http://ansinet.com/itj

ITJ

ISSN 1812-5638

# INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 4 (1): 44-54, 2005
ISSN 1812-5638
© 2005 Asian Network for Scientific Information

# Extending XML Schema with Object-oriented Features

Guoren Wang, Donghong Han, Baiyou Qiao and Bin Wang
College of Information Science and Engineering, Northeastern University, Shenyang, China, 110004

**Abstract:** In this study, we extend XML Schema with nonmonotonic inheritance due to its powerful modeling ability to support multiple inheritance, overriding of elements or attributes inherited from super-elements, blocking of the inheritance of elements or attributes from super-elements and conflict handling. Another key feature of object-oriented data models is polymorphism. We introduce it into XML Schema to support polymorphic elements and polymorphic references. Moreover, we define a formalization to represent these features and present some validation rules to validate whether an XML instance document conforms to an Extended XML Schema based on the formalization. Finally, we extend XQuery to support the polymorphic feature.

**Key words:** XML Schema, nonmonotonic inheritance, object-oriented features

## INTRODUCTION

In order to constrain and define a class of XML documents, a dozen of XML schema languages have been proposed, such as DTD[1], SOX[2], XML Schema[3], Schematron[4], DSD[5], XDR[6]. However, they do not support inheritance at all except for XML Schema and SOX[7].

In XML Schema, a new type can be derived by extending or restricting the base type, which may be either complex or simple. However, it is required that a derived type has no more than one base type in an extension hierarchy, that is, multiple inheritance is not supported. Moreover, many important inheritance-related features such as overriding, blocking, polymorphism and conflict resolution cannot be supported directly by XML schema.

In XML Schema, the redefine mechanism can be used to support evolution and versioning of schemas. Unlike the include mechanism, which enables users to use external schema components without any modification, the redefine mechanism allows users to incorporate external schema components with modifications. Because attribute group definitions and model group definitions may be supersets or subsets of their original definitions, the redefine mechanism can be used to simulate overriding and blocking of element inheritance in an element hierarchy, in a two-steps way. For example, for the element hierarchy with person and student, element addr is overridden with a simple type in sub-element student. With XML Schema, this can be simulated in the following two steps. (1) A temporary type definition student is derived from the base type person with the extension mechanism and they have the same element definitions. The derived definition is stored as a temporary schema document student_tmp.xsd. (2) The external schema document student_tmp.xsd is redefined with necessary modifications for overriding element addr and then the redefined schema is stored as student.xsd.

The main shortcoming of the two-steps way to simulate overriding and blocking is that a temporary external schema document must be generated, because type definitions must use themselves as their base type definition in the redefine mechanism.

In XML Schema, there is a substitution group, which allows elements to be substituted for other elements and can be used to simulate the feature of polymorphism. Figure 1 declares two new elements chineseComment and englishComment and makes them substitutable for the comment element in the instance document. Although the substitution mechanism can be used to simulate the polymorphic feature, it has the following shortcomings:

1. For an element hierarchy, the user has to declare a substitution group for each super-element;
2. If a new sub-element is added into the element hierarchy, then the declarations of substitution groups of its super-elements have to be modified.

```
<xsd:element name="chineseComment" type="string"
substitutionGroup="comment"/>
<xsd:element name="englishComment" type="string"
substitutionGroup="comment"/>
```

Fig. 1: Complex type restriction in XML Schema

**Corresponding Author:** Dr. Guoren Wang, College of Information Science and Engineering, Northeastern University, Shenyang, China, 110004 E-mail:wanggr@mail.neu.edu.cn

Nonmonotonic multiple inheritance is a fundamental feature of object-oriented data models[8,9]. In object-oriented languages with multiple inheritance, a class may inherit attributes and methods from more than one superclass. For example, class TA might inherit attributes and methods directly from classes teacher and student. In a multiple inheritance hierarchy, users can explicitly override the inherited attributes or methods and block the inheritance of attributes or methods from superclasses[9]. For example, class person has attribute addr with a complex type, while subclass student has an overridden attribute addr with a simple type. Class person has an attribute homephone, which might be blocked in the subclass teacher.

One of the problems with multiple inheritance is that ambiguity may arise when same attribute or method is defined in more than one superclass. For example, both class teacher and class student inherit attribute addr from class person, which is overridden in class student. As a subclass of both teacher and student, TA inherits all attributes and methods from its two superclasses. In this case, an ambiguity on attribute addr arises. Therefore, conflict resolution is very important in object-oriented database systems with multiple inheritance and most systems use the superclass ordering to solve the conflicts[8,9].

In this study, we extend XML Schema with nonmonotonic inheritance due to its powerful modeling ability to support multiple inheritance, overriding of elements or attributes inherited from super-elements, blocking of the inheritance of elements or attributes from super-elements and conflict handling. Another key feature of object-oriented data models is polymorphism. We introduce it into XML Schema to support polymorphic elements and polymorphic references. Moreover, we define a formalization to represent these features and present some validation rules to validate if an XML instance document conforms to an Extended XML Schema based on the formalization. We also extend XQuery to support the polymorphic feature.

