Graphical Representation of XML Schema

Flora Dilys Salim[1], Rosanne Price[2], Maria Indrawan[1], and Shonali Krishnaswamy[1]

[1]School of Computer Science and Software Engineering
[2]School of Business Systems
Monash University, Victoria, Australia
floradilys@yahoo.com.au,
[Maria.Indrawan|Shonali.Krishnaswamy|Rosanne.Price]@infotech.monash.edu.au

Abstract. XML is becoming the de-facto standard for exchanging information in distributed applications and services. This has resulted in the development of a large number of XML documents with their associated schemas, such as DTD and XML Schema. A major challenge in using XML Schema is the difficulty in reading and understanding even a relatively small XML Schema because of its textual nature and its XML syntax. In this paper, we present transformations from textual XML Schema to graphical UML to facilitate understanding of XML Schema. Our transformation approach is unique in that we focus on all thirteen building blocks of XML Schema and is based on existing UML notation without introducing new stereotypes that would require additional user training. *Keywords*: XML Schema, UML, transformation, graphical modelling, conceptual, reverse engineering.

1 Introduction

The number and size of XML documents developed based on XML Schema has grown rapidly in the last few years. Maintenance of these XML Schema documents is necessary. Maintenance requires understanding and, given that XML Schema is textual and is described using XML syntax, this becomes tedious. In general, the larger the XML Schema document, the more difficult it is to understand and maintain [Bird00]. Graphical and higher level conceptual languages can be used to clarify XML Schema structure. In particular, graphical modelling languages are a more effective means of detailing and communicating data requirements [Bird00]. As a result, research in the area of graphical and higher-level modelling of XML Schema is becoming increasingly relevant.

Existing work in the area of reverse engineering XML schemas have some limitations, such as:

- they deal with the transformation of XML Schema predecessors (such as SOX, DTD), as in [Booch99b, Mello01],
- they do not use UML (the de-facto standard in Object-Oriented conceptual modelling) as the higher-level model, as in [Mani01, Mello01, Widj02],

J.X. Yu, X. Lin, H. Lu, and Y. Zhang (Eds.): APWeb 2004, LNCS 3007, pp. 234–245, 2004.

- the transformation rules are not comprehensive or are missing completely, as in [Booch99b, Mani01, Rout02b, Widj02],
- the resulting diagrams are not simple and require special training as new UML constructs are created to represent XML constructs, as in [Booch99b, Rout02b],
- they depend on user defined rules and so are not automatic (i.e. existing automation tools), as in [Ratio00, Sun].

The most closely related work, involving XML Schema as a source of transformation and UML as a result of transformation is [Rout02b]. The transformation work is oriented towards reverse engineering; therefore, a new UML construct is created to represent each XML Schema construct. The method used relies on a one-to-one transformation of XML Schema to UML constructs, which has the advantage that most of the information in the original XML Schema document is preserved. However, the disadvantage of this method is that it employs an extensive collection of new constructs that must be learned and complicate the resulting UML diagram. In addition, this research does not cover transformation of all thirteen XML Schema building blocks.

The research presented in this paper aims to: (1) assist understanding and documentation of XML Schema by converting them to graphical form, i.e. UML, and (2) to adopt a transformation approach that does not require additional training beyond standard UML.

In terms of the research goals of the transformation work described here, having clarity in the resulting UML diagram takes precedence over complete reverse engineering (allowing generation of the original XML Schema document from the UML diagram). Reverse engineering an XML Schema requires that everything in the schema, including less important details and processing instructions, be transformed to UML constructs and captured within the resulting diagram. Including less important syntactic details of XML Schema complicates the resulting UML diagrams. On the other hand, total reverse engineering is only needed when the XML Schema is not preserved, which is not the case considered here. The XML Schema is still maintained because XML instance documents need to refer to an XML Schema document to be validated. Therefore, in this paper, maintaining clarity is our focus, rather than preserving completeness in the UML diagram resulting from the transformation process.

