED)=not null and XSDC="" (see
Fig.**20**),
ϕ (id$_{extended}$) becomes
ϕ(id)=<id; varchar; not null+ LAC + UC>.
whence ϕ(id$_{extended}$) is an attribute of the UDT journal with the following specifications:
- Id: name of the attribute;
- Varchar: type of id;
- not null, LAC, and unique are constraints of id (object attribute).

Computation of ϕ(issn$_{extended}$)
In the same way, we compute ϕ(issn$_{extended}$). We have the expression
ϕ(issn$_{extended}$)= ϕ (<issn; CDATA; #IMPLIED>), which becomes, using 0 and 0,
ϕ(issn$_{extended}$)=<issn; varchar; null + XSDC>.
Since XSDC="", the last expression becomes
ϕ(issn$_{extended}$)=<issn; varchar; null>.
Then the UDT journal becomes
journal(<id;varchar;not null+LAC+UC)>,<issn; varchar;null>).
End of the example.
The following algorithm generalizes these steps:

Algorithm OR_attribute_from_XDTD_attribute;
Input attri$_{extended}$: an attribute in XDTD;
Output ϕ(attri$_{extended}$): an attribute in ORM;
Begin
  Calculate ϕ(type);
  Calculate ϕ(description);
  Return <attri; ϕ(type) minus constraints; ϕ(description) plus constraints plus XSDC>;
End;
Listing 13. Algorithm returning an OR attribute from an XDTD attribute.

Calculation of φ(D$_{extended}$)

We recall that the expression of ϕ(E$_{extended}$) is based on ϕ(attrs$_{extended}$) and ϕ(D$_{extended}$) (see
Fig.22). Since we have computed ϕ(attrs$_{extended}$), we will now compute ϕ(D$_{extended}$).
The value of D$_{extended}$ is obtained using the grammar G$_{extended}$ presented above at 0.
To obtain the value of ϕ(D$_{extended}$), we associate to the grammar G$_{extended}$ the grammar ϕ(G$_{extended}$) defined by:

a) ϕ(E$_{extended}$)= ϕ(ANY) ::= AnyData[6] or AnyType[7]. (Generic type in object-relational model)
b) ϕ(E$_{extended}$)= ϕ(EMPTY) ::= Empty string
c) ϕ(E$_{extended}$)= ϕ(E1$_{extended}$, E2$_{extended}$) ::= ϕ(E1$_{extended}$), ϕ(E2$_{extended}$) E1 and E2 are used to distinguish between E at left of '=' and E at the right of the '='
d) ϕ(E$_{extended}$) ::= (+, ϕ(E1$_{extended}$), ϕ(E2$_{extended}$)). Here, we define a generic type that can hold the object types ϕ(E1$_{extended}$) and ϕ(E2$_{extended}$)
e) ϕ(E$_{extended}$) ::={ϕ(E$_{extended}$)}, list of ϕ(E$_{extended}$) with null constraint
f) ϕ(E$_{extended}$) ::={ϕ(E$_{extended}$)} list of ϕ(E$_{extended}$) with not null constraint

[6] Type used in Oracle DBMS.
[7] Type used in Oracle DBMS.

g) $\phi(E_{extended})$ ::= [$\phi(E_{extended})$], $\phi(E_{extended})$ with null constraint
h) $\phi(E_{extended})$ ::= ϕ(#PCDATA).

Fig.24 Elements of the grammar $\phi(G_{extended})$

By definition, the value ϕ(#PCDATA) is given by:
ϕ(#PCDATA) = <value; varchar; XSDC >
where
- value is an attribute of the UDT containing the value of the XML element,
- varchar is the type of the attribute value, and
- XSDC is the constraint of #PCDATA in XSD of the attribute value.

Since there is no constraint of #PCDATA in XSD, the XSDC is empty. Therefore, the equality above becomes:

ϕ(#PCDATA) = <value; varchar; _ >

Fig.25 Value of ϕ(#PCDATA)

In order to further understand the translation using ϕ, we next present some translation examples from XDTD to object-relational model.

