UML and XML Schema

Nicholas Routledge
School of Information Technology and Electrical Engineering
University of Queensland, QLD, 4072, AUSTRALIA
nickr@dstc.edu.au

Linda Bird and Andrew Goodchild
Distributed Systems Technology Centre (DSTC)
University of Queensland, QLD, 4072, AUSTRALIA
[bird,andrewg]@dstc.edu.au

Abstract

XML is rapidly becoming the standard method for sending information across the Internet. XML Schema, since its elevation to W3C Recommendation on the 2nd May 2001, is fast becoming the preferred means of describing structured XML data. However, until recently, there has been no effective means of graphically designing XML Schemas without exposing designers to low-level implementation issues. Bird, Goodchild and Halpin (2000) proposed a method to address this shortfall using the 'Object Role Modelling' conceptual language to generate XML Schemas.

This paper seeks to build on this approach by defining a mapping between the Unified Modeling Language (UML) class diagrams and XML Schema using the traditional three level database design approach (ie. using conceptual, logical and physical design levels). In our approach, the conceptual level is represented using standard UML class notation, annotated with a few additional conceptual constraints, the logical level is represented in UML, using a set of UML stereotypes, and the XML Schema itself represents the physical level. The goal of this three level design methodology is to allow conceptual level UML class models to be automatically mapped into the logical level, while minimizing redundancy and maximizing connectivity.

Keywords: UML, XML Schema, DTD, XML

1 Introduction

The eXtensible Markup Language (XML) (W3C 2000) is rapidly becoming the premier method for exchanging information across the Internet. The Document Type Definition (DTD) language, which has traditionally been the most common method for describing the structure of XML instance documents, lacks enough expressive power to properly describe highly structured data. XML Schema (W3C 2001), on the other hand, provides a much richer set of structures, types and constraints for describing data and is therefore expected to soon become the most common method for defining and validating highly structured XML documents.

A number of software packages currently exist, which are capable of designing DTDs and XML Schemas using a graphical, tree-based approach. This allows the user to visualize the schema more effectively than is possible with traditional text-based editors. The better-known packages available are XML Spy and XML Authority[2].

The disadvantage of a tree-based approach is that the user is still exposed to some low-level implementation issues. A better approach would be to allow the user to generate the physical data structures, by first designing the appropriate, conceptual domain model. One such approach described by Bird, Goodchild and Halpin (2000), uses an Object Role Modeling conceptual diagram to algorithmically generate an XML Schema. This paper aims to build upon this approach by providing a technique for modeling XML Schemas using Unified Modeling Language (UML) class diagrams.

UML was chosen because it is a popular method for designing software and has proven to be valuable for data modelling. Another benefit of UML is that it is extensible, using stereotypes in a UML Profile. This means that a method for designing XML Schemas can be developed, which is compatible with existing UML tools. It has been claimed by Booch, Christerson, Fuchs and Koistinen (1998), that this method can also unify a development team, by allowing XML developers to work more closely with other team members by sharing a common design platform.

A number of approaches to relating XML Schemas and UML have been described in other works (Booch, Christerson, Fuchs and Koistinen 1998, Rational 2000, Conrad, Scheffner and Freytag 2000). However, there are a number of problems with these existing solutions. One such approach (Booch, Christerson, Fuchs and Koistinen 1998), is somewhat out of date, as it describes a UML profile for SOX (a forerunner to XML Schema). Other papers (Rational 2000, Conrad, Scheffner and Freytag 2000) describe a UML profile for DTDs, and therefore miss many of the additional structures and constraints, which can be found in XML Schema. Finally, a common problem with all these works is that they do not separate the conceptual level of the model from the logical level of the model – forcing analysts to make implementation decisions about the structure of the XML Schema too early in the design process.

[2] http://www.xmlspy.com ; http://www.extensibility.com/

In this paper, we explore an approach in which conceptual level UML class diagrams are transformed through successive steps into XML Schemas. In section 2, we briefly describe our three level design approach using a small example. Section 3 describes the UML profile for XML schema used at the logical level, in more detail. In section 4, we describe a general algorithm for transforming a conceptual level UML diagram into a logical level UML diagram. Finally, conclusions and future work are described in section 5.

2 Three Level Design Approach Example

In this section, we describe our three level XML schema design approach, using a small example. In particular, we demonstrate how a small university student-rating system can be modelled using a conceptual level UML diagram, transformed into a logical level UML diagram based on an XML schema profile, and mapped into a physical level XML schema. Figure 1 summarises this design process.

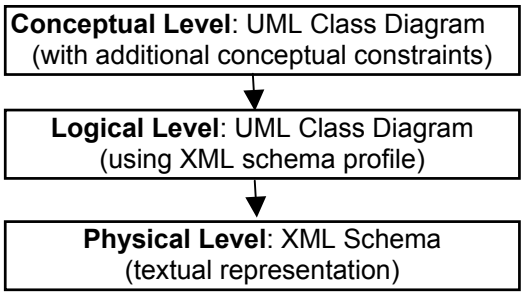

Figure 1: Three level design approach

As mentioned, the Universe of Discourse (UoD) being modeled in XML schema is a university student-rating system (as shown in the following two output reports).

Subject	*Title*	*Year*	*NrEnrolled*	*Lecturer*
CS100	Intro to Computer Science	1982	200	P.L. Cook
CS121	Software Engineering	1982	150	L.P. Green
CS100	Intro to Computer Science	1983	250	A.B. White

Table 1: First Subject table for University UoD

Subject	*Year*	*Rating*	*NrStudents*	*%*
CS100	1982	7	10	5.00
		6	10	5.00
		5	75	37.50
		4	80	40.00
CS121	1982	7	4	2.67
		6	8	5.33
		5	60	40.00
		4	70	46.67
		1	8	5.33
CS100	1983	7	15	4.00
		6	30	12.00
		5	100	40.00
		4	80	32.00
		3	30	12.00

Table 2: Second Subject table for University UoD

Given this example, our goal is to produce an XML Schema that satisfies all major conceptual integrity constraints that exist, while at the same time has minimal redundancy and maximum connectivity.