Using existing constructs instead of creating new constructs or datatypes have several advantages. Firstly, it simplifies the resultant UML diagram. Secondly, the transformation result can be understood by anyone who knows UML. There is no need to learn special symbols or additional stereotypes. This contrasts with the transformation approach of Routledge et al. [Rout02b]. As existing UML constructs are used to represent the semantics of each given XML Schema construct in our transformation approach, this implies the use of a pattern-based approach as described in [Price00]. In resolving which UML modeling constructs should be used to represent a set of given XML Schema constructs, several alternative UML patterns (UML diagram fragments) are provided with guidelines as to the context in which each alternative should be used. Existing UML constructs are used to specify a UML pattern [Price00], as illustrated in the transformation examples in Section 3. These

examples illustrate representative constructs of XML Schema that are commonly found in a XML Schema document. In this paper, the use of C++ to describe individual constructs is for illustrative purposes. In fact, any commonly understood language, whether programming, natural, or logic-based could be used instead, as specified by [Rumb99]. As stated previously, this approach has the advantage over stereotypes of being immediately understandable and thus not requiring training. Furthermore, the readers of UML are not distracted by an array of special symbols.

This paper is structured as follows. Section 2 presents the overview of transformation rules. Section 3 demonstrates the application of the transformation rules for the primary components of XML. Section 4 concludes the paper and discusses future work.

2 Overview of the Transformation Rules

The XML Schema constructs transformed comprise the thirteen XML building blocks defined in the XML Schema Recommendation [W3C].

In UML, a construct should not be anonymous; however, in XML Schema, it is possible to find anonymous constructs. In such cases, artificial names need to be used for naming UML constructs that are transformed from anonymous XML Schema constructs. For consistency, in generating artificial names, a set of rules should be followed. The naming rules employed in this paper follow the rules described in [Sal03].

Table 1 summarizes the rules of transforming XML Schema constructs to UML. Refer to [Sal03] for full details and examples of transformation of each XML Schema construct. The working definition of each XML Schema construct in this research is based on [Duck01, W3C], whereas the definition of each UML construct is based on [Booch99a, Rumb99b].

Table 1. Summary of Transformation Rules of XML Schema Constructs

XML Schema Constructs		UML Constructs
Namespace		Package
Element declaration	Global element	Class and `{root}` property (if necessary)
	Local element	Attribute
	Occurrence constraints (`minOccurs`, `maxOccurs`)	Multiplicity of attribute
	`fixed` constraint	`{frozen}` constraint
	`default` constraint	`{changeable}` constraint
	`Nillable`	Individual constraint `{can be NULL}`
Attribute declaration	Global attribute	Class
	Local attribute	Attribute
	`fixed` constraint	`{frozen}` constraint
	`default` constraint	`{changeable}` constraint
	`optional` constraint	Multiplicity `[0..1]`
	`required` constraint	Multiplicity `[1..1]`

Simple type definition	Global simple type		Class and `<<type>>` stereotype (if necessary)
	Local simple type		Attribute
	restriction		Attribute
		Child constructs	Individual constraints of the attribute
		Base type	Type of the attribute
		enumeration	C++ `enum` datatype
	`list` datatype		C++ `list` datatype
	`union` datatype		C++ `union` datatype
Complex type definition	`sequence` compositor		C++ `struct` datatype, or `{ordered}` running constraint in attribute compartment
	`choice` compositor		C++ `union` datatype
	`all` compositor		Transform each child element using the rules for element transformation.
	Simple content		Follows simple type transformation rules. Additional XML Schema attributes will be transformed to UML attributes.
	Empty content		Follows the rules for transforming elements of complex type.
	Mixed content		A class with optional artificial string attributes and either a running constraint or a UML note specifying the UML class must contain text or string
	Derived by extension		Subclass with generalization relationship to the class that represents the base type of the XML Schema derived type
	Derived by restriction		A new class
Substitution group			C++ `union`, or classes associated with `{xor}` constraint
Model group			Not transformed. Text substitution rule is used if necessary.
Attribute group			Not transformed. Text substitution rule is used if necessary.
Identity constraint definition			Qualifier on an association between the qualified class and the target class
	`key` element		Qualified class
	`keyref` element		Target class
Notation declaration			Note
Ancillary XML Schema construct			Note

3 Discussion

This section illustrates the application of the transformation rules to selected XML Schema constructs. The constructs include simple type and complex type. We start the discussion by examining the transformation rule for simple type definition.