Examples

Example 1

For element $nb\_pages_{extended}$, we have (see Fig.**16**):
$nb\_pages_{extended}$=<nb_pages; ; #PCDATA ; TC(nb_pages), VC(nb_pages)>.
If we apply the function ϕ to nb_pages, we obtain
$\phi(nb\_pages_{extended})=\phi$(<nb_pages; ;#PCDATA ;TC(nb_pages), VC(nb_pages))>)
=nb_pages(ϕ(#PCDATA); TC(nb_pages), VC(nb_pages)).
If we replace ϕ(#PCDATA) by its value <value; varchar; _ > (see Fig.**25**), we get
$\phi(nb\_pages_{extended})=\phi$(<nb_pages; ;#PCDATA ;TC(nb_pages), VC(nb_pages))>)
=nb_pages(<value;varchar;_>;TC(nb_pages),VC(nb_pages)).
In this last expression, nb_pages is a UDT that has TC(nb_pages) and VC(nb_pages) as constraints and value as an attribute with type varchar.

Example 2

In the same way, we can apply the function ϕ to the element author defined by:
$author_{extended}$ =<author; ; #PCDATA; VC(author)), (see Fig.**18**):
and we get
$\phi(author_{extended})$ = ϕ(<author; ; #PCDATA; VC(author))
=author(ϕ(#PCDATA); VC(author))
=author(<value; varchar; _ >; VC(author))
whence,
$\phi(author_{extended})$ = author(<value; varchar; _ >;VC(author))
Thus, nb_pages is a UDT that has VC(author) as a constraint and "value" as an attribute with type varchar.

Example 3

For a somewhat sophisticated example, we suppose that the element "author" has two attributes "fn" and "ln" specified by:
$fn_{extended}$ =<fn; ; #PCDATA; VC(fn)) and
$ln_{extended}$ =<ln; ; #PCDATA; VC(ln)).
In this case, the specification of the "author" in XDTD is
$author_{extended}$=<paper; ; $fn_{extended}$, $ln_{extended}$; XSDC>.
By applying ϕ, we obtain
$\phi(author_{extended})$= ϕ(<author; ; $fn_{extended}$, $ln_{extended}$; XSDC>).
=author($\phi(fn_{extended}$, $ln_{extended})$); _), we suppose that XSDC for author is empty,
= author($\phi(fn_{extended})$, $\phi(ln_{extended})$; _), since ϕ is a map function.
The value of $\phi(fn_{extended})$ and $\phi(ln_{extended})$ are obtained by replacing in the previous example author by fn and ln. Thus, we have the following expressions:
$\phi(fn_{extended})$ = fn(<value; varchar; _ >;VC(fn)) and
$\phi(ln_{extended})$ = ln(<value; varchar; _ >;VC(ln)).
If we replace ϕ(fn) and ϕ(ln) by their values computed earlier, we obtain
$\phi(author_{extended})$=author(fn(<value;varchar;_>;VC(fn)), ln(<value; varchar; _ >;VC(ln)); _).
Hence paper is a UDT with two attributes: fn and ln. Each of them is a UDT with an attribute named value and constraints returned by the function VC.
More generally, the computation of $\phi(D_{extended})$ is given by the following algorithm:

Algorithm
UDT_Attribute_From_Extended_Content_Model;
Input: $D_{extended}$, a model of content of an XDTD element E;
Output: $\phi(D_{extended})$, list of UDT attributes;
Begin
Loop
select an arbitrary ϕ(v) in ϕ ($D_{extended}$) with v different to E;
If (ϕ(v) is not in v (to avoid recursion)) then
Calculate ϕ(v) using the grammar $\phi(G_{extended})$ and algorithm in <!ELEMENT journal (volume+)>
<!ATTLIST journal id ID #REQUIRED category CDATA #IMPLIED >
<!ELEMENT volume (issue+)>
<!ELEMENT issue (paper+)>
<!ELEMENT paper (title, author)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
Listing 1;
End If;
If (there is no ϕ(v) in $\phi(D_{extended})$) or (each ϕ(v) in $\phi(D_{extended})$ is in v /*case of recursion*/ or $\phi(v)=\phi(E_{extended})$) then

```
  Exit; /*to leave loop*/
 End If;
End Loop;
End;/*end of algorithm*/
```