2.1 Conceptual Level

The first step, in our proposed approach, is to model the domain using a conceptual level UML class diagram. A conceptual diagram is used to describe the UoD in terms of objects and relationships from the real world. Below, we show a conceptual level UML class diagram, which represents the example, university student-rating UoD.

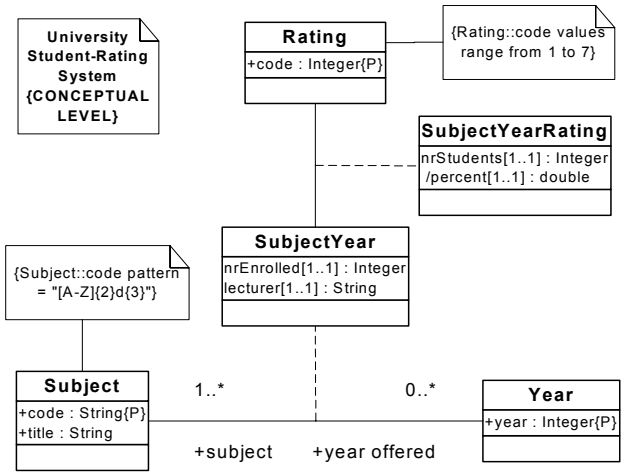

Figure 2: Example conceptual UML class diagram

Figure 2 uses standard UML class diagram notation – for example, classes are shown as rectangles, attributes are listed within the associated class rectangles, and relationships are shown as lines linking two or more classes. Attribute and relationship multiplicity constraints are also represented using standard UML notation. It should be noted, however, that a number of non-standard annotations are also required to represent some common conceptual constraints, such as the primary identification of a class (attributes suffixed with "{P}"). For a more detailed discussion on the use of UML for conceptual data modelling please refer to Halpin (1998).

2.2 Logical Level

Once a conceptual level model has been designed, and validated with the domain expert, it can be used to automatically generate a logical level diagram. A logical level diagram describes the physical data structures in an abstract and often graphical way. In our approach, the logical level model is a direct (ie. one-to-one) representation of the XML Schema data structures. To this end, we represent the logical level as a UML diagram, which uses the stereotypes defined in an XML Schema profile.

In figure 3, we show a logical level UML class diagram, which has been generated from the conceptual level diagram from figure 2. The logical level diagram shown uses stereotypes (such as "element", "complexType", "simpleType", "elt" and "attr") that we have defined within a UML profile for XML Schema (described in more detail in section 3). This allows the logical level

UML diagram to directly capture the components of the physical level XML Schema.

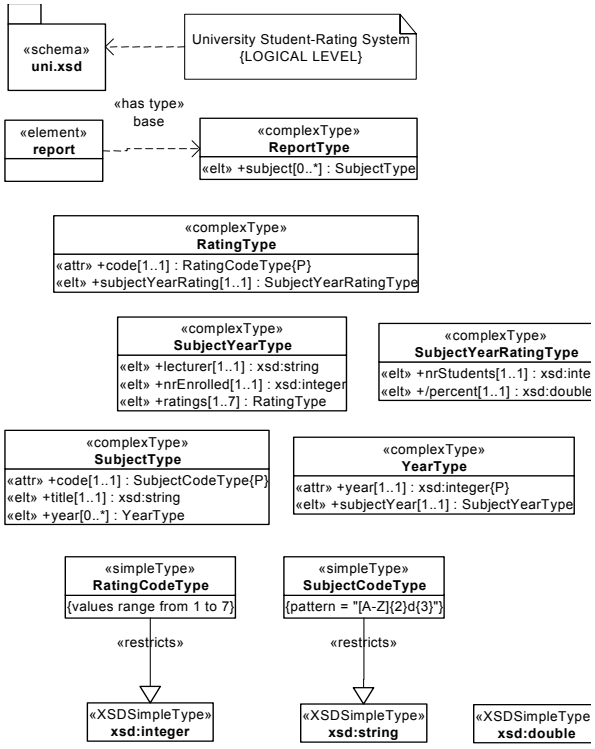

Figure 3: Example logical level UML class diagram

It is important to note that this logical diagram can be automatically generated from the conceptual model using the approach described in section 4. This removes the need for the data designer to be concerned with implementation issues. However, because there are many ways to map a conceptual level model into a logical level model, this transformation should be configurable with design options. Similarly, the data modeller may wish to directly 'tweak' the logical design to (for example) introduce controlled redundancy or make other logical-level design decisions.

2.3 Physical Level

In appendix A, we show a physical level XML Schema, which corresponds directly to the logical level diagram shown in figure 3. A physical level model defines the data structures using the implementation language – in this case XML Schema. The physical schema shown in Appendix A uses the standard textual language defined by the World Wide Web Consortium (W3C) in its March 2001 XML Schema Recommendation (W3C 2001).