3.1 Simple Type

A simple type definition is a named set of constraints applicable to a single attribute or a single element having no child elements or attributes [Duck01, W3C]. Simple types may be defined locally within an element or globally as a direct child of a `schema` construct.

The general rules to be followed in transforming simple type definitions are as follows:

- A simple type definition that is defined globally as a direct child of `schema` construct is transformed to a UML class having an artificial attribute.
- A local simple type definition is transformed to a UML attribute.
- The datatype of the resulting UML attribute is either the restriction base type or an appropriate C++ datatype (i.e. depending on the specific constructs used in the simple types).
- Restrictions on simple types may be transformed either to individual constraints applied on the UML attribute or incorporated in the C++ datatype used by the UML attribute.

As described in [Rumb99], any programming language can be used to describe attribute datatypes in UML. C++ is used throughout this paper to describe UML attribute datatypes because it is one of the most widely recognised and established programming languages. Using C++ is different than using stereotypes because there is no need to learn constructs specific to the transformation approach. Any language can be used in UML to describe attribute types, as specified in [Rumb99]. We choose C++ as an example.

Atomic datatypes are simple types whose values cannot be divided. Atomic datatypes are typically used in element declarations or attribute declarations. Atomic datatypes include built-in datatypes, which are primitive, and derived datatypes of XML schema previously defined by W3C [Duck01, W3C]. An atomic datatype that is used within an element declaration will be transformed to UML as follows. The XML element declaration will be represented as an UML attribute whose type matches that of the nested XML atomic datatype.

For example, the declaration of global element `Name` uses xs:string, a built-in primitive datatype that is also an atomic datatype. The element will be transformed to a class since it is global. An artificial attribute is created with the atomic datatype as the type of the UML attribute (Figure 1).

```
<xs:element name = "Name" type = "xs:string" />
```

Code Listing 1. Name global element declaration

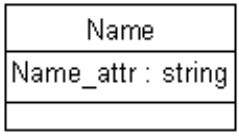

Fig. 1. Transformation of atomic datatype used by a global element

If the above element is a local declaration within a `Person` element, it will be transformed to a UML attribute within the `Person` class (Figure 2)

```
<xs:element name = "Person" >
    <xs:complexType>
        <xs:sequence>
            <xs:element name = "Name" type =
            "xs:string" />
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Code Listing 2. Name local element declaration

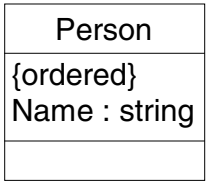

Fig. 2. Transformation of atomic datatype used by a local element

In addition to existing simple types, there are user-defined atomic datatypes that can be derived from built-in datatypes by applying restrictions. There are several constraining features that are applicable to any suitable derived datatype, which are: `length`, `minLength`, `maxLength`, `whitespace`, `pattern`, `enumeration`, `minExclusive`, `maxExclusive`, `minInclusive`, `maxInclusive`, `totalDigits`, `fractionDigits` [Duck01].

The general transformation rule for a user-defined atomic datatype definition are to apply an individual constraint in the resultant class and to convert the base type of the XML Schema simple type definition to be the datatype of the UML attribute. If the simple type definition is a global definition, it will be converted to a class with an artificial UML attribute with an individual constraint and with type the same as the base type of the original XML Schema simple type definition. Otherwise, if the simple type definition is not global (i.e. converted from an XML Schema element or attribute having local atomic-datatype definition) it will only be mapped to an individual constraint applied to the UML attribute with type the same as the base type of the original XML Schema simple type definition.

Next, complex type transformation is discussed. A *complex type definition* is a set of element and/or attribute declarations or a content type specification that is applicable to an element. A complex type can be named or anonymous. An anonymous complex type is a local complex type definition that is nested within an element declaration. A named complex type is a global complex type definition that has a name, which then can be used by any element declaration [Duck01, W3C].

3.2 Complex Type

A complex type definition needs one or more compositors. A *compositor* is an XML Schema construct that creates a set of element declarations and provides rules for all elements declared within it. There are 3 types of compositors, which are `sequence`, `choice`, and `all` [Duck01, W3C]. A `sequence` construct is a compositor that constrains the elements in the instance documents to appear in the order in which they are declared in the schema. A `choice` construct is a compositor whose constraint is that only one immediate child element can appear in the instance document, unless if the maximum occurrence is specified to be more than one. An `all` construct is a compositor that allows the immediate child elements to appear in any order with multiplicity of either 0 or 1 [W3C]. Due to paper size limitation, we illustrate only of the three possible complex type compositors.

A `choice` construct is a compositor that enforces the constraint that only one immediate child element can appear in the instance document, unless if the maximum occurrence is specified to be more than one [Duck01]. The `choice` compositor is transformed to `union` datatype (as in C++); likewise, every `choice` compositor is transformed to an artificial UML attribute of `union` datatype (as in C++). Each child element of the choice compositor is transformed to a C++ variable that is initialised with the child element's type as its type. All these variables are the members of the C++ `union`.

For example, the contact details of a student may either be a mailing address, a phone number, or an email address. A student may have more than one contact details. The following is the schema (Code Listing 3) and its transformation result in UML (Figure 3)

