Synopsis of PhD Dissertation
Modeling of Constraints in Distributed Object Oriented Environment

Chhanda Ray
R.C.C Institute of Information Technology
Canal South Road, Beliaghata, Calcutta-15, INDIA.
e-mail: chhandaray@hotmail.com

Abstract

Integrating data from many heterogeneous sources require interoperability between the data repositories. Object Oriented Data Modeling is gaining popularity because of its elegance in representing real world situations. On the other hand, Integrity constraints traditionally form an essential part of schema definition. In this context, Modeling of constraints in Object Oriented Model has been the focus of attention of many researchers in recent times. We have introduced a constraint model for object-oriented data. We have formalized the constraint model by expressing it using Unified Modeling Language (UML) and ODMG’s ODL. To increase interoperability, the constraint model is also represented by using a standard web based interoperable medium like extensible Markup Language (XML). For efficient manipulation of data into a more manageable model that is native to the programming language, an efficient way of storing XML data with constraint information into Java Object Oriented Programming Language is also introduced. Finally, a Wrapper based translator has been developed to accomplish two-way interoperability between XML and Java Source Code towards this end.

Keywords: Interoperability, Object Oriented Databases, ODL, UML, XML.

1. Introduction

Object Oriented data Modeling is gaining popularity because of its elegance in representing real world situations [2], [4]. Methods and tools, such as UML [10], for Object Oriented Modeling are being developed. Object Oriented Databases are also gaining focus over relational ones.

Integrity constraints traditionally form an essential part of schema definition. In addition, they are useful for query optimization, update anomaly prevention and for information preservation in data integration. Integrity constraints are also used to model references in relational databases, through keys and foreign keys. In this context, Modeling of constraints in Object Oriented Data is an active research topic [1], [5], [6], [7], [9], [10], [17]. In [6], an extension of XML DTD has been proposed that specifies both semantic structure as well as integrity constraints for XML data. A Unified Constraint Model (UCM) is proposed in [7] using XML DTD which is both simple and expressive. However, XML Schema Definition language is more powerful than XML DTD for semantic specifications.

Integrating data from many heterogeneous resources require interoperability between the data repositories. Due to increasing popularity of XML as a standard interoperable medium for exchanging data between web applications, there has been a lot of research for transformation of data from various sources including RDBMS repositories to XML and vice versa [3], [15], [16]. An Object Oriented Model for interoperability is introduced in [18] to solve the data and operation inconsistency problem in legacy systems.

In this work, our contribution is as follows. First, we have proposed a constraint model for Object Oriented Data in [11]. The syntax and semantics of such modeling is extended to deal with constraints in single inheritance. The first step of semantic modeling is to identify useful semantic concepts. The useful semantic concepts, then, are captured in formal modeling languages such as UML or ODMG’s ODL. Therefore, we have formalized the constraint model by expressing it using UML and ODL in [12]. To increase interoperability, in [13], the constraint model is also represented by using a standard web based medium like extensible Markup Language (XML). For efficient manipulation of data into a more manageable model that is native to the programming language, an efficient way of storing XML data with constraint information into Java Object Oriented Programming Language is introduced in [14]. Finally, a wrapper-based translator has been developed to accomplish two-way interoperability between XML and Java Source Code.
The layout of this paper is as follows. The next section describes the constraint model and Section 3 introduces the formalization of the model using UML and ODL. The derivation of ODMG database design from UML class diagram has been represented in Section 3.1. In Section 4, Representation of XML data with constraints is introduced. An example is also illustrated in Section 4.1. Section 5 describes the interoperability between XML and Java with constraint information and we conclude in Section 6.

2. Modeling Constraints

An integrity constraint is semantic information in an object or a relationship among objects. A constraint specifies a condition and a proposition that must be maintained as true. We attempted to model each constraint as a Boolean method, which returns either true value or false value. If the predicate within a method is satisfied by a model element then the method will return true, otherwise the method returns false. These methods are clearly distinguished from the usual methods of a class by their usage and therefore, the constraint methods are accommodated differently in object oriented data model.