## MOTIVATION

A typical application about university teaching is taken into account as shown in Fig. 2. In the application there are seven kinds of objects: person, student, teacher, TA, course, underCourse and gradCourse. They are represented as complex elements and denoted graphically by ■. Elements person, student, teacher and TA and elements course, underCourse and gradCourse are grouped into two element hierarchies, respectively. The super-element and sub-element relationship is denoted by ➔. Element person has an ID attribute @pid denoted by ○, two simple elements name and homehpone denoted by □ and one complex element addr with three component elements day, month and year. The composite relationship is denoted by —. Element student inherits the attributes and elements from person and has its own specific elements dept and takes. Element addr from person is overridden in student, denoted by ⫲, since a simple type is used for students' addr. A student can takes some courses and this is modeled with element takes with an IDREFS attribute @courses pointing to course, denoted by ➔*. Element teacher has four specific component elements workphone, salary, dept and teaches. Element homephone from person is blocked in teacher, denoted by ⇥, since teachers usually prefer to use workphone rather than homephone as their contact phone. Similar to takes, teaches is an element with an IDREFS attribute @courses pointing to course. TA is declared as a sub-element of student and teacher, which is a nonmonotonic inheritance. Because TA may have different teaching and studying dept, element dept is declared in student and teacher, respectively, to record this information. And they are renamed as student-dept and teacher-dept in TA, respectively. Element course is declared with an ID attribute cid and four component elements name, desc, takenBy and taughtBy. takenBy is declared as an empty element with an IDREFS attribute @students to specify the students who takes the course. Similarly, taughtBy is declared as an empty element with an IDREFS attribute @teachers to specify the teachers who teaches the course. underCourse and gradCourse are declared as sub-elements of course and has no specific component elements and attributes. The root element univ is composed of the set of person and the set of course. ⩲ means an element can contain zero or more occurrences of the component element.

With nonmonotonic inheritance, there may be conflicting element or attribute declarations when elements or attributes with the same name are declared in more than one super-element. Thus, conflict handling mechanisms have to be designed to solve conflicts in element hierarchies with nonmonotonic inheritance. Notice that TA also inherits all elements and attributes from its indirect super-element person, for example, element name and attribute pid, but no conflicts arise in TA because they are inherited from the same super-element.

In an element hierarchy with nonmonotonic inheritance, a sub-element definition may override element or attribute declarations from super-elements, block the inheritance of elements or attributes from super-elements. There may be conflicts when the same elements or

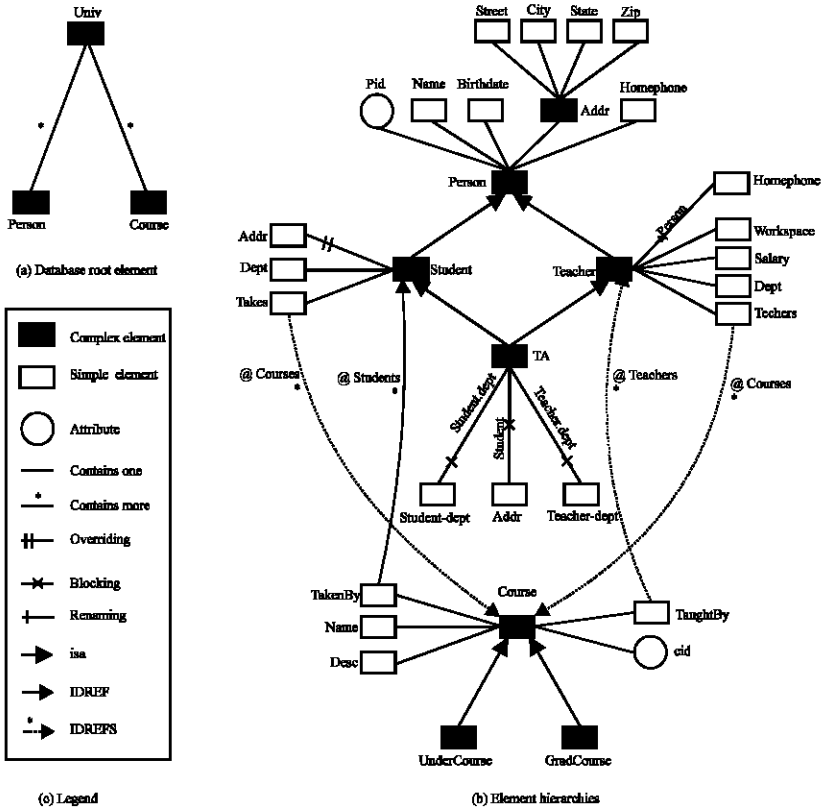


Fig. 2: Example of university application

attributes are declared in more than one super-element. The current XML Schema language only supports simple single inheritance and they do not provide mechanisms or facilities for multiple inheritance, overriding, blocking, polymorphism and conflict handling. So, in this study, we will extend XML Schema systematically with these key features so that the extended XML Schema language has "pure" object-oriented modeling ability.