```
<xs:element name="Student">
   <xs:complexType>
       <xs:sequence>
           <xs:element name = "Name" type="NameType" />
              <xs:choice>
                  <xs:element name = "MailingAddress">
                     <xs:complexType>
                         <xs:sequence>
                             <xs:element name ="Street" type =
                                                  "xs:string" />
                             <xs:element name ="Suburb" type =
                                                  "xs:string" />
                             <xs:element name ="PostCode" type =
                                                  "PostCode_AUS" />
                         </xs:sequence>
                      </xs:complexType>
                  </xs:element>
                  <xs:element name ="EmailAddress" type ="xs:string" />
             </xs:choice>
           <xs:element ref = "Phone" />
           <xs:element name = "DateOfBirth" type = "xs:date" />
        </xs:sequence>
        <xs:attribute name = "StudentID" type = "xs:NMTOKEN" />
   </xs:complexType>
</xs:element>
```

Code Listing 3. Complex type with nested compositors

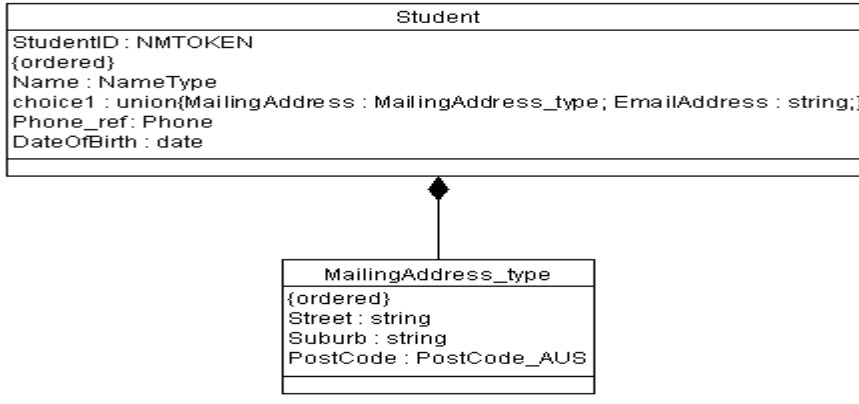

Fig. 3. Transformation of complex type with nested compositors

The following is the code listing of a full XML Schema (Code Listing 4) and the transformation result (Figure 4). The association between instances of `Student` and `NameType`, and instances of `Student` and `Phone`, can be represented in the diagram either by an association line between the two classes concerned, or by the use of class-valued attributes (i.e. an attribute whose domain is a class) [Rumb99]. We chose to use class-valued attributes for illustrative purposes

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name = "Enrollment">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref = "Student" />
                <xs:element name = "CourseID" type = "xs:string" />
                <xs:element name = "StudyMode" type =
               "studyModeType"
                                default = "FullTime" />
                <xs:element name = "DateEnrolled" type = "xs:date"
               />
            </xs:sequence>
            <xs:attribute name = "StudentID" type = "xs:NMTOKEN" />
        </xs:complexType>
        <xs:keyref name = "RefEnrollmentToStudent" refer =
        "StudentKey">
            <xs:selector xpath = "." />
            <xs:field xpath = "@StudentID" />
        </xs:keyref>
   </xs:element>
   <xs:element name="Student">
        <xs:complexType>
            <xs:sequence>
                <xs:element name = "Name" type = "NameType" />
                     <xs:choice>
                          <xs:element name = "MailingAddress">
                               <xs:complexType>
                                      <xs:sequence>
                                           <xs:element name =
                                           "Street"
                                                 type = "xs:string"
                                           />
                                           <xs:element name =
                                           "Suburb"
                                                 type = "xs:string"
                                           />
                                           <xs:element name =
                                           "PostCode"
                                                 type =
                                           "PostCode_AUS" />
                                      </xs:sequence>
                               </xs:complexType>
                          </xs:element>
                          <xs:element name = "EmailAddress" type =
                                "xs:string" />
                     </xs:choice>
                <xs:element ref = "Phone" />
                <xs:element name = "DateOfBirth" type = "xs:date"
                />
            </xs:sequence>
            <xs:attribute name = "StudentID" type = "xs:NMTOKEN" />
        </xs:complexType>
        <xs:key name = "StudentKey">
              <xs:selector xpath = "." />
               <xs:field xpath = "@StudentID" />
        </xs:key>
   </xs:element>
   <xs:complexType name = "NameType">
       <xs:sequence>
          <xs:element name = "GivenName" type = "xs:string" />
```

```
          <xs:element name = "Initial" type = "xs:string"
                minOccurs = "0" />
          <xs:element name = "FamilyName" type = "xs:string" />
       </xs:sequence>
   </xs:complexType>
   <xs:element name = "Phone">
       <xs:simpleType>
           <xs:restriction base = "xs:integer" >
                <xs:length value =  "11" />
           </xs:restriction>

       </xs:simpleType>
   </xs:element>
   <xs:simpleType name = "PostCode_AUS" >
        <xs:restriction base = "xs:integer" >
             <xs:length value="4" />
        </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name = "studyModeType">
        <xs:restriction base = "xs:string">
             <xs:enumeration value = "FullTime" />
             <xs:enumeration value = "PartTime" />
             <xs:enumeration value = "DistanceEducation" />
          </xs:restriction>
   </xs:simpleType>
</xs:schema>
```