We have considered three different types of constraints present in an object oriented data model - Single attribute constraints, Multiple attributes constraints and Class constraints. Single attribute constraints or constraint attributes are applicable to individual attributes of a class. For example, suppose that there is a class “Employee” with usual attributes and methods. A constraint on the attribute “age” of any “Employee” object may be described as “age of an employee must between 20 & 60”. Multiple attributes constraints or constraint methods involve more than one attribute of a class. For example, there could be a constraint described as “if experience of an employee is less than 5 years then the salary can not be more than $4500”. Class constraints are applicable to all objects of individual classes of an object oriented database system. A typical example of a class constraint in the “Employee” class may be described as “id of an employee must be unique”. To handle single and multi attribute constraints, the element “class” is defined as a 4-tuple <A, CA, M, CM> where A & M represent attributes and methods of a class and CA, CM represent constraint attributes and constraint methods.

To represent class constraints, we introduced a singleton collection class associated with each general user-defined class, where the singleton collection class would always contain a collection of all objects of the user-defined class. All constraints that need to check all objects of the user-defined class for validation become Boolean methods of the singleton class. For example, for the class “Employee”, a collection class “Employee_Collection” may be defined. The class “Employee_Collection” has two attributes, a pointer to the list of all existing “Employee” objects and the total number of “Employee” objects. Whenever a new object of the class “Employee” is created, a reference to that object is added to the list “allElements” and the integer “noOfElements” is incremented. Similarly, whenever, an “Employee” object is deleted then the reference of that object is deleted from the list “allElements” and the integer “noOfElements” is decremented. The collection class also contains all the boolean methods to check the class constraint mentioned for the class “Employee”.

3. Expressing Constraint Model in UML and ODL

The Unified Modeling Language (UML) [10] is the result of an effort in developing a single standardized language for object-oriented modeling. The UML meta model is represented by a Foundation Package that contains all of the constructs provided by UML for modeling software systems. The UML Foundation Package is made up of three sub packages among which the Core Package defines the basic abstract and concrete constructs needed for the development of the object models. As the constraint model [11] enhances the definition of a Class, the UML meta model needs to be extended to accommodate constraints. We introduced four new model elements to the core package of UML meta model as follows.

a) **ConstrainedAttribute** – **ConstrainedAttribute** class is introduced as a subclass of **Attribute** class, which is also a subclass of **StructuralFeature** class. **ConstrainedAttribute** class can contains a number of methods. So, this new model element **ConstrainedAttribute** class is connected with the UML meta model class **Method**, which is a subclass of **BehaviouralFeature** class. In UML meta model, **Class** class is a subclass of **Classifier** class which describes behavioral and structural features and thus, each **ConstrainedAttribute** class is associated with **Class** class. The **ConstrainedAttribute** class extends the UML meta model **Attribute** class. By the virtue of inheritance all the attributes of the **Attribute** class, such as **Changeable, frozen, addOnly, initialValue, multiplicity** etc. will
be inherited into this class. In addition, it has three new attributes \textit{name, attributeName, methodCount}, which specifies the name of the constraint, the names of the attributes associated with the constraint and the total number of constraints.

b) \textit{ConstrainedMethod} - The \textit{ConstrainedMethod} class extends the \textit{Method} class of the UML meta model. In addition to all attributes of the \textit{Method} class of UML meta model, it has four extra attributes \textit{name, attributeName, methodCount and attributeCount}. \textit{Name attribute} indicates the name of the constraint, \textit{attributeName} is an array of name of the attributes associated with a constraint, \textit{methodCount} represent the total number of constrained methods and \textit{attributeCount} represent the number of attributes associated with a constraint. The \textit{attributeCount} attribute represents the number of elements within the \textit{attributeName} array.

c) \textit{CollectionClass} – The \textit{CollectionClass} class is introduced as a subclass of \textit{Classifier} class of the UML meta model. In addition to all attributes of \textit{Classifier} class, it has three new attributes \textit{className, objectTotal and constraintCount} which represent the name of the user-defined class associated with the \textit{CollectionClass} class, the total number of objects within the user-defined class and the total number of constraints of the user-defined class respectively.

d) \textit{MyClass} – \textit{MyClass} is introduced as a subclass of \textit{Class} class in UML meta model. By the virtue of object orientation, all the attributes of \textit{Class} class will be inherited into \textit{MyClass} class. In addition, it has one new attribute \textit{eclassName}, which represents the name of the \textit{CollectionClass} class associated with the user-defined (\textit{MyClass}) class.