## EXTENDED XML SCHEMA

We extend XML Schema, called Extended XML Schema, with multiple inheritance, blocking the inheritance of attributes and elements from super-elements, overriding the inherited attributes and elements from super-elements, conflict handling, polymorphic element and polymorphic reference.

**Extensions to XML schema:** The type definitions for elements univ, person and addr are given in Fig. 3. Although they have the same syntax as the original XML Schema, some of them (for example lines (04)-(07) of Fig. 3) have different semantics constraints on XML instance documents due to the introduction of the polymorphic feature. We will discuss it later on.

Figure 4 shows the type definition for element student that inherits personType. Because the inheritance mechanism provided by XML Schema is not flexible and powerful, we extend it in the following aspects.

1. In a type hierarchy, a subtype may have more than one supertypes to support nonmonotic multiple inheritance. Therefore, the attribute base of the extension mechanism is modified as bases, e.g. in line (03) of Fig. 4.
2. In the original XML Schema, a subtype inherits all elements but not attributes from its supertype. Although attributes are different from elements, they are a special kind of information from users' point of view. Therefore, in the Extended XML Schema, a subtype inherits not only elements but also attributes from its supertypes. For example, subtype studentType inherits an attribute pid and four elements name, birthdate, addr and homephone. Note that no specific ID attributes are allowed to be declared in the subtype since pid is an ID attribute inherited by the subtype.

```
(01) <xsd:element name="univ" type="univType"/>
(02) <xsd:complexType name="univType">
(03)   <xsd:sequence>
(04)    <xsd:element name="person" type="personType"
(05)                  minOccurs="0" maxOccurs="unbounded"= >
(06)    <xsd:element name="course" type="courseType"
(07)                  minOccurs="0" maxOccurs="unbounded"= >
(08)   </xsd:sequence>
(09)  </xsd:complexType>
(10)  <xsd:complexType name="personType">
(11)    <xsd:sequence>
(12)       <xsd:element name="name" type="xsd:string"/>
(13)       <xsd:element name="birthdate" type="xsd:date"/>
(14)       <xsd:element name="addr" type="addrType"/>
(15)       <xsd:element name="homephone" type="xsd:string"/>
(16)     </xsd:sequence>
(17)    <xsd:attribute name="pid" type="xsd:ID" use="required"/>
(18) </xsd:complexType>
(19) <xsd:complexType name="addrType">
(20)       <xsd:sequence>
(21)       <xsd:element name="street" type="xsd:string"/>
(22)       <xsd:element name="city" type="xsd:string"/>
(23)       <xsd:element name="state" type="xsd:string"/>
(24)       <xsd:element name="zip" type="xsd:string"/>
(25)    </xsd:sequence>
(26) </xsd:complexType>
```

Fig. 3: Type definitions for elements univ., person and addr.

```
(01) <xsd:complexType name="studentType">
(02)    <xsd:complexContent>
(03)     <xsd:extension bases="personType">
(04)       <xsd:sequence>
(05)       <xsd:element name="addr" type="xsd:string"/>
(06)          <xsd:element name="dept" type="xsd:string"/>
(07)          <xsd:element name="takes">
(08)          <xsd:attribute name="courses" type="IDREFS"
(09)                        target="courseType" use="implied"/>
(10)          </xsd:element>
(11)         </xsd:sequence>
(12)       <xsd:attribute name="sno" type="xsd:string" use="required"/>
(13)    </xsd:extension>
(14)   </xsd:complexContent>
(15) </xsd:complexType>
```

Fig. 4: Type definition for element Student

3. In the Extended XML Schema, a specific component element or attribute in the subtype may override the element or attribute defined in the supertype. For example, in the subtype student component element addr inherited from personType is overridden with a new simple type, as shows in line (05) of Fig. 4. Note that there is no special syntax extension for overriding of element and attribute.

Sometimes, it is necessary to allow a subtype blocks the inheritance of attributes and elements from its supertypes. For example, teachers usually prefer to use workphone rather than homephone as their contact phone. It is reasonable that in the definition of the subtype teacherType the inheritance of homepone is blocked from its supertype personType. Therefore, the blocking mechanism is introduced as shown in lines 13-15 of Fig. 5. The blocking mechanism has an attribute from specifying from which type the inheritance is blocked and some components specifying attributes and elements to be blocked. Note that in the original XML Schema a blocking mechanism is provided to control type derivations. For example, if we want to block any derivation-by-extension from being used in it place of teacherType, we can append an attribute block with value "extension" to the element complexType for teacherType, as shown in the following:

```
<xsd:complexType name="teacherType" block="extension">
```

