Modeling of Constraints in Distributed Object-Oriented Environment

By

CHHANDA RAY

DISSERTATION SUBMITTED FOR THE DEGREE OF DOCTOR OF PHILOSOPHY (ENGINEERING)

JULY 2005

JADAVPUR UNIVERSITY
CALCUTTA – 700032
INDIA
Synopsis of PhD Dissertation
Modeling of Constraints in Distributed Object Oriented Environment

By

Chhanda Ray
R.C.C Institute of Information Technology
Canal South Road, Calcutta -700015, INDIA.
e-mail: chhandaray@hotmail.com

under the Supervision of

(i) Prof. Swapan Bhattacharya
Department of Computer Science & Engineering
Jadavpur University, Calcutta – 700032
INDIA
e-mail: bswapan2000@yahoo.co.in

&

(ii) Prof. Samiran Chattopadhyay
Department of Information Technology
Jadavpur University, Salt Lake Campus
Calcutta – 700098
INDIA
Abstract

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. Integrating data from many heterogeneous sources require interoperability between the data repositories. Consistency preservation in the context of interoperability is also a key concern in distributed object computing. My work attempts to address these two major issues. This work introduces a novel constraint model for object-oriented data. The constraint model is formalized by expressing it using enhanced Unified Modeling Language (UML) and ODMG’s ODL. This enhancement has been done by extending UML metamodel. To increase efficacy of 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. A Wrapper based translator has been developed to accomplish two-way interoperability between XML and Java with constraints. Finally, an XML based unification of data between interoperable object-oriented systems such as Java and C++ has been presented so that efficient queries can be performed in XML environment.

Keywords: Integrity Constraints, 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. On the other hand, 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 has been introduced in [18] to solve the data and operation inconsistency problem in legacy systems.

In this work, my contribution is as follows. First, I have developed a novel constraint model for Object Oriented Data [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, I have formalized the constraint model by expressing it using UML and ODL [12]. To increase efficacy of interoperability, the constraint model is also represented by using a standard web based interoperable medium [13] 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 has introduced [14]. A wrapper-based translator has been developed to accomplish two-way interoperability between XML and Java Source Code. Finally, an XML based unification of data between interoperable object-oriented systems such as Java and C++ has been presented so that efficient queries can be perform in XML environment.
The layout of this synopsis is as follows. The next section describes the constraint model and Section 3 introduces the formalization of the model using UML and ODL. 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 using Castor [8] Data binding Tool. XML based unification with interoperable object-oriented systems such as Java and C++ has been introduced in Section 6 and I conclude in Section 7.

2. Modeling of Constraints

An integrity constraint is a semantic information in an object or a relationship among objects. A constraint specifies a condition and a proposition that must be maintained as true. I have 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.

I have considered three different types of constraints present in an object oriented data model, namely, 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 constrained attributes and constrained methods.
To represent class constraints, I have 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”.

I have demonstrated the application of my constraint model by exploring the possibility of developing a pre-processor that would add validity code to satisfy integrity constraints in the methods defined by the user. The development of such a pre-processor is to be based on a language and platform (such as Java) capable of run-time type identification, reflection and introspection.

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. I have 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 *Constrained Attribute* 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 *name, attributeNames, methodCount*, which specifies the name of the constraint, the names of the attributes associated with the constraint and the total number of constraints.

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

c) *CollectionClass* – The *CollectionClass* class is introduced as a subclass of *Classifier* class of the UML meta model. In addition to all attributes of *Classifier* class, it has three new attributes *className, objectTotal and constraintCount* which represent the name of the user-defined class associated with the *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) *MyClass* – *MyClass* is introduced as a subclass of *Class* class in UML meta model. By the virtue of object orientation, all the attributes of *Class* class will be inherited into *MyClass* class. In addition, it has one new attribute *cclassName*, which represents the name of the *CollectionClass* class associated with the user-defined (*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). 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 efficacy of interoperability, I have described a mapping scheme from classes and objects with constraints information to XML Schema and XML data [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) I have handled 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) I have represented each constrained method (multiple attributes constraint) as attribute of ComplexType in XML Schema 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, I have taken one single attribute constraint and one multi-attribute constraint out of those provided.