The motivation behind expressing constraint model in UML is formalization. It is expected that such formalization would lead to the development of sound tools for analysis and synthesis.

After conceptual modeling, the model is transformed to a database design (represented by a database schema definition), which can be implemented in an object oriented database (OODB) system. The language used to define the specification of object types for the ODMG/OM is called Object Definition Language (ODL). Extensions to the ODMG/OM to accommodate constraints are discussed in the following.

(i) \textit{Single attribute constraints & multiple attribute constraints} – We regard these constraints definition as a part of class definition and extend the class declaration accordingly.

\begin{verbatim}
<class>:: = <class header>{<interface_body>[<cons_dcl>]}<cons_dcl>::= <cons_spec><attributes_lists>:<action>
<cons_spec>::= constrainedattribute|constrainedmethod <attribute_lists>::={attr1, attr2,.....,attrn} <action>::= <user_defined_procedure>
\end{verbatim}

(ii) \textit{Class constraints} - To show the connection between a general purpose class and the corresponding collection class we use a \textit{relationship} with in the class definition of the general purpose class. To include the integrity definitions in the ODMG ODL, the BNF for relationship declaration is extended accordingly.

\begin{verbatim}
<rel_dcl>:= relationship<collection_type><class_name><relationship_name> inverse <inverse_class_name> <collection_type>:= set|bag|list|array|dictionary
\end{verbatim}

To declare the constrained methods in collection class the corresponding BNF is as follows.

\begin{verbatim}
<class>:: = <class header>{<interface_body>[<cons_dcl>]}<cons_dcl>::= <cons_spec><subclass attributes_lists>:<action>
<cons_spec>::= constrainedmethod <subclassattribute_lists>::={subclassname.attr1,subclassname.attr2,....,subclassname..attrn } <action>::= <user_defined_procedure>
\end{verbatim}
3.1 ODMG Database Design from UML Class Diagram

The derivation of an ODMG schema begins from a package. Within a package, the following steps are needed to map a UML class diagram to an ODMG-ODL schema:

For each class in UML, create a class definition in ODMG-ODL with the same class name. Then examine the UML class specification:

(i) if the isAbstract attribute is false, then add an extent definition to the ODMG class. The name of the extent is the plural name of the class name.
(ii) For each attribute in the UML class with cardinality of the form m..1 (m>=0), define the attribute as single value; otherwise, define the attribute as a set.
(iii) For each operation in the UML class, make a correspondent operation definition in the ODL class.
(iv) For each constraint of the UML class:
   - If the name of the constraint is constrainedattribute, then first make a correspondent attribute definition in the ODL class and with this definition define an operation with unique name. The return type of the operation is Boolean and the input parameters of the operation is the name of the attribute.
   - If the name of the constraint is constrainedmethod, then make a correspondent operation in the ODL class. The return type of the operation is Boolean and the input parameters are the names of the attributes associated with the constraint.
   - If the name of the constraint is class constraint, then create a correspondent relationship definition in the ODMG-ODL class. For example,

   relationship set<Employee>contained_in inverse
   Employee::Employee_collection;

   The related collection class would be define by using the similar class definition as general purpose class.

(v) For each interface in UML, create an interface definition in ODMG-ODL with the same interface name.
(vi) For each generalization between a supertype stype and one or more subtype type1, type2, ………...
   - If stype is a class, then add the EXTENDS relationship from type1, type2, …. to stype.
   type1 extends stype
   type2 extends stype
   - If stype is an interface, then add the ISA relationship from type1, type2, ….. to stype.
   type1: stype
   type2: stype

The correspondence between the extended UML meta model and a standard Object database model developed by ODMG has been established in [12].

4. Representation of constraints using XML

Retrieval and validation techniques developed for XML documents make it a good candidate for retrieval of Object Oriented Framework. It offers a convenient syntax for representing data from heterogeneous sources. To increase interoperability, we have described a mapping scheme from classes and objects with constraints information to XML Schema and XML data in [13]. A brief description of mapping of classes with constraints to XML Schema is as follows.