Listing 14. Algorithm of computation of UDT Attributes

Example of the computation with recursion

To show the behavior of ϕ in the case of recursion, we propose the following example:

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

where the element "cite" represents the papers that appear in the references of the paper.
By definition, we have
$paper_{extended}$ = <paper; ; $D_{extended}$; XSDC>.
Since XSDC is empty (we suppose that paper has no XSD constraints), this equality becomes,
$paper_{extended}$ = <paper; ; $D_{extended}$; _>.
If we apply ϕ to this equality, we obtain
$\phi(paper_{extended}) = \phi(<paper; ; D_{extended}; _>)$ then

$\phi(paper_{extended}) = paper(\phi(D_{extended}); _)$

Fig.26 The value of $paper_{extended}$

That needs the value of $\phi(D_{extended})$ that we will find.
We have
D = (title, author, [cite])
So
$D_{extended}$ = ($title_{extended}$, $author_{extended}$, [$cite_{extended}$]), since "_" and "extended" are map functions.
If we apply ϕ to the last equality, we obtain
$\phi(D_{extended}) = \phi(title_{extended}, author_{extended}, [cite_{extended}])$
$= \phi(title_{extended}), \phi(author_{extended}), \phi([cite_{extended}])$
Since ϕ is a map function, we can write
$\phi(D_{extended}) = \phi(title_{extended}), \phi(author_{extended}), \phi([cite_{extended}])$
The computing of $\phi(title_{extended})$ and $\phi(author_{extended})$ are similar to $\phi(fn_{extended})$. This allows us to write the following expressions:
$\phi(title_{extended})$ = title(<value; varchar; _ >;VC(title)) and
$\phi(author_{extended})$ = author(<value; varchar; _ >;VC(author)).
As to the computing of $\phi([cite_{extended}])$, we will do this as follows.
Using the rule (g) in the grammar $\phi(G_{extended})$ (see **Fig.24**), We have
$\phi([cite_{extended}] = [\phi(cite_{extended})]$
In addition, from the expression <!ELEMENT cite (paper*)>, we can write:
$cite_{extended}$ = <cite; ;{$paper_{extended}$}>.
If we apply the function ϕ to the above equality, we get
$\phi(cite_{extended}) = \phi(<cite; ;\{paper_{extended}\}>)$
$= cite(\phi(\{paper_{extended}\}))$
$= cite(\{\phi(paper_{extended})\})$.
Then, if we replace $\phi(title_{extended})$, $\phi(author_{extended})$ and $\phi([cite_{extended}]$ with their values in Fig.**26**, we obtain
$\phi(paper_{extended}) = paper(\phi(D_{extended}))$
= paper(title(<value; varchar; _ >;VC(title)),
author(<value; varchar; _ >;VC(author)),
[cite({ϕ($paper_{extended}$)})]).
The process stops there because there is no ϕ(v), with v different to $paper_{extended}$, in $\phi(D_{extended})$.
In the next paragraph, we explain how are created the UDT from XSD.

Creation of UDT attribute

In this section, we describe how are generated the UDT attribute of ORM from their antecedents XDTD. This making employs the recursive function CreateAttribute(attr( )) to create the attributes of the UDT. We give in detail, in the next subsections, how this function works.

The aim of the function CreateAttribute is to transform the items delimited by symbols: ( ), [ ], { } and < > into UDT attributes. For ease, we decompose it into functions each dealing with one of the pairs of these symbols. Below, we give the definition of these sub-functions.