This blocking mechanism can be used to control type derivations but not flexible enough. Therefore, we provide

```
(01) <xsd:complexType name="teacherType">
(02)   <xsd:complexContent>
(03)     <xsd:extension bases="personType">
(04)       <xsd:sequence>
(05)          <xsd:element name="workphone" type="xsd:integer"/>
(06)          <xsd:element name="salary" type="xsd:float"/>
(07)          <xsd:element name="dept" type="xsd:string"/>
(08)          <xsd:element name="teaches">
(09)          <xsd:attribute name="courses" type="IDREFS"
(10)                          target="courseType" use="implied"/>
(11)        </xsd:element>
(12)           </xsd:sequence>
(13)           <xsd:block from="personType">
(14)             <xsd:element name="homephone"/>
(15)           </xsd:block>
(16)          <xsd:attribute name="tno" type="xsd:string" use="required"/>
(17)       </xsd:extension>
(18)     </xsd:complexContent>
(19) </xsd:complexType>
```

Fig. 5: Type definition for element Teacher

```
(01) <xsd:complexType name="TAType">
(02)   <xsd:complexContent>
(03)    <xsd:extension bases="studentType teacherType">
(04)     <xsd:rename>
(05)      <xsd:element name="dept" from="studentType" as="student-dept"/>
(06)      <xsd:element name="dept" from="teacherType" as="teacher-dept"/>
(07)      </xsd:rename>
(08)       <xsd:block from="studentType">
(09)         <xsd:element name="addr"/>
(10)       <xsd:block>
(11)      </xsd:extension>
(12)    </xsd:complexContent>
(13) </xsd:complexType>
```

Fig. 6: Type definition for element TA

a new blocking mechanism as component elements of the extension element to control the inheritance of attributes and elements from supertypes.

The return type none is used to specify blocking, but the superclass from which the inheritance is blocked is not specified[11]. This way can work well in the case of single inheritance, but not in the case of multiple inheritance with selectable blocking; that is, superclass attributes can be blocked from some subclasses. Another advantage of the selectable blocking mechanism is that it can be used to resolve conflicts, described later on.

Another major extension to XML Schema is typing of IDREF and IDREFS. Just as pointed out by Lewis *et al.*[1], both XML Schema and DTD do not support typing of IDREF and IDREFS. In this case, a reference may point to any kind of element instance. One cannot require a reference points to only an expected kind of element instances. For example, it is possible that attribute @courses of the element takes in student references a person rather than a course. No XML 1.0 compliant processor can detect this problem[10] since we cannot type IDREF and IDREFS at the schema level. Therefore, we extend attribute declaration with attribute target specifying the type of IDREF or IDREFS, for example, in lines (08)-(09) of Fig. 4 and in lines (09)-(10) of Fig. 5.

With multiple inheritance, conflicts can easily occur. In Fig. 6, type TAType inherits elements and attributes from both supertypes studentType and teacherType. There are two conflicts to be resolved, since elements addr and dept are declared on both supertypes studentType and teacherType. In our Extended XML Schema, three ways can be used to handle conflicts. In the first way, a conflict resolution declaration is specified explicitly to indicate from which supertype an element or attribute is inherited, for example, the block construct in lines (08)-(10) of Fig. 6 indicates that the declaration of addr is inherited from the supertype teacherType rather than from studentType. In the second way, the names of elements or attributes causing conflicts are explicitly renamed in the inheriting element declaration, for example, in the subtype TAType declaration, the rename construct in line (05) of Fig. 6 renames element dept inherited from supertype student-Type to student-dept while the rename construct in line (06) of Fig. 6 from teacherType to

```
(01) <xsd:complexType name="courseType">
(02)   <xsd:sequence>
(03)     <xsd:element name="name" type="xsd:string"/>
(04)    <xsd:element name="desc" type="xsd:string"/>
(05)    <xsd:element name="takenBy">
(06)     <xsd:attribute name="students" type="xsd:IDREFS"
(07)                          target="studentType" use="implied"/>
(08)     <xsd:attribute name="teachers" type="xsd:IDREFS"
(09)                          target="teacherType" use="implied"/>
(10)    </xsd:element>
(11)   </xsd:sequence>
(12)  <xsd:attribute name="cid" type="xsd:ID" use="required"/>
(13) </xsd:complexType>
(14) <xsd:complexType name="underCourseType">
(15)     <xsd:complexContent>
(16)       <xsd:extension bases="courseType"/>
(17)     </xsd:complexContent>
(18) </xsd:complexType>
(19) <xsd:complexType name="gradCourseType">
(20)    <xsd:complexContent>
(21)      <xsd:extension bases="courseType"/>
(22)    </xsd:complexContent>
(23) </xsd:complexType>
(24) <xsd:element name="student" type="studentType">
(25) <xsd:element name="teacher" type="teacherType">
(26) <xsd:element name="TA" type="TAType">
(27) <xsd:element name="underCourse" type="underCourseType">
(28) <xsd:element name="gradCourse" type="gradCourseType">
```

Fig. 7: Type definitions for elements course, underCourse and gradCourse

teacher-dept. Finally, if there is a conflict and there is no conflict resolution declaration, then the element or attribute is inherited from the supertype in the order the supertypes are listed in the extension construct of the type definition. For example, if there is a conflict for element addr and there is no explicit conflict resolution declared for it in the definition of type TAType, then element addr in supertype teacherType is inherited.