(i) For each class,
   - Define class as complexType in XML Schema.
   - Define attributes of “class” as Attributes of the complexType .
   - Define Data Type of attributes as built-in or user-defined types in XML Schema.
Introduce an additional attribute cStatus, which indicates whether the complexType represents a
general purpose class or a collection class. Set the appropriate value gClass or colClass for
representing the above.

(ii) We can handle single attribute constraints in two different ways:

   - Create a SimpleType for each constraint.
   - Derive the SimpleType from base type by restriction.
   - Set the range of values for numeric SimpleType using the components minInclusive, minExclusive,
     maxInclusive, maxExclusive etc.
   - For nonnumeric simpleType set the pattern of values using Pattern component.
   - The attribute which has a single constraint must be of the type representing the constraint.

b) Representing constraint methods as attributes
   - Create a complexType for each constraint.
   - Derive the complexType from base type by restriction.
   - Introduce the attribute(s) of the class associated with the constraint as Attribute(s) of the
     complextype.
   - Enumerate with a single value, where the value is the code of the constraint method.
   - Introduce an additional attribute cType, which indicates whether the constraint represents a single
     attribute constraint, a multi attribute constraint or a class constraint. Set the appropriate value
     cAttribute, cMethod or cClass for representing the above.
   - Introduce the name of the class as an additional attribute of the complexType and enumerate with
     a single value where the value is the class name.

(iii) We can represent each constraint method (multi-attribute constraint) as attribute as described in 4.(ii)b).

(iv) For each class constraint,
   - Introduce a constraint method into singleton collection class associated with the user-defined class.
   - Each collection class along with attributes and constraint methods would be represented as
     ComplexType in XML Schema using the same convention as described earlier for other user-defined
     classes.
   - Introduce an additional attribute cStatus, which indicates whether the complexType represents a
general purpose class or a collection class. Set the appropriate value gClass, or colClass for
     representing the above.
   - Introduce the name of the general purpose class as an additional attribute of the complexType and
     enumerate with a single value where the value is the class name.
   - The attributes and the methods of this class are adequately generic and hence can be handled in an
     independent manner.

During interoperability, special care must be taken about the constraint methods represented as attributes in XML
schema.

4.1 An Example of XML Schema with Constraints

This section describes an example demonstrating the mapping scheme described in the previous section, by
translating the “Employee” Class (with Constraints) to XML Schema. For simplicity, we have taken one single
attribute constraint and one multi-attribute constraint out of those provided.

(ii) First, we define a simple type for the Single Attribute Constraints “idC” which represents “Employee Id
must be greater than zero”.

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:simpleType name = "idC">
    <xs:restriction base = "xs:integer">
      <xs:minExclusive value = "0"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```
Correspondingly, the XML Schema for the class “Employee” becomes as follows.

```xml
<xs:complexType name = "Employee">
  <xs:complexContent>
    <xs:attribute name = "empId" type = "xs:idC" use = "required" />
    <xs:attribute name = "eName" type = "xs:string" use="required"/>
    <xs:attribute name = "eDesig" type = "xs:string" use="required"/>
    <xs:attribute name = "empAge" type = "xs:int"  use = "required"/>
    <xs:attribute name = "salary" type = "xs:float"  use = "required"/>
    <xs:attribute name = "experienc" type = "xs:byte" use = "required"/>
    <xs:attribute name = "cStatus" type = "xs:status" use = "required"  />
  </xs:complexContent> </xs:complexType>
```

(ii) We introduce a multi-attribute constraint “empC1” which represents “Employee salary can not be more than $2500 if experience of an employee is less than 5 years”. The translation of the constraint “empC1” is shown in the following.

```xml
<xs:complexType name = "empC1">
  <xs:complexContent>
    <xs:attribute name= "experienc" type = "xs:byte" use = "required" />
    <xs:attribute name = "salary" type = "xs:string" use = "required"/>
    <xs:attribute name = "cType" type = "xs:conType" />
    <xs:attribute name = "className" type = "xs:cName"/>
    <xs:simpleType name = "value"> <xs:restriction base = "xs:string">
      <xs:enumeration value = "body Code for the constraint Boolean method "/>
    </xs:restriction> </xs:simpleType>
  </xs:complexContent> </xs:complexType>
```