2.4 XML Instance

To help the reader understand the logical and physical models in sections 2.2 and 2.3, we show below an example XML instance document. This XML instance, which incorporates some of the information from the output reports shown earlier, correctly satisfies the XML Schema definitions presented in figure 3 and appendix A.

```
<report>
    <subject code="CS100">
        <title>Introduction to Computer Science</title>
            <year year="1982"> <subjectYear>
                <lecturer>P.L.Cook</lecturer>
                <nrEnrolled>200</nrEnrolled>
                <ratings code="7"> <subjectYearRating>
                        <nrStudents>10</nrStudents>
                        <percent>5.00</percent>
                    </subjectYearRating> </ratings>
                <ratings code="6"> <subjectYearRating>
                        <nrStudents>10</nrStudents>
                        <percent>5.00</percent>
                    </subjectYearRating> </ratings>
                <ratings code="5"> <subjectYearRating>
                        <nrStudents>75</nrStudents>
                        <percent>37.50</percent>
                    </subjectYearRating> </ratings>
                <ratings code="4"> <subjectYearRating>
                        <nrStudents>80</nrStudents>
                        <percent>40.00</percent>
                    </subjectYearRating> </ratings>
                <ratings code="3"> <subjectYearRating>
                        <nrStudents>20</nrStudents>
                        <percent>10.00</percent>
                    </subjectYearRating> </ratings>
                <ratings code="2"> <subjectYearRating>
                        <nrStudents>5</nrStudents>
                        <percent>2.50</percent>
                    </subjectYearRating> </ratings>
            </subjectYear> </year> </subject> </report>
```

3 XML Schema Profile for UML

In this section, we outline the XML Schema profile, which we have developed as the basis for logical level UML class diagrams. It is intended that every concept in XML Schema has a corresponding representation in the UML profile (and vice versa). As a result, there is a one-to-one relationship between the logical and physical XML Schema representations.

The following set of diagrams graphically describe the XML Schema UML profile developed by the authors, using standard UML class diagrams. Figure 4 (section 3.1) shows the relationships between XML Schema elements and types; figure 5 (section 3.2) shows the relationships between XML Schema schemas and namespaces, and figures 6, 7 and 8 (section 3.3) show how schemas, content models and types are built from various XML Schema constructs.

3.1 Element-Type Metamodel

The metamodel in figure 4 shows the relationships between XML Schema concepts such as 'element', 'complexType', 'simpleType' and 'XSD simpleTypes' (which represents those primitive types found in the XML Schema namespace). These XML Schema concepts are represented as stereotyped classes, allowing them to be used in logical level UML class diagrams to represent the corresponding XML Schema concept. Two of the relationships between these concepts, namely "restricts", and "extends", are represented as stereotyped specialisations. This was done to allow for instance substitutability between related user-defined types. The relationship "has type" is representated as a stereotyped dependency between an 'element' and either a 'simpleType' or 'complexType'. A dependency is a special type of association in UML, in which the source element is dependent on the target element.

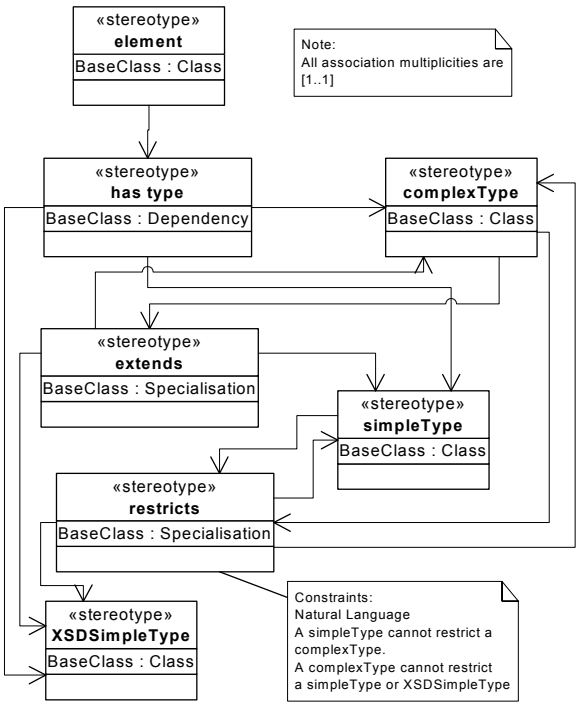

Figure 4: Element-Type metamodel for XML Schema

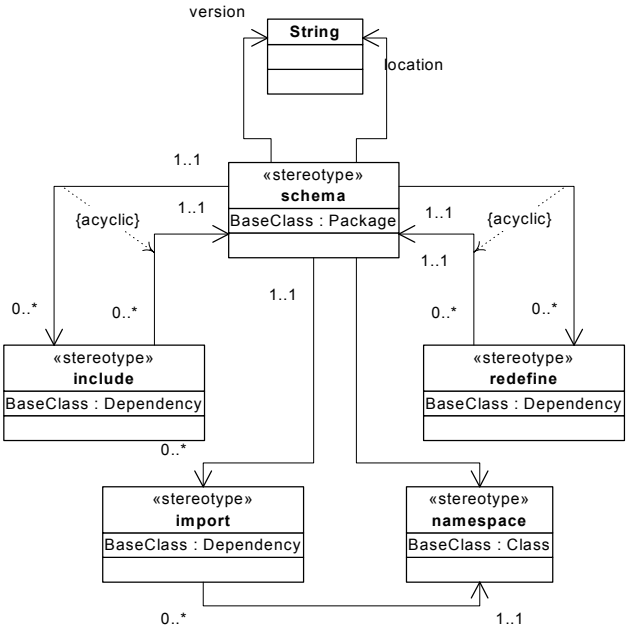

Figure 5: XML Schema-Namespace metamodel

3.2 Schema-Namespace Metamodel

The metamodel in figure 5 shows the relationship between schemas and namespaces in XML Schema. This model introduces the concept of a 'schema' as a stereotyped package. A schema can 'include' or 'redefine' another schema. To indicate this, there are two corresponding 'ring relationships' attached to 'schema'. These relationships are acyclic, because a schema can not include or redefine either itself, or another schema, which includes or redefines itself, etc. Another important stereotyped class in figure 5 is the 'namespace' class. The 'namespace' class is associated with a stereotyped dependency called 'import', which in turn is associated with a 'schema'. This indicates that a 'schema' can 'import' schemas from other 'namespaces'.

3.3 Schemas, Content Models and Types

Figure 6 introduces a number of new stereotyped classes and stereotyped attributes, each of which represents an XML Schema construct used to create the structure of a model (such as 'choice', 'group', 'seq' etc). This diagram shows how XML Schema content models and types are built using UML constructs, according to the XML Schema Part 1: Structures specification (W3C 2001).