Transformation of simple items

The processing of simple items, i.e. items of type "<…>", is done by the algorithm 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 correspond to each item of listOfItems;
4)  end if;
5) return <"attr"; attr; constraints + XSDC>; // "attr" is a
   UDT attribute:
6) end if;
end; // end of simpleItems
```

Listing 15. simpleItems algorithm

For an example that uses this algorithm, consider the expression :
title(<value; varchar; _ >;VC(title))
Since title contains only items that match " <…>", the call of simpleItems(title(<value; varchar; _ >;VC(title))) creates a UDT named title with one attribute named value and returns a UDT attribute defined by : <"title"; title; VC(title)>.

Transformation of items of type e( ) and recursion

In the following listing, we show the definition of the algorithm "recursion" that deals with items of type e( ), direct recursion and mutual recursion.

```
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 CreateAttribute(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 16. recursion algorithm

As an application of this algorithm, let's find CreateAttribute(author(…)) .
We have
$\phi(author_{extended})$=author(fn(<value; varchar; VC(fn)>),
ln(<value; varchar; VC(ln)>).
In this expression, author has items (fn and ln) that match "e(…)". In this case, to have CreateAttribute(author(…)), we use recursion(author(…)) and we get
<"fn"; fn; VC(fn), > (obtained by CreateAttribute(fn(…)))
and
<"ln"; ln; VC(ln), > (obtained by CreateAttribute(ln(…))).
After this substitution, author becomes
author(<"fn"; fn; VC(fn)>, <"ln"; ln; VC(ln)>).
Then, we call simpleItems to
author(<"fn"; fn; VC(fn)>, <"ln"; ln; VC(ln)>),
and we get an attribute defined by <"author", author, _>.

Transformation of items of type {x( )}

The body of the function that deals with items of type{x( )} representing the closure without ϕ is given below. It transforms these items into UDT attributes.

```
Algorithm closure_without_ϕ(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 XDTD element*/
    2) y = createAttribute(x(…)) ;
    3) create a multiset 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 + XSDC>;
    5) end loop;
end; //end of closure_without_ϕ
```

Listing 17. closure_without_ϕ algorithm

The processing of the closure without ϕ can be illustrated by the example of the Listing 7.
We have from
Fig.**17**:
$authors_{extended}$ =<authors; {$author_{extended}$}; SC(authors)>
By applying ϕ to this equality, we obtain the followings equalities
$\phi(authors_{extended})$=ϕ(<authors;{$author_{extended}$};SC(authors)>
)
= authors({ϕ($author_{extended}$)}; SC(authors))
=authors({ϕ(<author;;#PCDATA;VC(author)>)};SC(authors)
)
= authors({author(ϕ(#PCDATA); XSDC)}; SC(authors))
=authors({author(<value; varchar; _>; XSDC)}; SC(authors)).
If we apply the algorithm closure_without_ϕ to the last obtained authors, we get the expression:
authors(<"authors"; authors; _>; constraints)
where "authors" is an attribute of type authors which is a multiset type, and "constraints" comprises constraints on author and authors.

Transformation of items of type {φ(x)}

The goal of this transformation is to delete the symbols "{", "}" and ϕ. The following listing gives the body of the algorithm that does this work.

```
Algorithm closure_with_ϕ(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 multiset type xs (name of x concatenated to 's')
 based on the reference of the UDT x: "ref x";
6)   replace in attr {ϕ(x)} by <"xs"; xs; constraints_on_x + XSDC >;
7)  end loop;
end; //end of closure_with_ϕ
```

Listing 18. closure_with_ϕ algorithm

Let consider the following expression to see how this algorithm works:
cite({ϕ($paper_{extended}$)}).
To transform this expression, CreateAttribute calls the algorithm "closure_with_ϕ" to delete symbols "{" , "}" and ϕ by executing the operations:
1) it creates an incomplete UDT named paper;
2) it creates a multiset type based on "ref paper" named papers; and
3) it replaces {ϕ ($paper_{extended}$)} by <"papers"; papers; constraints_on_paper + XSDC >.