The resulting XML schema for the class “Employee has one more attribute “C1” of the type “empC1” as presented next.

```xml
<xs:complexType name = "Employee">
  <xs:complexContent>
    .................
    <xs:attribute name = "cStatus" type="xs:status" use ="required" />
    <xs:attribute name = "C1" type="xs:empC1"/>
  </xs:complexContent> </xs:complexType>
```

The XML application has to identify that the attribute named “C1” is not a general attribute but “C1” denotes an integrity constraint. This can be ensured by a proper nomenclature, possibly by having names from a pr-defined name space.
We can handle Class Constraints in a number of different ways. To deal with constraints representing unique-ness, we can use the built-in facilities of XML Schema such as unique, key, keyref etc. But general class constraints become constraint methods of the collection class associated with the user-defined class. Using the same concept as mentioned earlier 4.1(ii), we can describe class constraints “uniqueC1” which represents “id of an employee must be unique” in “Employee_Collection” class. As before, a new attribute “BC1” of type “uniqueC1” is added in the schema for the class “Employee-Collection”.

5. Interoperability between XML and Java Object Model

Interoperability between XML and Java Object Model requires the tool that transform the XML data from XML Schema representation to Java representation and vice versa. In this context, data binding is the process of mapping the components of a given data, such as SQL table or an XML schema, into a specific representation for a given programming language that depicts the intended meaning of the data format (such as objects, for example). XML Data binding consists of three primary concepts – the specification of XML to Object model bindings, marshalling and unmarshalling. Different binding frameworks will specify bindings differently and typically a data binding framework will have more than one way to specify such bindings. A binding schema specifies details about how classes are generated from XML schema and vice versa. A binding schema also provides facility to specify type conversions, name transformations and specification of super classes for generated objects.

In this context, Castor [8] has three core data binding methodologies: automatic data binding using built-in introspection, user specified data binding and XML schema based data binding with complete source code generation. These methodologies can be used independently or together, which makes Castor a very powerful and easy to use Data Binding Framework. The motivation behind using Castor is powerfulness, simplicity and supporting conversion from XML Schema to Java Programming Language. During translation from XML Schema to Java classes, the Castor Data Binding package generates an associated class definition for each class definition created, called Class Descriptor class. The Class Descriptors are classes that hold the binding and validation information for their associated class and used by the marshalling framework. During transformation for XML to Java, the Class Descriptors also contains the binding and validation information regarding constraints of XML data that are represented as attributes in XML schema.

In this section, we introduced an efficient way of storing XML data with constraint information into Java Object Oriented Programming Language. We have described two approaches to achieve such interoperability. We proposed an extension of the binding file of the Castor Source Code Generator and the Castor XML Mapping file of Castor data binding tool to store the appropriate binding and validation information regarding constraints into class descriptor classes. To implement this, extension of some of the interfaces and classes of Castor data binding tool are also identified and completely specified. In the second approach, we have developed a wrapper-based translator, which can be used for interoperability between XML data with constraints into Java Source Code in addition with Castor Data Binging Tool.

5.1. Extension of the Binding File of the Castor Source Code Generator

The aim of this section is to provide an extension of the binding file of the Castor Source Code Generator for taking care about constraints during translation from XML Schema to Java Source Code. We introduced the extension of the componentBinding element of the binding file is as follows.

\[
<\text{componentBinding}> \text{element} \\<\text{elementBinding}|\text{attributeBinding}|\text{complexTypeBinding}|\text{groupBinding} \text{name} = \text{xsd:string}>\((\text{java-class}|\text{interface}|\text{member}|\text{constraint}),\text{elementBinding}*, \text{attributeBinding}*, \text{complexTypeBinding}*, \text{groupBinding}\)<\text{elementBinding}|\text{attributeBinding}|\text{complexTypeBinding}|\text{groupBinding}>
\]

Here, the first child element (<java-class/>, <interface>, <member> or <constraint>) will determine the type of binding. The following is the brief description of newly introduced element constraint of the binding file of the Castor Source Code Generator.
<constraint> element

The detailed description of the attributes of newly introduced element constraint is as follows.

name – the name of the constraint that will be generated.
ctype – the SchemaType that corresponds to the java type chosen to represent the constraint.
return-type – the return type of the constraint method which is boolean.
constraint-code – the body code for the constraint method.
constraint-type – represents the type of the constraint, whether it is a constrained attribute or constrained method or class constraint.
attribute-list – represents the attributes associated with the constraint.
handler - the fully qualified name of the constraintHandler to use.
validator - the fully qualified name of the constraintValidator to use.