Figure 7 shows the type definitions for elements course (lines (01-13)), underCourse (lines (14-18)) and gradCourse (lines (19-23)) and other element declarations (lines (24-28)). The complete Extended XML Schema definition for the university application in Fig. 2 consists of the type definitions and the element declarations in Fig. 3-7.

**Extensions to XML instance document:** We extend XML with polymorphic element and polymorphic reference. Figure 8 shows a valid XML instance document supporting polymorphic element and polymorphic reference with respect to the extended XML Schema shown earlier.

In object-oriented paradigm, polymorphism is a very useful and important feature, which provides the possibility of manipulating polymorphic collections. Consider three classes person, teacher and student. Class person is the common superclass of teacher and student and the extents of these three classes are persons, teachers and students, respectively. Therefore, the set persons contains objects of classes person, teacher and student due to polymorphism. Thus, the extent persons contains three possible collections of the elements. It is important to extend XML with the polymorphic feature.

Consider the examples described before, type personType has three direct or indirect subtypes, studentType, teacherType and TAType and type courseType two direct subtypes, underCourseType and gradCourseType. When polymorphism is introduced into XML, an element instance of personType in a valid instance document can be substituted with an instance of elements of its subtypes and the instance document should still be valid. If the type of an element has at least one subtype, then the element is polymorphic. For example, element person is polymorphic since type personType has three direct or indirect subtypes. the person element instance can be substituted by instances of student, teacher, or TA since their types all are subtypes of personType. Similarly, the instance of course can be substituted by instance of underCourse and gradCourse. The substituting element instances is referred to as polymorphic instance. From lines (04)-(07) of Fig. 3, we can see that element univ can contain more person element instances and more course element instances; that is univ→person*;course*. Therefore, element univ can contain seven component element instances due to polymorphism: (1) person instances; (2)

```
<univ>
<-person instance ->
  <person pid="100">
  <name> Jaoune Barbosa </name>
  <birthdate> 1965-04-07 </birthdate>
  <addr>
   <street> 310 University </street>
   <city> Ottawa </city>
   <state> Ontario </state>
   <zip> K1S 5B6 </zip>
</addr>
   <homephone> 5073322 </homephone>
</person>
<-student instance ->
   <student pid="200">
   <name> Jones Gillmann
   <birthdate> 1976-02-25 </birthdate>
   <addr> 708D Somerset St </addr>
   <homephone> 6185708 </homephone>
   <dept> Computer Science </dept>
   <takes courses="CS200 CS300" />
   </student>
<-teacher instance ->
   <teacher pid="300">
   <name> Alley Srivastava </name>
   <birthdate> 1957-06-26 </birthdate>
<addr>
   <street> 56 Broson </street>
   <city> Ottawa </city>
   <state> Ontario </state>
   <zip> K2B 6M8 </zip>
   </addr>
   <workphone> 2314343 </workphone>
   <salary> 1200.00 <salary>
   <dept> Computer Science </dept>
    <teaches courses="CS200 CS400"/>
    </teacher>
    <-TA instance ->
     <TA pid="400">
      <name> Alice Bumbulis </name>
<birthdate> 1976-08-29 </birthdate>
<addr>
   <street> 440 Albert </street>
  <city> Ottawa </city>
  <state> Ontario </state>
   <zip> K1R 6P6 </zip>
 </addr>
   <homephone> 2915318 </homephone>
   <workphone> 2502600 </workphone>
   <student-dept> CS </student-dept>
<teacher-dept> SE </teacher-dept>
   <takes courses="CS400" />
    <teaches teaches="CS300" />
</TA>
<-course instance ->
  </name> <course cid="CS100" >
  <name> Introduction to CS </name>
  <desc> Continuing Education </desc>
</course>
   <-underCourse instances ->
   <underCourse cid="CS200" >
   <name> Introduction to DBS </name>
   <desc> Basic concepts </desc>
   <takenBy students="200" />
   <taughtBy teachers="300" />
</underCourse>
   <underCourse cid="CS300" >
   <name> Introduction to SE </name>
   <desc> Basic concepts </desc>
   <takenBy students="200" />
   <taughtBy teachers="400" />
</underCourse>
   <-gradCourse instance ->
   <gradCourse cid="CS400" >
   <name> DBMS </name>
   <desc> Impl. Techniques </desc>
   <takenBy students="400" />
   <taughtBy teachers="300" />
 </gradCourse>
</univ>
```

Fig. 8: XML instance document of Fig. 2

student instances; (3) teacher instances; (4) TA instances; (5) course instances; (6) underCourse instances and (7) gradCourse instances.