After that, since there is no constraint on paper and papers, we get the expression:
cite(<"papers"; papers; _>)
Thus, we have eliminated symbols "{", "}" and ϕ;

Transformation of items of type […]

The purpose of this transformation is to delete the symbols "[" and "]" which we do by the following algorithm.

```
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 = CreateAttribute(x(…)) ;
    3)    add to y a null constraint;
    4)    replace  [x(…)] in attr by y;
    5)  end loop;
end; //end of optional
```
Listing 19. optional algorithm

The example that illustrates the work of this algorithm is:
[cite(<"papers"; papers; _>)]
To transform [cite(<"papers"; papers; _>)] that results from the previous algorithm, createAttribute use the algorithm "optional" to eliminate symbols '[' and ']' and returns the expression :
<"cite"; cite; null_constraint>
where "cite" is a UDT attribute with type cite.

Processing of named alternative

The algorithm namedAlternative is defined to take in charge the expression that results from the named alternative, e.g. <!ELEMENT a (b|c)>). The following listing is the definition of this algorithm.

```
Algorithm 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 a UDT named “x” if it’s not created;
5)      end loop;
6)      create a UDT called 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 lines between 3 and 5;
     // we call this constraint : c_attr
8)    end if;
9)    return <"attr"; attr; c_attr>;
10) end if;
end; // end of namedAlternative
```
Listing 20. namedAlternative algorithm

As an example of the application of this algorithm, we assume that the element author contains the element address defined by:
<!ELEMENT address (email | phone)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
We have
$\phi(address_{extended})=address(\phi(email_{extended} \mid phone_{extended}))$
then
$\phi(address_{extended})=address(+, \phi(email_{extended}), \phi(phone_{extended}))$
We have also
$\phi(email_{extended})$=email (<value; varchar; XSDC>) and
$\phi(phone_{extended})$=phone(<value; varchar; XSDC>).
then, we obtain the expression
address(+, <"email"; email; XSDC>, <"phone"; phone; XSDC>)
with lines between 6 and 9 in Listing 20, we obtain the expression
<"address"; address; c_address>
where address is UDT with one attribute named value.

Processing of unnamed alternative

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

```
Algorithm unnamedAlternative (attr(listOfItems)) ;
i integer; /* variable for counting the number of attributes that are added in the case of the alternative which has no name (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
          createAttribute(_attr_i(+,…));
   // _attr_i is created for the unnamed alternative
    5)  end loop;
end; // end of unnamedAlternative
```
Listing 21. unnamedAlternative algorithm
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
$\phi(author_{extended})=author(\phi(fn_{extended}),\phi(ln_{extended}), \phi(email_{extended} \mid phone_{extended}))$
then
$\phi(author_{extended})=author(\phi(fn_{extended}), \phi(ln_{extended}), (+, \phi(email_{extended}), (phone_{extended})))$ (see

Fig.24).
If we use lines between 2 and 5 in Listing 21 the expression
(+, ϕ($email_{extended}$),($phone_{extended}$))
becomes successfully
_author_1 (+, <"email"; email; XSDC>, <"phone"; phone; XSDC>)),
and
<"_author_ "; _author_; c_author_1>
where _author_ is a UDT with one attribute named value.

Definition of the function CreateAttribute

Now we give in the following listing the body of the algorithm createAttribute that creates the UDT attribute. It calls the sub-functions that we have previously described.