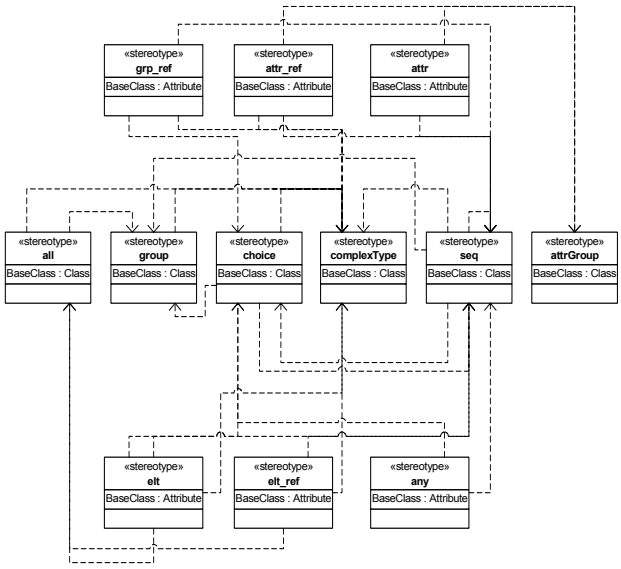

Figure 6: Building content models and complexTypes

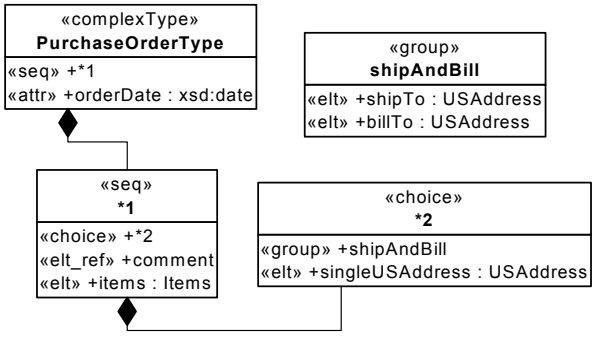

Figure 7: An example XML schema at the logical level

To illustrate how these XML Schema classes are used on a logical level UML diagram, an example is shown in figure 7. This logical level diagram represents a fragment of the 'PurchaseOrder' XML Schema code presented in the XML Schema Part 0: Primer (W3C 2001). Note that the definitions of the types 'Items' and 'USAddress', and the element 'comment' are omitted from this example for the sake of brevity.

The corresponding XML schema code fragment for figure 7 is:

```
<xsd:complexType name="PurchaseOrderType">
    <xsd:sequence>
        <xsd:choice>
            <xsd:group ref="shipAndBill"/>
```

```
            <xsd:element name=”singleUSAddress”
            type=”USAddress”/> </xsd:choice>
        <xsd:element ref=”comment” minOccurs=”0”/>
        <xsd:element name=”items” type=”Items”/>
    </xsd:sequence>
    <xsd:attribute name=”orderDate” type=”xsd:date”/>
</xsd:complexType>
<xsd:group name=”shipAndBill”>
    <xsd:sequence>
        <xsd:element name=”shipTo” type=”USAddress”/>
        <xsd:element name=”billTo” type=”USAddress”/>
    </xsd:sequence> </xsd:group>
```

The logical level diagram in figure 7 highlights several important features of the XML Schema profile, which are not obvious from the metamodels shown. Firstly, the concept of nesting XML schema content models (e.g. a ‘choice’ nested inside a ‘sequence’) is represented at the logical level by introducing separate stereotyped classes, for each nesting level, linked by a ‘composition’ association. In figure 7, this feature was used to link the ‘choice’ class to the ‘seq’ class with a composition association. The direction of the composition association indicates the direction of the nesting, and the ordering of the attributes within each class indicates the ordering of the content models.

Note that the default content model for a complexType is a ‘sequence’. Therefore, when a complexType is mapped from the logical to the physical level, the “<<elt>>” attributes within the ‘complexType’ class are automatically mapped to a sequence of elements within the physical complexType.

Another important feature of the XML Schema profile, is that it needs some way of representing anonymous types and nested content models. In our XML Schema profile, these are represented in the same way that nesting has been described, with an additional naming scheme which ensures uniqueness and the preservation of order. In particular, anonymous types and content models are named by appending a sequential number (indicating order) to an asterix (indicating an anonymous reference).

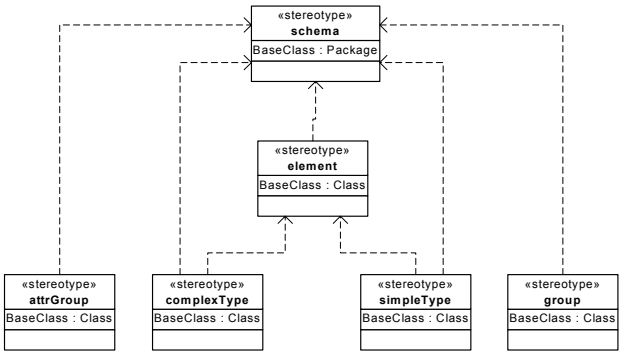

Figure 8: Building a schema

Figure 8 shows how the schema constructs used to create a content model in XML Schema (from figure 6) are related back to a ‘schema’ package.

4 Conceptual to Logical Level Mapping

4.1 Goals

As XML Schemas are hierarchical in nature, generating a logical level model from a conceptual level model requires us to choose one or more conceptual classes to start the XML Schema tree-hierarhcy. One option would be to select a single class as the XML root node, and progressively nest each related class as child elements of the root node. An example of an XML-instance generated by choosing the ‘Rating’ class as the root of the XML hierarchy is:

```
<?xml version=”1.0” encoding=”UTF-8”?>
<report>
    <rating code=”7”>
        <subject code=”cs100”> <year year=”1982”>
                <nrEnrolled>200</nrEnrolled>
                <lecturer>P.L.Cook</lecturer>
                <nrStudents>5</nrStudents>
                <percent>2.50</percent> </year>
        </subject> </rating>
    <rating code=”6”>
        <subject code=”cs100”> <year year=”1982”>…
    </rating>
</report>
```

However, as this example illustrates, this approach can lead to redundant data at the instance level. In this example, the information relating to a subject is repeated for each ‘rating’ that has been given in that subject.