Now we extend XML Schema with polymorphic reference, which is similar to polymorphic element. A little bit complicated example for polymorphic reference is that a teacher may teach several courses including underCourses and gradCourses as well, see the definition of element teacher in Fig. 5 and its instance in Fig. 8. In the definition, teaches is an IDREFS to course. If polymorphic references are supported by the system(that is, teaches can also be used to reference to either underCourse or gradCourse elements as their types all are subtypes of the type of element course), the following six combinations are valid in the instance document: (1) a teacher teaches courses; (2) a teacher teaches underCourses; (3) a teacher teaches gradCourses; (4) a TA teaches courses; (5) a TA teaches underCourses and (6) a TA teaches gradCourses.

Polymorphic reference is introduced to meet the above requirements. An IDREF or IDREFS attribute of a given element can point to instance(s) of the substituting elements of the element. It is referred to as polymorphic references.

For the above example, takes can point to instances of element course as well as its substituting elements underCourse and gradCourse.

## FORMALIZATION

Before discussing the formalization for element hierarchy, we give the following notations:

1. Notations for attributes: $attr_i$ is used to represent an attribute; $ai_i$ for an attribute instance; AN for the set of all attribute names; AT for the set of built-in attribute types.
2. Notations for elements: $se_i$ is used for simple element; $ce_i$ for complex element; $sei_i$ for simple element instance; $cei_i$ for complex element instance; EN for the set of element names; ET is used to represent the set of element types.
3. Notations for document schemas and instances: dsch is used for a document schema; dins for document instance; E for the set of elements; EI for the set of element instances; A for the set of attributes; AI for the set of attribute instances; V(ae) for the set of values of attribute or simple element ae;
4. Notations for functions: $f_t(val)$ is used for the function that returns the type of the value val ; $f_{targetElem}(ai)$ for the function that returns the target element name pointed by ai; $f_{se}(ce)$ for the function which returns the set of sub-elements of the element ce; $f_{len}(s)$ for the function which returns the length of the set or ordered set s;

Next, we define a formalization for element hierarchy, including schema, complex element, simple element, attribute, document instance, complex element instance, simple element instance and attribute instance.

**Definition 1:** An attribute is defined as a 4-tuple attr=<attr_name, attr_type, elem, target_elem>, where attr_name∈AN is an attribute name, attr_type∈AT is an attribute type, elem is used to specify the element attribute attr belongs to and target_elem is used to specify the target element attribute attr points to in the case of IDREF or IDREFS.

**Definition 2:** A simple element se is defined as a 3-tuple se=<elem_name, elem_type, attrs>, where elem_name∈ EN is an element_name; elem_type∈ET is the type of the simple element se; attrs is a set of attributes of the simple element se;

**Definition 3:** A complex element ce is defined as a 6-tuple ce=<elem_name, super_elems, component_elems, attrs, $f_e$, $f_a$>, where elem_name∈εN is the name of the complex element; super_elems⊆E is an ordered set consisting of the direct super-elements of the element ce; component_elems⊆E is an ordered set consisting of the component elements of the element ce and attrs is a set of attributes of the element ce; $f_e$ is the set of rules to represent renaming, blocking and overriding for elements, similarly fa is the set of rules for attributes.

$f_e$ consists of three kinds of rules: $f^r_e$ , $f^b_e$ and $f^o_e$ and they have the following forms.

1. $f^r_e$ (superElem):oldElem→newElem. It is an element renaming rule, means that element oldElem inherited from super-element superElem is renamed as newElem.
2. $f^b_e$: elem→{superElems}. It is an element inheritance blocking rule, means that the inheritance of element elem is blocked from super-elements superElems.
3. $f^o_e$ :elem→elemType. It is an element overriding rule, means that element elem is overridden with the new element type elemType.

We can define $f_a$ as well as $f^r_a$ , $f^b_a$ and $f^o_a$ in a similar way.

**Definition 4:** A schema dsch is defined as a 4-tuple dsch=<root, ces, ses, attrs>, where root∈ε is the root element, ces⊆E is the set of all complex elements, ses⊆E is the set of all simple elements and attrs⊆A is the set of all attributes.

**Definition 5:** An attribute instance ai is defined as a 3-tuple ai=<attr_name, attr value, elem_instance>, where attr_name∈AN is the name of the attribute, attr_value∈V (attr_name) is the value of the attribute, elem_instance is the element instance ai belongs to.

**Definition 6:** A simple element instance sei is defined as a 3-tuple<elem_name, elem value, attr_instances>, where elem_name∈EN is the name of the simple element, elem value ∈ V(elem_name) is the value of the simple element sei and attr_instances∈AI is the set of attribute instances of the simple element sei.

**Definition 7:** A complex element instance cei is defined as a cei=<elem_name, elem_instances, attr_instances>, where elem_name∈EN is the name of the complex element defined as; elem_instances⊆EI is the ordered set of the sub-elements of the complex element cei and the sub-elements may be either complex or simple; attr_instances⊆AI is the set of attribute instances of the complex element cei.