Algorithm createAttribute(attr(listOfItems)) return UDT attribute;
begin
1) closure_without_ϕ(attr(listOfItems)); //to delete eliminates {}
2) closure_with_ϕ(attr(listOfItems)); //to delete {} and ϕ
3) optional(attr(listOfItems)); //to eliminate […]
4) Loop
5) simpleItems(attr(listOfItems));// to handle items of type < ..>

//case of alternative with named element
6) namedAlternative (attr(listOfItems));
//case of alternative with unnamed element
7) unnamedAlternative(attr(listOfItems));
8) recursion (attr(listOfItems)); // to delete the symbols ( )
9) end loop;

end; //end of createAttribute
Listing 22. CreateAttribute algorithm

Algorithm of translation

The below listing shows the algorithm of translation. It takes a valid XML document with its XSD schema and creates an object-relational schema. The data of the XSD document will be stored in the object table created by the last instruction (at line 4) of the algorithm in Listing 23. The object type of this table is the root element of the XSD document.

This algorithm starts with the computation of $E_{extended}$ what represents the expression of E in XDTD; then we transform this expression into the object-relational schema by using the function ϕ and the function CreateAttribute. The result of this latter will be used to create the object table and its constraints to store the object-relational data.

Algorithm translation;
input: a valid XML document with its XSD schema; Let be E the root of this document;
output: an object-relational schema;
begin
//logical translation
1) Compute $E_{extended}$ = Ψ(E); //see
2) Fig.**15**
3) Compute E(listOfItems) = ϕ($E_{extended}$) ; //see
4) Fig.**21**

//physical translation
5) Compute <"E"; E; Constraints> = CreateAttribute(E(listOfItems));
6) Create an object table named "E_Table" based on the UDT E and constraints "Constraints";

/*"E_Table" is an object table where we store the data of the XML document.*/
end; //end of translation
Listing 23. Translation Algorithm

Then, as we have seen in the last above algorithm, we finish the translation of the schema.

Conceptual schema of translation

In the next figure, we present a schema that recapitulates the steps of the translation.

We first give specifications for XSD, XDTD, and ORM; and then we define two mappings, one from XSD to XDTD and the second from XDTD to ORM. In the last operation, we generate the ORS from the image of the XSD obtained by the composition of Ψ and ϕ.

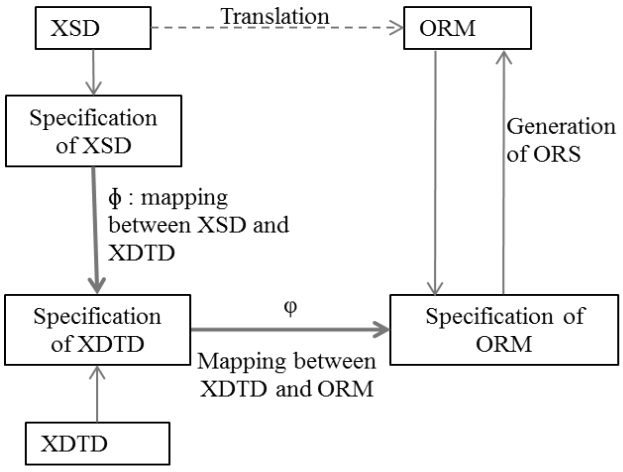

Fig.27 Conceptual schema of translation

Example and test of the translation

We consider, for instance of translation that uses the proposed method, the XSD schema shown in the following figure. It describes the element journal.

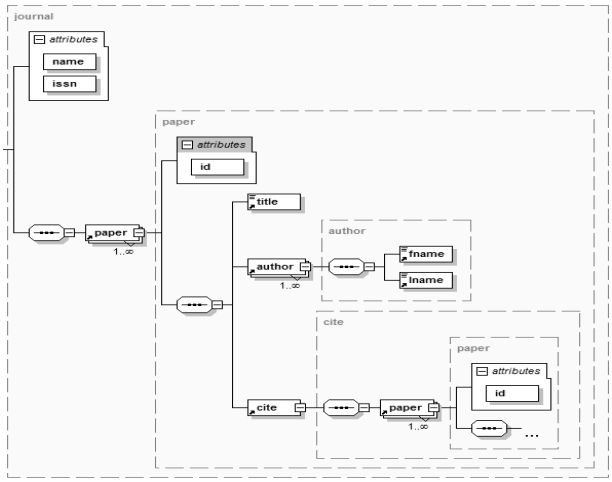

Fig.28 The XSD of the element journal

The following figure shows the ORS associated to the XSD describing the element "journal".