Another approach would be to create a relatively flat schema, in which every class is mapped to a separate element directly under the root node. The attributes and associations of each class would be mapped to sub-elements of these top-level elements. The example below illustrates this approach:

```
<?xml version=”1.0” encoding=”UTF-8”?>
<report>
    <subject code=”cs100”>
        <year> 1982</year>
        <year>1983</year> </subject>
    <subject code=”CS121”>
        <year>1982</year> </subject>
    <year code=”1982”>
        <subject code=”CS100”/>
        <subject code=”CS121”/> </year>
    <year code=”1983”>
        <subject code=”CS100”/> </year>
    … </report>
```

However, as this example illustrates, this approach can lead to disconnected and difficult to read XML instances, which also have some degree of redundancy.

In contrast to these two approaches, the approach presented in this paper aims to minimize redundancy in the XML-instances, while maximising the connectivity of the XML data structures as much as possible. The approach presented in this paper for mapping UML conceptual models into XML Schema is directly based on the one defined by Bird, Goodchild and Halpin (2000), in which Object Role Modeling (ORM) diagrams are mapped into XML Schema. The algorithm described by

Bird, Goodchild and Halpin (2000) is highly suited to our goals, as we have reason to believe that it generates an XML Schema that is in Nested Normal Form (ie. nested XML Schema with no data redundancy).

A number of significant modifications to the algorithm, however, have had to be made to cater for the inherent differences between ORM and UML. In particular, because ORM does not distinguish between classes and attributes (everything in ORM is either an 'object type' or a 'relationship type'), the algorithm described by Bird, Goodchild and Halpin (2000) uses the notion of 'major object types' to determine the first nesting operation. In contrast, however, 'classes' in UML are roughly equivalent to 'major object types' in ORM, and therefore the process used to automatically determine the default 'major object types' is no longer necessary.

A second major point of difference is that the concept of 'anchors', introduced by Bird, Goodchild and Halpin (2000) to identify the most conceptually important player(s) in a relationship type, and to consequently determine the direction of nesting in some cases, are not required in our approach. Instead, we use a closely analogous concept in UML called "navigation". Defining navigation on an association indicates that "given an object at one end, you can easily and directly get to objects at the other end, usually because the source object stores some references to objects of the target" (Booch, Rumbaugh and Jacobson 2000). For this reason, navigation on a UML association tends to point from the more important player in the association towards the less important player (which is the opposite direction to that of 'anchors')

In the remainder of this section, we will describe our general approach to mapping conceptual UML models into logical level XML Schemas.

4.2 Methodology

As discussed earlier, the goal of our mapping approach is to produce an XML Schema, with minimal redundancy and maximum connectivity in the corresponding instance documents. The algorithm, that we have designed to achieve this goal, involves four major steps. Once the logical level class diagram has been generated from the conceptual level one, creating the physical XML Schema is a simple process, due to the direct, one-to-one mapping between the logical and physical levels.

4.2.1 Step 1: Generate Type Definitions

The first step in the methodology is to create type definitions for each attribute and class in the conceptual diagram. The following two rules are used to map attributes to the appropriate logical level types :

1. Attributes, which have additional constraints applied to their primitive types (such as integer and string), map into simpleTypes, which restrict the associated primitive type. For example, in figure 9 'SubjectCodeType' restricts 'string' by adding a pattern constraint.
2. Primitive types, used by an attribute, are mapped into XSD simpleTypes from the XML Schema namespace. For example, the primitive type 'string' maps to xsd:string.

Based on the example conceptual model from figure 2, the following types would be created in this step:

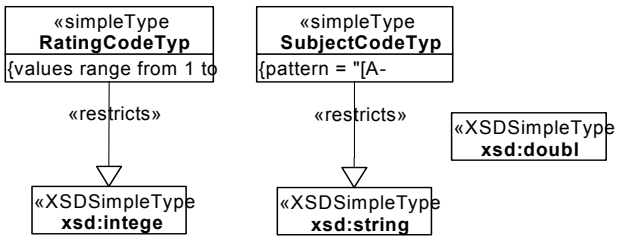

Figure 9: Types created in Step 1

Next, a logical level complex type definition is created for each class at the conceptual level. Each conceptual class is mapped into a complexType, with child elements representing each of its non-primary attributes. Primary key attributes (which are based on simple types) are included in the complexType definition as XML Schema attributes (i.e with a «attr» stereotype). Based on the example from figure 2, the following complexTypes would be created in this step:

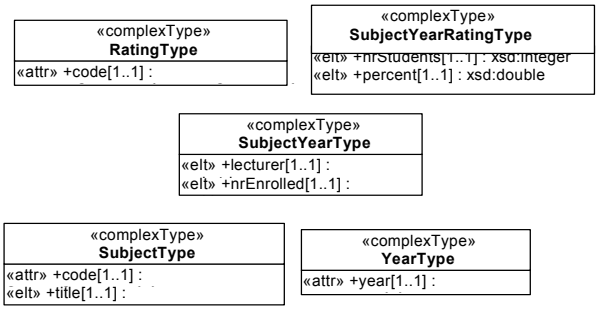

Figure 10: ComplexTypes generated in Step 1

Note that in future steps, some complexTypes may be removed and nestings of child attributes may be performed (including primary key attributes).