5.2 Extension of the Castor XML Mapping File

The objective of this section is to extend the Castor XML Mapping file, so that the Class Descriptors classes contains the corresponding binding and validation information regarding integrity constraints during two-way interoperability between XML and Java. Castor XML Mapping is a good way to simplify the binding of Java classes to XML document. Castor allows one to specify some of its marshalling/unmarshalling behavior using a mapping file outside the Castor’s default behavior.

(i) First, we have introduced the extension of the class element of Castor XML mapping file, which contains all the information used to map a Java class into an XML document is as below.

<class> element
<! ELEMENT class (description?, cache-type?, Map-to?, field+, constraint+)>
<! ATTLIST class……………. >
where constraint represents zero or more <constraint> to describe the constraint information associated with the Java objects.

(ii) A new element <constraint> is introduced into Castor XML mapping file, which is as below.

<constraint> element
<! ELEMENT constraint (description?, bind-xmlconst?)>
<! ATTLIST constraint name NMTOKEN #REQUIRED
ctype NMTOKEN #IMPLIED
constraint-type( cAttribute | cMethod | cClass) return-type( boolean )
handler NMTOKEN #IMPLIED
direct(true | false) “true”
concreate-method NMTOKEN #IMPLIED >

The following is the brief description of the elements and the attributes of the <constraint> element.

description – an optional <description>.
bind-xmlconst – the name of the XML element or attribute which will be mapped into java constraint.
name – the java constraint name.
ctype – the SchemaType that corresponds to the java type chosen to represent the constraint.
return-type – the return type of the constraint method which is Boolean by default.
constraint-type – represents the type of the constraint, whether it is a constrained attribute or constrained method or class constraint.
handler - the fully qualified name of the constraint handler to use.
direct  - If true, Castor will expect a public constraint method in the object. By default, this is true because all constraint methods should be public.

conscreate-method- Method for the creation or instantiation of constraint.

(iii) The element `<bind-xmlconst>` will be used to describe how a given java constraint should appear in an XML document. It is used both for marshalling and unmarshalling. The element is described below.

```
<bind-xmlconst>

<! ELEMENT bind-xmlconst EMPTY>
<! ATTLIST bind-xmlconst
  name NMTOKEN#IMPLIED
  type NMTOKEN #IMPLIED
  location CDATA #IMPLIED
  matches NMTOKENS #IMPLIED
  node ( attribute | element | text ) />
```

The following is the detailed description of the attributes of the `<bind-xmlconst>` element.

- **name** – the name of the element or attribute of XML Schema.
- **type** - the name of the XML Schema type that requires for specific handling in the Castor Marshalling Frame Work.
- **location** - allows the user to specify the sub-path for which the value should be marshaled to and from.
- **matches** - allows overriding the matches rules for the name of the element. It is a standard regular expression and will be used instead of name attribute. For example, we can use the constraint names as C* or BC*.

To implement this, the brief description of extension of some of the interfaces and classes of Castor data binding tool are identified and completely specified in [15].

### 5.3. Brief Description of the Wrapper-based translator

A translator serves as an inter-mediator between different systems. The translation function is anticipated by implemented as part of a software wrapper. In this section, we introduced a wrapper to convert the Java Source Code produced by Castor ( suppose representation A) into our presentable Java Source Code format (suppose representation B).

A wrapper is a piece of software used to alter the view provided by one interface to another without modifying the underlying component code. The translator must be capable of converting instances of a class’s attributes, methods and constraints (or both attributes, methods and constraints in the form of an object of the class) from representation A to representation B and vice versa. Since we are only concerned about the classes with constraint information, the operational parameters can either be attributes, constraints, methods, objects or their combinations. The wrapper would intercept the parameters and follow the appropriate translation rule to accomplish conversion from representation A to representation B and vice versa. A set of translation rules govern the translations that take place in the wrapper is as follows.