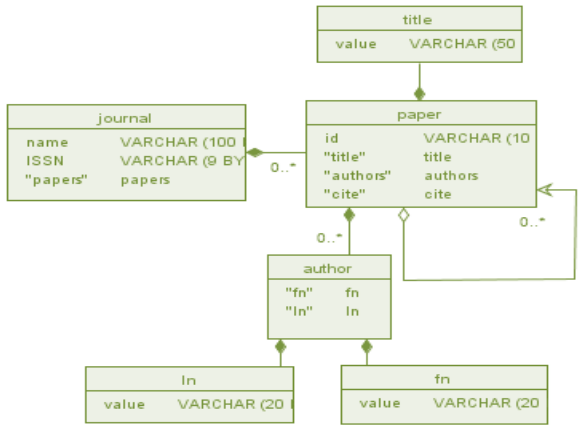

Fig.29 The ORS associated to the XSD describing journal

In the appendix, we give the scripts that create an Oracle physical schema where are stored the object-relational data.

Conclusions

In this paper, we have developed a methodology that translates an XSD data into the object-relational model. We have introduced a novel formalism to handle concepts of XSD and ORS and allow a mapping between them. We have used the ORM of Oracle database to test and validate this method. The most important features of this translations are that it preserves data and schema constraints, realizes a high integration data and is reversible.

In future works, we envisage to take into account more constraints and develop a framework that automatically reverses engineer the XSD schema from the object-relational schema to which is associated. Also, we think to apply this method to translate another XML Schema into ORM.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, (2007), Compilers Principles, Techniques, & Tools, pp. 116-122,159-163

[2] L. Al-Jadir and F. El-Moukaddem, (2002), F2/Xml: Storing Xml Documents in Object Databases, International Conference on Object Oriented Infomation Systems, Montpellier, France,

[3] M. Bahaj and A. Elalami, (2013), The Migration of Data from a Relational Database (Rdb) to an Object Relational (Ordb) Database, Journal of Theoretical and Applied Information Technology, vol. 58,

[4] V. Bisova and K. Richta, (2000), Transformation of Uml Models into Xml, ADBIS-DASFAA Symposium on Advances in Databases and Information Systems, Prague, Czech Republic,

[5] A. Boccalatte, D. Giglio, and M. Paolucci, (1998), An Object-Oriented Modeling Approach Based on Entity-Relationship Diagrams and Petri Nets, IEEE Internal conference on Systems, Man and Cybernetics,San Diego, CA,

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

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

[8] H. Darwen and C. J. Date, (1995), The Third Manfesto, SigMOD Record 24(1), pp. 39-49

[9] C. J. Date, (1998(8)), Preview of the Third Manifesto, Database Programming & Design Journal (San Francisco, CA: Miller Freeman Publications), vol. 11(8),

[10]C. J. Date and H. Darwen, Databases, Types and the Relational Model: The Third Manifesto, 3 ed.: Addison-Wesley, 2007

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

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

[13]A. El Alami and M. Bahaj, (2015), Framework for a Complete Migration from a Relational Database Rdb to an Object Relational Database Ordb, *International Journal of Scientific Engineering and Applied Science (IJSEAS)*, vol. *1*,

[14] A. El Alami and M. Bahaj, (2014), The Road to a Full Migration of Relational Database (Rdb) to Object Relational Database (Ordb): Semantic Enrichment, Target Schema, Data Mapping, *International Journal of Advanced Information Science and Technology (IJAIST)*, vol. *30*,

