Automation Method for testing XML/DB/XML Layers

Debarshi Raha, Mohan K Jadhav
IBM India Software Labs, Bangalore, India
debarshi.raha@in.ibm.com, mohan.jadhav@in.ibm.com

Abstract

eXtensible Markup Language (XML) is a very convenient format for exchanging data in an application-neutral way. Most of the middleware products usually produce or consume data in XML format at some interfaces. This data either comes from or needs to be persisted into a relational database. Thus, two end points can be discerned clearly - one is XML input and/or output, and another is the relational database. Same data is converted from one format to another between these two end points. It is usually imperative to assure the accuracy of data at these two end points. Though this accuracy can be verified manually for development purposes, for efficient quality assurance of the product, the manual verification needs to be replaced with automated verification. In this paper, we present an automation technique to verify the accuracy of data at these two end points.

1. Introduction

XML is a simple and flexible format [1] for data exchange in the software application space. The data can be structured in an *XML* format to be sent across interfaces. After validating or processing, the whole or a part of these messages can be persisted to the database for future operations.

It may be the case that, in future point of the application life, there might be a need to construct the same XML data from the database and send it back to the same or another interface again according to business needs. Thus, the quality assurance team might need to validate the accuracy of data at each interface or in the database. This obviously becomes a challenge because of the volume of data that flows in and out of the system, and the variation of data that the quality assurance team needs to test. For example, one system takes XML messages from input interface and persists relevant data into the database. (Figure 1)

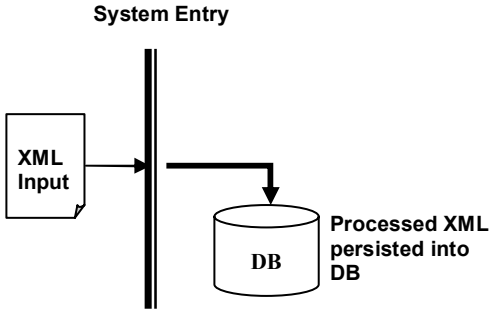

Figure 1. Two end points of the system

The quality assurance team needs to test with all the valid and invalid *XML* messages, and verify whether the data was correctly persisted in the database as expected. If the schema [2] of the input *XML* or the database is complex enough, then manually verifying the database against *XML* input becomes more difficult. And given that regression test would multiply the number of tests that need to be done, automating the tests is a better available option.

2. Problem definition and Solution alternatives

For functional verification or system verification of the system, the quality assurance team does not focus on how efficiently the system processes the input *XML* and persists it into the database, but is interested in the end points of the XML and the database as to the correctness of the data transformation. Thus, the crux of the problem lies in efficient way of comparing the data in these two end points.

There would not have been much problem if any of the proven XML-database technologies [6, 7, 8, 9, 10] were used by the application to persist the XML into the database. But these methods are not always favorable, since this will make the application XML-database dependent, or even worse, database

978-0-7695-3127-4/08 $25.00 © 2008 IEEE
DOI 10.1109/ICST.2008.68

dependent. Recently, many middleware vendors want to make their application database-independent, so that it can run on any relational database. Otherwise, they just might have to package the XML-database too with their product. Here, we assume that relational database is used instead of an XML-supported database.

The schemas for *XML* and database differ in structure: the former is usually hierarchical, while the latter is relational. If a comparison of *XML* data has to be made with the combined data in various tables in the database, several options can be considered:

1) Create the *XML* from the data in the database and compare with the input *XML*.
2) Parse the input *XML* and compare the resulting data with that in the database.
3) Agree on a common object model. Make two sets of objects - one from input *XML* and another from the database and compare those.

For the first option, we can convert relational data into XML Document through object mapping [3, 11], schema conversion [5] or by text formatting. Assuming that the *XML* is generated at the target end point, comparing the *XML* data in the source and target end points may not be easy. For instance, ordering of the data items poses a challenge if a string comparison is to be made. For example, consider the *XML* in Example 1.

```
<products>
    <product>a</product>
    <product>b</product>
</products>
```

Example 1. Source XML

If the system persists product 'b' first into the database and then the product 'a' in the next row, then the created XML from the database might look as given in Example 2.

```
<products>
    <product>b</product>
    <product>a</product>
</products>
```

Example 2. Target XML

In the above scenario, the string comparison will fail.

Choosing the **second option** necessitates parsing the input XML to get the input data, before comparing it against the data in the database. This means that a function of the system pertaining to the parsing of the input XML will be rebuilt in the test system, which makes the option unviable.

In the **third option**, a common object model is needed so as to facilitate the easy and efficient comparison of the data. Before evaluating this option, it will be helpful to look at some of the current *XML* technologies. If a valid schema exists for the *XML*, then this schema can be mapped to a set of classes. There are several XML data binding technologies [3] that can generate classes out of a schema. These classes can be treated as common structure so that we can convert both input *XML* and persisted data in database into these objects and compare them. We then need to evaluate how easy it is to convert XML data and database data into a common set of objects. It is also important to note that the whole *XML* data may not be persisted into database and thus we need control over comparison so we can decide when to assert two objects as equal. We will describe in the following sections how this third option can make the automation process easy.