4.2.2 Step 2: Determine Class Groupings

The next step is used to determine how best to group and nest the conceptual classes, based on the associations between them. The approach taken is based directly on the approach from Bird, Goodchild and Halpin (2000), in which a combination of 'mandatory-functional' constraints and 'anchors' are used to determine the appropriate nesting choices. In contrast to this, a similar approach based on UML uses 'multiplicity' constraints and 'navigation' to determine an appropriate nesting for the schema.

An approach to automatically determine the default navigation directions is summarised below:

1. If no navigation is defined on an association, then use the multiplicity constraints to determine the navigation direction.

a. If exactly one association end has a minimum multiplicity of 1 (i.e. (1..1) or (1..*)), then define the navigation in the direction of the opposite association end, or
b. If one association end has a smaller maximum multiplicity than the other, (e.g. '0..7' is smaller than '0..*') then navigate towards the end with the smaller maximum multiplicity, or
c. If exactly one class has only one attribute, then navigate towards it.

The nesting is then determined as follows:

1. If exactly one association end has a multiplicity of '(1,1)' (i.e. it is mandatory and functional), then nest the class at the other association end towards it.
2. If both association ends have a multiplicity of (1,1), then nest the classes in the opposite direction to the direction of navigation – i.e. nest the target of the navigation towards the source of the navigation

The reasoning behind this "mandatory-functional" rule was first discussed by Bird, Goodchild and Halpin (2000), and can best be explained using an example.

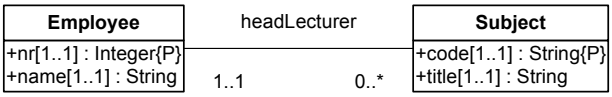

Figure 11: A mandatory, functional relationship

The UML fragment in figure 11 consists of an employee being the head lecturer of many subjects, and each subject having exactly one head lecturer. If the 'headLecturer' was nested towards 'Subject', then the corresponding logical level representation would be:

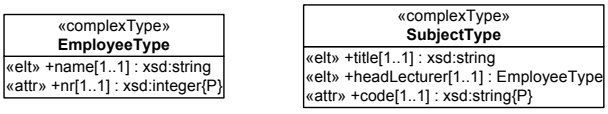

Figure 12: Logical level mapping of figure 11

For example, an instance of the Subject type might look like:

```
<subject code="INFS4201">
    <title>Advanced Distributed Databases</title>
    <headLecturer nr="123456">
        <name>Joe the Lecturer</name> </headLecturer>
</subject>
<subject code="COMP4301">
    <title>Distributed Computing</title>
    <headLecturer nr="123456">
        <name>Joe the Lecturer</name> </headLecturer>
</subject>
```

There are however, a number of problems with this nesting approach. The first issue is redundancy created by repeating employee details with each subject occurrence. This happens because an employee may be the head lecturer of *more than one* subject (according to the UML model in figure 11). The redundancy is clearly evident in the corresponding XML instance where 'Joe the Lecturer' has his details repeated for both the 'INFS4201' and 'COMP4301' subjects.

The other issue arising from the above schema is that not all employees are assigned as head lecturer of a subject (as is indicated on the conceptual level UML model in figure 11). Therefore, a separate global element for employee has to be added to the schema for employees who aren't in charge of any subjects. This is undesirable because it reduces the connectivity of the schema.

The solution to both of these problems is to nest towards the mandatory-functional end of the association. In the example, this would mean nesting the Subject class inside the Employee class, therefore producing the following logical level representation and XML Schema fragment:

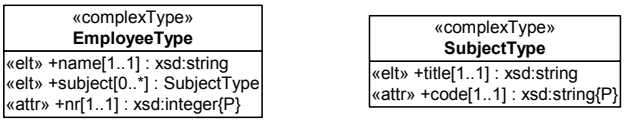

Figure 13: Nesting SubjectType within EmployeeType

This type of nesting is preferable because:

a) The minimum frequency of 1 at the Employee end of the association requires that each Subject be headed by *at least one* Employee, and
b) The maximum frequency of 1 at the Employee end of the association requires that each subject be headed by *at most one* Employee.

The minimum frequency of 1 (uniqueness constraint) is very important in this grouping example because if this constraint didn't apply to a subject (i.e a subject doesn't need a head lecturer), subjects without a head lecturer allocated would not be represented. Similarly, the maximum frequency of 1 (mandatory constraint) is vital because if a subject could be headed by more than 1 lecturer, a subject would be repeated for each corresponding employee thus introducing redundancy into the schema. Therefore, when a class A is associated with exactly 1 instance of class B, class A can be nested inside class B.

Also note that since each association class has an implicit "mandatory, functional" relationship with each of the players of the association (i.e. each association class is related to exactly one object at each end of the association), association classes are always nested towards their association (based on nesting rule 1). These associations are then nested in the opposite direction of the navigation (based on rule 2). An example of nesting the association classes from figure 2 is shown in figure 14.

In this example, the 'SubjectYearRating' association class is nested, together with the 'Rating' class, towards the 'SubjectYear' class. Similarly, the 'SubjectYear' class, together with the 'Year' class, are nested towards the 'Subject' class. Note that the thick-headed arrows in figure 14 represent the direction of nesting, while the thin-headed arrows represent the navigation direction.

The dotted line circles indicate classes grouped for nesting purposes.

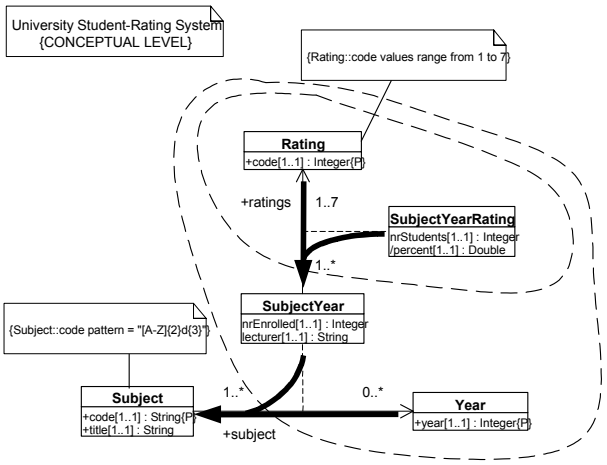

Figure 14: Nesting the conceptual classes.