[15] A. El Alami and M. Bahaj, (2015), Schema and Data Migration of a Relational Database Rdb to the Extensible Markup Language Xml, *World Academy of Science, Engineering and Technology International Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. *9*,

[16] B. Elisa and G. Giovanna, *Object Oriented Databases*: John Wiley & Sons inc, 2008

[17] G.Powell, Beginning Xml Databases*, Indianapolis: wiley Publishing Inc ,2007, PP 131-135.*

[18] J. Hou, Y. Zhang, and Y. Kambayashi, (2001), Object-Oriented Representation for Xml Data, *International Symposium on Cooperative Database Systems for Advanced Applications, Beijing, China*,

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

[20] J. Kim, D. Jeong, and D.-K. Baik, (2009/1), 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
[21] D. Lee, M. Mani, and W. W. Chu, (2003/7), Solving Schema Conversion Problem between Xml and Relational Models: Semantic Approach, *ResearchGate*,
[22] M. Machkour and K. Afdel, (2016), Transforming Xml into Object-Relational Schema *IOSR Journal of Computer Engineering (IOSR-JCE)*, vol. *18(5)*, pp. 40-52
[23] M. Machkour, K. Afdel, and Y. Idrissi Khamlichi, (2014), Conversion Methodology from Hierarchical Model to Object-Relational Model with Structural and Semantic Aspects Preservation, *International Journal OF Mathematics AND Computer Research*, vol. *2(7)*, pp. 503-511
[24] M. Machkour, K. Afdel, and Y. I. Khamlichi, (2016), A Reversible Conversion Methodology between Xml and Object-Relational Models, Proc. *IEEE International Conference on Information and Communication Systems (ICICS)*, IRBID, Jordan
[25] M. Machkour, S. Aminzou, K. Afdel, and Y. I. Khamlichi, (2015), Converting Xml Schema into Object-Relational Model with Data Constraints Preservation, *International Journal of Multidisciplinary and Current Research*, vol. *3*, pp. 523-536
[26] J. Melton, *Advanced Sql:1999: Understanding Object-Relational and Other Advanced Features (the Morgan Kaufmann Series in Data Management Systems)*, 2003
[27] S. Navathe and R. Elmasri, (2011), Fundamentals of Database Systems*, Addison-Wesley*, pp. 353-413
[28] J. Sebastian, *The Art of Xsd* Simple Talk Publishing, 2009
[29] P. Walmsley:, *Definitive Xml Schema*: Prentice Hall, 2001
[30] M. Wang, (2010), Using Object-Relational Database Technology to Solve Problems in Database Development, *Issues in Information Systems*, vol. *XI*,

Appendix

```
Creation of simple object types
create type title as object (value varchar(50))
/
create or replace type lname as object(value varchar(20))
/
create or replace type fname as object(value varchar(20))
/
create or replace type author as object("fname" fname, "lname" lname)
/
Creation of the collection authors
create or replace type authors is table of author not null
/
Creation of the incomplete type "paper" to allow mutual reference between cite and paper
create or replace type paper
/
Creation of the type collection "refpapers" of references to the object paper
create type refpapers is table of ref paper
/
create type cite as object("papers" refpapers)
/
Complete the type paper
create or replace type paper as object
 (id varchar(10),
"title" title,
"authors" authors,
"cite" cite)
/
Creation of the type collection "papers" of paper
create type papers as table of paper not null
/
Creation of the type journal
create type journal as object (name varchar(100), ISSN varchar(9), "papers" papers)
/
Creation of the object table "journals" where we store the object-relational data.
create table journals of journal
(constraint uniqueISSN unique(ISSN),
constraint requiredISSN ISSN  not null,
constraint requiredName name  not null,
constraint patternISSN check (regexp_like(ISSN,'[0-9]{4}\-[0-9]{4}$')))
nested table "papers" store as nested_papers
(nested table "authors" store as nested_authors
nested table "cite"."papers" store as nested_refpapers)
```