Code Listing 4. The Whole XML Schema Document

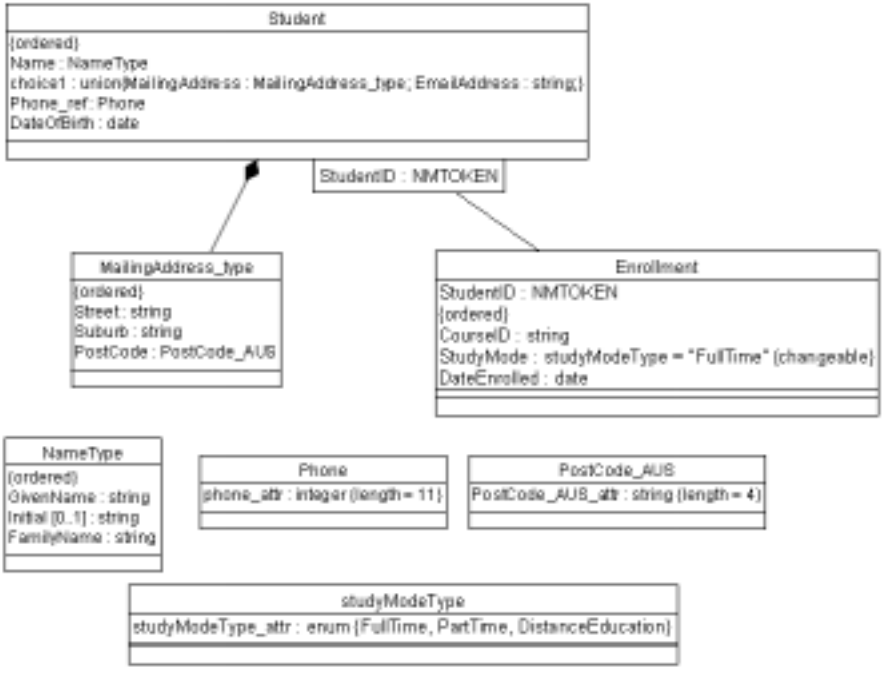

Fig. 4. The Whole UML Diagram

4 Conclusion and Future Work

A major challenge in using XML Schema is the difficulty in reading and understanding even a relatively small XML Schema because of its textual nature and its XML syntax. Therefore, to address this issue, we have aimed to, firstly, assist understanding and documentation of XML Schema by converting them to graphical form, i.e. UML, and, secondly, adopt a transformation approach that does not require additional training beyond standard UML.