If navigation cannot be determined between two classes (say classes A and B), or both ends of the association are navigable, then the following option exists:

1. If an association class exists between class A and class B, merge class A and B into the association class.

For example, if we take the 'title' attribute from Subject, and change the multiplicity of Subject's association end to '0..*', navigation cannot be determined. Figure 2, as shown previously, illustrates this:

In this case, navigation cannot be established between Subject and Year because both have optional participation and unbounded maximum frequencies. Also, both classes have only one attribute making rule 1c inapplicable. The solution is to merge Subject and Year with the association class SubjectYear. This merge is valid because for every instance of SubjectYear, there is exactly one instance of the Subject and Year classes.

A final point on the nesting topic is the representation of conceptual level subtypes on the logical level. In our approach, subtype relationships will be carried down to the logical level. Also, a class acting as a supertype for a class or set of classes must not be eliminated from the mapping process.

4.2.3 Step 3: Build the Complex Type Nestings

After the nesting directions have been identified, the next step is to perform the complex type nesting. In the example case study, the 'SubjectYearRatingType' class is nested within the 'RatingType' class. The 'RatingType' class is then nested as an element within the 'SubjectYearType' class. The result of this operation is shown below:

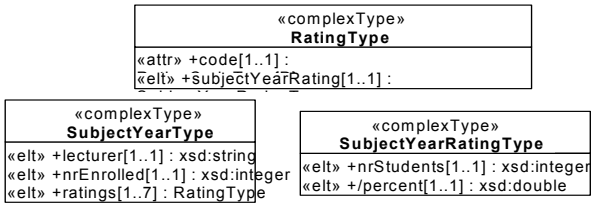

Figure 15: Nesting SubjectYearRatingType within RatingType

It is important to note that when the 'RatingType' complexType is nested, the multiplicity constraint of the resulting element is set to '1..7'. This is because the multiplicity constraint on the association end attached to 'Rating' is '1..7'.

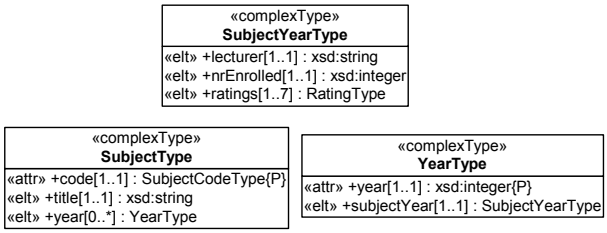

Figure 16: Final nesting of SubjectYearType

In figure 16, the result of the final operation required in the case study is shown, in which the 'SubjectYearType' class is nested within the 'YearType' class, and the 'YearType' class is nested in turn within the 'SubjectType' class.

Note that we have chosen to represent those primary keys, which have a simpleType and a maximum multiplicity of 1, as 'attributes' of the parent complexType. When nesting, primary keys remain an attribute of their respective class after the nesting takes place. The only exception to this rule is when attributes are removed from classes being eliminated from the mapping process. In this case, the attribute will become a primary key of its new parent class. This choice was made to simplify the associated XML instance documents – however, ideally this should be a configurable option.

4.2.4 Step 4: Create a Root Element

Because each XML document must have a root element, a root node is introduced in this step, which represents the conceptual model as a whole. For example, in figure 17, we show that a root element called 'report' was introduced when mapping figure 2 to a logical model. This root element is then associated with a complexType (in this case called 'ReportType'), which represents the set of complexType groupings generated in step 3. In our example, the only complexType grouping is called 'subject'.

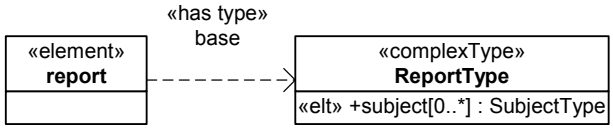

Figure 17: The root node of the schema

4.3 Options and Limitations

4.3.1 Options

A number of options are available when mapping from the conceptual level to the logical level. For example, an attribute from the conceptual level can be represented either as an XML schema attribute or as an XML schema element at the logical level. By default, we have decided to map primary key attributes to XML schema attributes, and non-primary key attributes to XML schema elements. This decision is suitable in the majority of cases, as primary key atributes are usually based on simple types, and have multiplicities of 'exactly one' (as are XML schema attributes).

Other options that may need to be made available to the data modeller, include the introduction of controlled redundancy at the logical level, and the decreasing of the connectivity of the resulting schema.

4.3.2 Limitations

Certain limitations were evident when modelling XML schemas in UML. These are summarised below:

1. UML is aimed at software design rather than data modelling, so some new notation for representing conceptual constraints was required
2. Mixed Content in XML schema is difficult to express in UML without introducing additional non-standard notation to the conceptual level.
3. The UML constraint language, OCL, is syntactically different to the XML schema regular expression language.
4. Unlike UML, XML Schema does not support multiple inheritance. Therefore a conceptual level UML class diagram should not contain classes with more than one supertype.
5. Some constraints represented on the conceptual level such as subtype constraints and acyclic constraints can not be expressed in XML Schema.
6. In situations where navigation is present in both directions, the mapping algorithm must be able to determine the 'stronger' of the two navigations, to determine the most appropriate direction for nesting.

5 Conclusion

This paper proposes a method for designing XML Schemas using the Unified Modeling Language (UML). The UML was chosen primarily because its use is widespread, and growing. Secondly, the UML is extensible, so the new notation being written is totally compatible with existing UML tools.