**Definition 8:** A document instance dins is defined as a 4-tuple dins=<root, ceis, seis, ais>, where root∈EI is the instance of a root element, ceis⊆EI is the set of all complex element instances, seis⊆EI is the set of all simple element instances and ais⊆AI is the set of all attribute instances.

## INSTANCE VALIDATION

We discuss how to check whether or not a document instance conforms to a schema; that is, validation of semantics. Assume that dsch=<root, ces, ses, attrs> is an Extended XML Schema schema and dins=<root, ceis, seis, ais> is a document instance. The validation rules of semantics to support element hierarchy are defined as follows:

**Rule 1:** Let attr∈dsch.attrs be an attribute of an Extended XML Schema schema and ai∈dins.ais be an instance of an attribute. The instance ai is valid with respect to attribute attr, denoted by attr|=ai, if and only if

1. The attribute name of the instance ai conforms to the name of the attribute attr ; that is, ai.attr_name=attr.attr_name;
2. The type of value of the instance ai conforms to the type of the attribute attr; that is, ft(ai.attr value)=attr.attr_type;
3. The name of the instance ai conforms to the name of the attribute attr ; that is, ai.elem_instance.elem name=attr.elem.elem_name or ai.elem_instance.elem name is the sub-element of attr.elem.elem_name and
4. If the type of value of the attribute attr is IDREF or IDREFS, then the type of the element instance pointed by the attribute instance ai conforms to the target element of attr ; that is, ftargetElem(ai)=attr.target_elem or ftargetElem(ai) is the sub-element of attr.target_elem.

**Rule 2:** Let se∈dsch.ses be an simple element of an Extended XML Schema and sei∈dins.seis an simple element instance. The instance sei is valid with respect to the simple element se, denoted by se|=sei, if and only if

1. The element name of the instance sei conforms to the name of the simple element se; that is, sei.elem_name=se.attr_name;
2. The type of value of the instance sei conforms to the type of the simple element se; that is, ft(sei:elem_value)=se.attr_type and
3. Sei.attr_instances is valid with respect with se, denoted by se |= sei.attr_instances; that is, (∀x)(x∈sei.attr_instances)((∃y)(y∈se.attrs_)∧(y |=x))

**Rule 3:** Let ce∈dsch.ces be an complex element of an Extended XML Schema and cei∈dins.ceis an complex element instance. The instance cei is valid with respect to the complex element ce, denoted by ce|=cei, if and only if

1. Because of the polymorphism of element, the element name of the instance cei is either the name of the complex element ce or the name of any sub-element of the complex element ce; that is,
   a. cei.elem_name=ce.elem_name or
   b. $(\exists x \in f_{se}(ce))(x.elem\_name=ce.elem\_name)$

2. The elem_instances of the complex element instance cei must conform to the closure component elements of the complex element ce one by one; that is,
   a. $f_{len}(cei.elem\_instances)=f_{len}(ce.component\_elems^*)$ and
   b. For $(i=1; i \le f_{len}(cei.elem\_instances))$ ce.component_elems*[i]|=cei.elem_instances[i];

ce:component_elems* can be computed as follows.

a. For $(i=1; i \le f_{len}(ce.super\_elems))$ tmp1=tmp1 ∪super_elems[i].component_elems*;
b. tmp1 = tmp1 ∪ ce.component_elems;
c. Applying renaming operations $ce.f^r_e$ on tmp1;
d. Applying blocking operations $ce.f^b_e$ on tmp1;
e. Applying overriding operations $ce.f^o_e$ on tmp1;
f. ce.component_elems* = tmp1.

3. cei.attr_instances is valid with respect to ce, denoted by ce|=cei.attr_instances; that is, (∀x)(x∈cei.attr_instances)((∃y)(y∈ce.attrs*∧y|=x)).

The computation of ce.attrs* is similar to ce.component_elems*.

**Rule 4:** Let dsch=<root, ces, ses, attrs> be an Extended XML Schema and dins=<root, ceis, seis, ais> be a document instance. The document instance dins is valid with respect to the schema dsch if and only if dsch.root|=dins.root.

## QUERYING

We extend XQuery[11,12], called Extended XQuery, to support queries on XML instance documents conforming to Extended XML Schema. In an XML instance document conforming to an Extended XML schema, an instance of a sub-element may occur anywhere an instance of its super-element is expected. Similarly, in a statement written

with the Extended XQuery, a sub-element may occur anywhere its super-element is expected, due to polymorphism.

For example, the following query is to get all students' names:

```
FOR $a in document("univ.xml")/univ/student
RETURN <student pid=$a/pid/text()>
        <name> $a/name/text() </name>
    </student>
```

The above extended XQuery statement does not have different syntax but different semantics from the original XQuery. In the original XQuery, it has incorrect semantics since element student is not a component element of univ.