(ii) First, I define a simple type for the Single Attribute Constraints “idC” which represents “Employee Id must be greater than zero” using built-in facilities in XML Schema.

```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:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name = "Employee">
    <xs:complexContent>
      <xs:sequence>
        <xs:element name="idC" type="idC"/>
        <xs:element name="status" type="status"/>
      </xs:sequence>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```
I have introduced 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:simpleType name = "conType">
    <xs:restriction base = "xs:string">
        <xs:enumeration value = "cMethod"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name = "cName">
    <xs:restriction base = "xs:string">
        <xs:enumeration value = "Employee"/>
    </xs:restriction>
</xs:simpleType>

<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" use = "required"/>
        <xs:attribute name = "className" type = "xs:cName" use = "required"/>
    </xs:simpleType>
</xs:complexType>
```
The resulting XML schema for the class “Employee has one more attribute “C1” of the type “empC1” as presented next.

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="Employee">
    <xs:complexContent>
      <xs:attribute name="empId" type="xs:idC" use="required" />
      <xs:attribute name="cStatus" type="xs:status" use="required" />
      <xs:attribute name="C1" type="xs:empC1" use="required"/>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```

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.

Class Constraints can be handled in a number of different ways. To deal with constraints representing unique-ness, the built-in facilities of XML Schema such as unique, key, keyref etc. can be used. But general class constraints become constrained methods of the collection class associated with the user-defined class. Using the same concept as mentioned earlier in 4.1(ii), I have described class constraints such as “uniqueC1” which represents “id of an employee must be unique” in “Employee_Collection” class in the following. 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 a tool that transforms 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). Currently, a number of data binding tools are available in the market, but no standardization has been made and this will eventually change.

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.

I have introduced an efficient way of storing XML data with constraint information into Java Object Oriented Programming Language. I have described two approaches to achieve such interoperability. In the first approach, I have proposed an extension of the binding file of 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 identified and completely specified. In the second approach, I have developed a wrapper-based translator, which can be used for ensuring smooth interoperability between XML data with constraints into Java Source Code in combination 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. I have introduced the extension of the componentBinding element of the binding file is as follows.

<componentbinding> element
<elementBinding|attributeBinding|complexTypeBinding|groupBinding name = xsd:string>
((java-class|interface|member|constraint),elementBinding*, attributeBinding*,
complexTypeBinding*, groupBinding)
</elementBinding|attributeBinding|complexTypeBinding|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
$constraint$ name? = xsd:string  ctype? = xsd:schemaType
return-type? = xsd:Boolean constraint-code? = xsd:string
constraint-type? = xsd:string attribute-list? = xsd:string
handler? = xsd:string  validator? = xsd:string />

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.
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 contain 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, I 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
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 [32] 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

`<! ELEMENT bind-xmlconst EMPTY>`
`<! ATTLIST  bind-xmlconst`

    name NMTOKEN #IMPLIED
    type NMTOKEN #IMPLIED
    location CDATA #IMPLIED
    matches NMTOKENS #IMPLIED"
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 Framework.
- **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, I 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 Salient Features 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, I have 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 I am 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.

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 fw1) 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;
}

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.

Public Class Emp
Public idC Idno; Public String Name;
Public cage Age; Public String Designation;
Public Float Salary; Public Byte Experience;
Public cStatus Status;

Public Boolean C1()
{If ((this.Experience <=5) && (this.Salary > 2500))
return false;
else
return true; } }

6. An XML based unification of Interoperable Object Oriented Systems

This section introduces the unification of XML data with constraints into Object Oriented Information Systems such as Java & C++. Java is an emerging general-purpose, strongly typed, class based Object-Oriented Programming Language for developing crucial applications. On the other hand, C++ is also a high performance language that many application developers have relied on for building their most essential applications. However, C++ is a compile time language, which makes it very difficult and complicated to map with dynamic systems like web services and writing scalable threaded infrastructure code. In this context, the unification and transformation of such data is immensely important for many modern applications to satisfy the requirements of different users. The applications can instantiate Java or C++ objects to hold and handle XML instances after generating Java or C++ classes from an XML Schema data type definition with constraints.

In order to facilitate unification between XML and Object Oriented Programming Languages such as Java and C++, efficient applications integration is necessary. The wrapper-based translator has modified to achieve the unification of XML data with constraints into Java and C++, so that efficient queries can be performed in XML
Moreover, consistency of XML constraints can be checked in an object oriented programming language environment such as Java and C++.

7. Conclusion

I have introduced a novel constraint model for Object Oriented data. I have further formalized the model by expressing it using UML and ODL. To increase interoperability, I 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. An XML based unification has also done so that efficient queries can be performed in XML environment. 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, an important direction of extension of the constraint model in objects distributed over a network maintained by a middle wares such as CORBA or EJB.