Presently, there exists a number of tree-based graphical tools for developing schemas. These tools are perfect for small and intuitive schemas, but the more complex the data is, the harder it is for the designer to produce a correct schema. The UML makes it easier to visualize the model, and to ensure that integrity constraints are defined.

The three-level Information Architecture is the fundamental methodology followed by many data modellers. This approach allows the data modeller to begin by focusing on conceptual domain modelling issues rather than implementation issues.

Because each conceptual level model has many possible logical level models, there is a need for a mapping algorithm, which uses sensible data design techniques to translate from one to the other. However, because of the different design choices, which can be made in this mapping process, it would be preferable to allow the designer to choose between common design options. Because there is a one-to-one relationship between the logical and physical levels, however, there is then only one possible mapping to the physical XML Schema itself.

The authors are currently planing to build a prototype tool, which uses the algorithm described in this paper to generate a logical level representation, based on a conceptual level UML class diagram. A prototype tool has been built however, that can generate an XML Schema from a corresponding logical level class diagram (expressed in XMI). In addition to this we also intend on exploring the generation of an XML Schema that is in nested normal form and look at reverse engineering a conceptual level diagram from the physical level.

6 References

BOOCH, G., CHRISTERSON, M., FUCHS, M. and KOISTINEN, J. (1998): UML for XML Schema Mapping Specification. http://www.rational.com/media/uml/resources/media/uml_xmlschema33.pdf

RATIONAL. (2000): Migrating from XML DTD to XML Schema using UML. *Rational Rose Whitepapers*, http://www.rational.com/media/whitepapers/TP189draft.pdf

CONRAD, R., SCHEFFNER D. and FREYTAG, J. C. (2000): XML Conceptual Modeling Using UML. *Proc. International Conceptual Modeling Conference*, Salt Lake City, USA, 558-571, Springer.

BIRD, L., GOODCHILD, A. and HALPIN, T. (2000): Object Role Modeling and XML Schema. *Proc. International Conceptual Modeling Conference*, Salt Lake City, USA, 309-322, Springer.

HALPIN, T. (1998): UML Data Models From An ORM Prospective. Part 1-10. http://www.orm.net/uml_orm.html

W3C (2000): Extensible Markup Language (XML) 1.0. *W3C XML Working Group*, http://www.w3.org/TR/REC-xml

W3C (2001): XML Schema Parts 0-2: [Primer, Structures, Datatypes]. *W3C XML Schema Working Group*, http://www.w3.org/TR/xmlschema-0/, http://www.w3.org/TR/xmlschema-1/ , http://www.w3.org/TR/xmlschema-2/

HALPIN, T. (2001): *Conceptual Schema & Relational Database Design: 3rd edition,* Heidelberg: Springer.

BOOCH, G., RUMBAUGH, J. and JACOBSON, I. (2000): *The Unified Modeling Language User Guide.* Massachusetts: Addison-Wesley.

7 Appendix A: XML Schema for Example

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
elementFormDefault ="qualified">
<!--====== Root Element Definition ===============-->
<xsd:element name="report" type="ReportType">
    <xsd:annotation>
        <xsd:documentation>Displays information regarding
                subject grade averages
        </xsd:documentation> </xsd:annotation>
<!--  Key and Uniqueness constraints  -->
<xsd:key name="pRatingKey">
    <xsd:selector xpath="report/subject/year/subjectYear/
        ratings"/>    <xsd:field xpath="@code"/>  </xsd:key>
<xsd:key name="pYearKey">
    <xsd:selector xpath="report/subject/year"/>
    <xsd:field xpath="@year"/> </xsd:key>
<xsd:key name="pSubjectKey">
    <xsd:selector xpath="report/subject"/>
    <xsd:field xpath="@code"/> </xsd:key>
</xsd:element>
<!--======= Complex Type Definitions ==============-->
<xsd:complexType name="ReportType">
    <xsd:sequence>
        <xsd:element name="subject" type="SubjectType"
            minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence> </xsd:complexType>
<xsd:complexType name="SubjectType">
    <xsd:sequence>
        <xsd:element name="title" type="xsd:string"/>
        <xsd:element name="year" type="YearType"
            minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="code" type="SubjectCodeType"/>
</xsd:complexType>
<xsd:complexType name="SubjectYearType">
    <xsd:sequence>
        <xsd:element name="lecturer" type="xsd:string"/>
        <xsd:element name="nrEnrolled" type="xsd:integer"/>
        <xsd:element name="ratings" type="RatingType"
            minOccurs="1" maxOccurs="7"/>
    </xsd:sequence> </xsd:complexType>
<xsd:complexType name="YearType">
    <xsd:sequence> <xsd:element name="subjectYear"
            type="SubjectYearType"/> </xsd:sequence>
    <xsd:attribute name="year" type="xsd:integer"/>
</xsd:complexType>
<xsd:complexType name="SubjectYearRatingType">
    <xsd:sequence>
        <xsd:element name="nrStudents" type="xsd:integer"/>
        <xsd:element name="percent" type="xsd:double"/>
    </xsd:sequence> </xsd:complexType>
<xsd:complexType name="RatingType">
    <xsd:sequence>
        <xsd:element name="subjectYearRating"
            type="SubjectYearRatingType"/> </xsd:sequence>
    <xsd:attribute name="code" type="RatingCodeType"/>
</xsd:complexType>
<!--========== Simple Type Definitions =============-->
<xsd:simpleType name="RatingCodeType">
    <xsd:restriction base="xsd:integer">
        <xsd:minInclusive value="1"/>
        <xsd:maxInclusive value="7"/> </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="SubjectCodeType">
    <xsd:restriction base="xsd:string">
        <xsd:length value="5"/>
        <xsd:pattern value="[A-Z]{2}\d{3}"/>
</xsd:restriction></xsd:simpleType>
</xsd:schema>[i]
```

The work reported in this paper has been funded in part by the Cooperative Research Centres Program through the Department of the Prime Minister and Cabinet of the Commonwealth Government of Australia.