Sometimes, users want to get information of an element as well as all its sub-elements. Finding information of all people including person, student, teacher and TA is an example of such a query. The query can be represented as union of four queries similar to the above query, but it is not concise and has no good performance. To address this problem, we introduce the concept of inclusive element, denoted by E↓. For example, the following extended XQuery is to get all people's information:

```
FOR $a in document("univ.xml")/univ/person↓
RETURN $a
```

In the above query statement, the path query "/univ/person↓" is to get the set of instances of person and its sub-elements student, teacher and TA under element univ.

We extended XML Schema with polymorphic reference and typing of reference. Now we extend XQuery to support these concepts. If an IDREF(S) attribute attr is declared as a reference to target element elem, then in the extended XQuery it can be represented as "@(elem)attr", which tells the query processor the reference attr points to element elem. Based on the polymorphic reference, an IDREF(S) pointing to a target element elem can point to any sub-elements of elem. Thus, "@(subElem)attr" is also valid if subElem is a sub-element of elem and its meaning is to get reference(s) pointing to element subElem.

For example, the following query is to get the names of all underCourses taught by TA "Alice Bunibulis":

```
FOR $a in document("univ.xml")/univ/underCourse
FOR $b in document("univ.xml")/univ/TA
FOR $c in $b/teachs/@(underCourse)courses
WHERE $b/name/text() = "Alice Bunibulis" and $c =
$a/@id
RETURN <underCourse>
    <name>
            $a/name/text()
        </name>
    </underCourse>
```

In the above query, the path query "$b/teaches/@(underCourse)courses" means to get the set of ID of underCourses taught by TAs.

Similar to inclusive element, we can also extend XQuery with inclusive reference. In a query statement, an inclusive reference, denoted by @(targetElem#)attr, means to get all IDs of instances of element targetElem as well as its sub-elements pointed by attr.

For example, we can use the following query to get all courses including underCourse and gradCourse are taught by teacher "Alley Srivastava":

```
FOR $a in document("univ.xml")/univ/teacher
FOR $b in document("univ.xml")/univ/course↓ WHERE
$a/name/text() = "Alley Srivastava" and
    $a/teaches/@(course↓ )courses = $b/@cid
    RETURN $b
```

In the above query statement, the path query "$a/teaches/@(course↓)courses" is to get all IDs of courses including elements course, underCourse and gradCourse.

## CONCLUSIONS

In this study, we extend XML Schema to support the key object-oriented features such as nonmonotonic inheritance, overriding, blocking, conflict handling, polymorphism and typing references. Also, we extend Xquery to support the features of Extended XML Schema. Moreover, we define a formalization to represent these features and some validation rules to validate if an XML instance document conforms to an Extended XML Schema based on the formalization. We are developing an XML parser and validation system to support the extended XML described in this study.

## ACKNOWLEDGMENTS

This research is partially supported by the Teaching and Research Award Programme for Outstanding Young Teachers in PostSecondary Institutions by the Ministry of Education, China (TRAPOYT) and National Natural Science Foundation of China under grant No. 60273079 and 60473074.

## REFERENCES


1. Bray, T., J. Paoli, C.M. Sperberg-McQueen and E. Maler, 2000. Extensible Markup Language (XML) 1.0. 2nd Edn., http://www.w3.org/ TR/REC-xml.
2. Davidson, A., M. Fuchs and M. Hedin, 1999. Schema for Object-Oriented XML 2.0. W3C Note. http://www.w3.org/TR/NOTE-SOX.
3. Fallside, D.C., 2001. XML Schema Part 0: Primer. W3C Recommendation. http://www.w3. org/TR/xmlschema-0/.
4. Jelliffe, R., 2000. Schematron. http:// www.ascc.net/xml/resource/schematron/.
5. Klarlund, N., A. Moller and M.I. Schwartzbach, 2002. The DSD schema language. Automated Software Engineering, 9: 285-319.
6. XML schema developer's guide, 2000. http://msdn.microsoft. com/xml/XMLGuide/schema-overview.asp.
7. Lee, D. and W.W. Chu, 2002. Comparative analysis of six XML schema languages. ACM SIGMOD Record, 29: 117-151.
8. Dobbie, G. and R.W. Topor, 1995. Resolving ambiguities caused by multiple inheritance. In: Proceedings of the 4th International Conference on Deductive and Object-Oriented Databases. Lecture Notes in Computer Science, Vol. 1013. Springer-Verlag, Berlin Heidelberg Singapore, pp: 265-280
9. Liu, M., G. Dobbie and T.W. Ling, 2002. A logical foundation for deductive object-oriented databases. ACM Transaction on Database Systems, 2: 117-151.
10. Lewis, P.M., A. Bernstein and M. Kifer, 2002. Databases and Transaction Processing: An Application-oriented Approach. Addison Wesley.
11. Chamberlin, D., D. Florescu, J. Robie and J. SimEeon, 2001. QXuery: A query language for XML. W3C Note. http://www.w3.org/TR/xquery.
12. Fankhauser, P., 2001. XQuery formal semantics: State and challenges. SIGMOD Record. 30: 14-19.