**Rule1.** For every class definition C (either general purpose or collection class) in representation A, a corresponding class C’ is created in representation B. Similarly, for every class definition C’ in representation B, a corresponding class C is created in representation A.

**Rule2.** For every field definition F whose names are not like –C* or _BC* in representation A, a corresponding field F’ is created and added to the corresponding class definition in representation B. Similarly, for every field definition F’ in representation B, a corresponding field F is created and added to the corresponding class definition in representation A.

**Rule3.** For every Method definition M in representation A, a corresponding method M’ is created and added to the corresponding class definition in representation B. For every Method definition M’ in representation B, a corresponding method M is created and added to the corresponding class definition in representation A.

**Rule4.** For every field definition F whose names are like –C* or _BC* in representation A, a corresponding constraint Boolean method is created and added to the corresponding class definition in representation B. Similarly,
for every constraint Boolean Method in representation B, a corresponding field F is created and added to the corresponding class definition in representation A.

The wrapper would invoke the appropriate translation rule to convert the Java Source code from representation A to representation B and also from representation B to representation A. Now, the resultant structure of the wrapper class is as below.

```java
public class wrapper {
    private String className;
    private String[] fieldName = new String[50];
    private String[] methodName = new String[50];
    private int totalNoofFields;
    private int totalNoofMethods;
    private int totalNoofConstraints;
    public void getClassName(BufferedReader br) throws IOException;
    public void getFieldName(BufferedReader br) throws IOException;
    public void getMethodName(BufferedReader br) throws IOException;
    public void getFieldType(BufferedReader br, FileWriter fw) throws IOException;
    public void createClass(BufferedReader br, FileWriter fw) throws IOException;
    public void fieldToField(FileWriter fw) throws IOException;
    public void methodToMethod(FileWriter fw) throws IOException;
    public void fieldToConst(BufferedReader br, FileWriter fw) throws IOException;
    public void constToField(BufferedReader br, FileWriter fw) throws IOException;
}
```

The brief description of the methods of the wrapper classes is as follows:

- `public void getClassName(BufferedReader br) throws IOException`: This method is used for getting the name of the class.
- `public void getFieldName(BufferedReader br) throws IOException`: This method is used for getting the name of the field.
- `public void getMethodName(BufferedReader br) throws IOException`: This method is used for getting the name of the method.
- `public void getFieldType(BufferedReader br, FileWriter fw) throws IOException`: This method is used for getting the corresponding field type.
- `public void createClass(BufferedReader br, FileWriter fw) throws IOException`: This method is used for creating a class definition from representation A to representation B and vice versa.
- `public void fieldToField(FileWriter fw) throws IOException`: This method is used for creating a field definition from representation A to representation B and vice versa and add to the corresponding class definition.
- `public void methodToMethod(FileWriter fw) throws IOException`: This method is used for creating a method definition from representation A to representation B and vice versa and add to the corresponding class definition.
- `public void fieldToConst(BufferedReader br, FileWriter fw) throws IOException`: This method is used for creating a Constraint method definition in representation B from the field definition of representation A and add to the corresponding class definition.
- `public void constToField(BufferedReader br, FileWriter fw) throws IOException`: This method is used for creating a field definition in representation B from a constraint method definition of representation B and add to the corresponding class definition.

For translations involving more than one class definitions, the process must be repeated to translate all the classes of representation A to representation B and vice versa.

### 5.4 An Example of Java Source Code generated by Wrapper

The XML Schema “Employee” with constraint information presented in Section 4.1.(ii) will be mapped into the following Java source code with the help of Wrapper in addition with extended Castor Data Binding Tool.
6. Conclusion

We have introduced a constraint model for Object Oriented data. We have further formalized the model by expressing it using UML and ODL. To increase interoperability, we have described a scheme for representing XML data with constraint information using XML Schema definition Language. In order to interact with data into a more manageable programming language format, a wrapper based translator have been developed to accomplish two-way interoperability between XML and Java object Model. Middlewares such as CORBA, EJB are developed to accommodate distributed objects and to facilitate efficient manipulation of these objects. Therefore, for efficient manipulation of distributed data and objects with constraint information, another important direction of extension of the constraint model in objects distributed over a network maintained by a middlewares such as CORBA or EJB.

References