We have presented a summary of rules for transforming constructs of XML Schema to UML patterns. The primary contributions of this research are as follows:

The transformation work includes all thirteen building blocks of the XML Schema.

The UML constructs used to form a UML pattern are predefined in UML. No new UML constructs are created. This makes the resulting UML diagram simpler and generally understandable (i.e. not requiring special training beyond that required for understanding standard UML).

There are several future directions proposed following from the work in this paper. One could be case studies or user testing to evaluate the degree of improved understanding using the graphical UML transformation versus the original textual XML schema. Other future work includes implementation of an automated transformation tool based on the transformation rules proposed in this research. Another direction would be to develop rules for mapping XML Schema datatypes to other datatypes in one or more programming languages. For example, one could map XML Schema datatypes to equivalent Java or C++ datatypes. We also intend to investigate the feasibility of extending this approach to support reverse engineering.

References

[Bird00] L. Bird, A. Goodchild, and T. Halpin. "Object Role Modeling and XML-Schema". In *19th International Conference on Conceptual Modeling (ER'2000)*, Salt Lake City, USA, October 2000, Springer, pp. 309-322.

[Booch99a] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, M.A., 1999.

[Booch99b] G. Booch, M. Christerson, M. Fuchs, and J. Koistinen. *UML for XML Schema Mapping Specification.* Rational Software and CommerceOne, 1999. Available at: http://www.rational.com/media/uml/resources/media/uml_xmlschema33.pdf

[Duck01] J. Duckett, O. Griffin, S. Mohr, F. Norton, I. Stokes-Rees, K. Williams, K. Cagle, N. Ozu, J. Tennison. *Professional XML Schemas*, Wrox Press Ltd, Birmingham, U.K., 2001.

[Mani01] M. Mani, D. Lee, and R.R. Muntz. "Semantic Data Modeling using XML Schemas". In *Proceedings. 20th International Conference on Conceptual Modeling (ER 2001)*, Yokohama, Japan, November 2001, Springer, pp. 149-163.

[Mello01] R.d.S. Mello and C.A. Heuser. "A Rule-Based Conversion of a DTD to a Conceptual Schema". In *Proceedings. 20th International Conference on Conceptual Modeling (ER 2001)*, Yokohama, Japan, November 2001, Springer, pp. 133-148.

[Price00] Price, R., N. Tryfona, and C.S. Jensen. *Extended Spatiotemporal UML: Motivations, Requirements, and Constructs*, Journal of Database Management, Vol. 11, Issue 4, Oct-Dec 2000, pp. 14-27.

[Ratio00] Rational Software Corporation. "Migrating from XML DTD to XML Schema using UML". *Rational Software White Paper*, 2000. Available at: http://www.rational.com/media/whitepapers/TP189draft.pdf

[Rout02a] N. Routledge, L. Bird, and A. Goodchild. "UML and XML Schema". In *Proc. Thirteenth Australasian Database Conference (ADC 2002), Melbourne, Australia, ACS*.

[Rout02b] N. Routledge, A. Goodchild, and L. Bird. *XML Schema Profile Definition*. Honours thesis extract, DSTC, Queensland, Australia, 2002. Available at: http://titanium.dstc.edu.au/papers/xml-schema-profile.pdf

[Rumb99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*, Addison Wesley, 1999.

[Sal03] F.D. Salim. *UML Representation of XML Schemas*. Honours thesis, Monash University, Australia, 2003.

[Sun] Sun Corporation. *Java™ Architecture for XML Binding (JAXB)*. Available at: http://java.sun.com/xml/jaxb/index.html

[W3C] W3C. *XML Schema*. Available at: http://www.w3.org/XML/Schema.

[Widj02] N. D. Widjaya, D. Taniar, J.W. Rahayu, and E. Pardede. "Association Relationship Transformation of XML Schemas to Object-Relational Database", *Proceedings of* IIWAS2002, Bandung, Indonesia, 2002.