3. Generating common classes

We assume that all input *XML* conform to a well-defined schema. We can make use of any *XML* schema binding technologies available to generate the classes. These technologies will generate a set of classes according to the schema defined for the *XML*. For instance, see Example 3 for a schema depicting the *Name*.

```
<xsd:complexType name="Name">
    <xsd:sequence>
        <xsd:element name="firstName"
          type="xsd:string" />
        <xsd:element name="lastName"
          type="xsd:string" />
    </xsd:sequence>
</xsd:complexType>
```

Example 3. 'Name' schema

The above schema may be converted to a single Java class:

```
public class Name{
  protected String firstName;
  protected String lastName;

  public Name(String firstName, String
lastName){...}
  public void setFirstName(String
firstName){...}
  ...
}
```

Example 4. 'Name' class

These classes are good enough as a common object model as long as the structure as defined in the schema is well retained in them and they are easy to manipulate.

4. Converting XML into Objects

Generated classes from XML schema are handy to convert an instance of the XML into objects of these classes. The same technology (XML data binding technology) lets us do that. These technologies will read the XML into objects of these classes. For example,

```
<Name>
        <firstName>abc</firstName>
        <lastName>xyz</lastName>
</Name>
```

Example 5. Sample 'Name' XML

For instance, the XML given in Example 5 may be read as '*new Name(abc,xyz)*' ('Name' class defined in Example 4).

So we see that generating common classes and converting XML into objects is easy.

5. Converting data in DB into Objects

Compared to converting XML data in objects, data in the database will not fit that easily into the generated common classes. Though several (O-R mapping) technologies such as Hibernate [12] facilitate object mapping of the databases, the objects created during this process might not be compatible with those created by the XML binding technologies. Thus, we need to handle the conversion in a customized approach as shown below.

Table1

id	firstName	lastName
1	abc	3

Table2

id	lastName
3	xyz

Figure 2. XML persisted into database

For the XML above (See Example 5), 'firstName' and 'lastName' can be persisted into different tables with foreign key relationship (See Figure 2). To fit this data into the 'Name' class (as given in Example 4), a join on these two tables is needed to read the 'firstName' and the 'lastName'. Similar method can be used to initialize all the classes.

An XML document can have only one root element and all other elements are children of this root element, thus making a parent-child relationship. This relationship is applicable for any element. Similarly, the classes generated from an XML schema will also have a parent-child relationship. For example, 'firstName' is child of the element 'Name' (See Example 5). In Example 4, we see that 'firstName' is a property of class 'Name' and there is an association (aggregation) relationship between classes 'Name' and 'String'. So all the generated classes from the XML schema will be associated like parent-child relationship in XML. This pattern is the **key** for converting data from database into objects.

Let us take another schema as given in Example 6.

```
<xsd:complexType name="Person">
  <xsd:sequence>
    <xsd:element name="age" type="xsd:int" />
    <xsd:element name="name" type="Name" />
  </xsd:sequence>
</xsd:complexType>
```

Example 6. 'Person' schema

(Note that the type 'Name' is defined as in Example 3.)

Generated class from this schema may look like:

```
public class Person{
    protected int age;
    protected Name name;

    public void setAge(int age){...}
    public int getAge(){...}
    public void setName(Name name){...}
    public Name getName(){...}
          ...
}
```

Example 7. 'Person' class

(Note that the class 'Name' is defined in Example 4.)

In the above example, there is an association (aggregation) relationship between 'Person' and 'Name' class.

Let us change the tables accordingly as given below:

Table1

id	firstName	lastName	age
1	abc	3	25

Table2

id	lastName
3	xyz

Figure 3. 'Person' schema in database

To load data from these tables into the 'Person' object, we can read the values from the database and call 'set' methods to set the properties of all the classes. However, if the schema is complicated and number of

classes generated is more, then reading from the database and initializing objects, will result in too much of test code for each of the test cases.

Instead, easier method is to use the association (aggregation) relationship between the classes. To do this, let us add a '*load()*' method to each of the classes. **The idea is that the class will load itself**. For example, modified 'Person' and 'Name' class with 'load()' methods are shown in Example 8.

```
public class Name{
  protected String firstName;
  protected String lastName;
  public       void       setFirstName(String
firstName){...}
  public String getFirstName() {...}
  public        void        setLastName(String
lastName){...}
  public String getLastName(){...}

  public void load(int id){
    /*Join Table1 & Table2 and query DB for
the values of 'firstName' & 'lastName' for the
person with given id */
    /*call 'setFirstName()' & 'setLastName()'
to assign the values */
  }
         ...
}

public class Person{
  protected int age;
  protected Name name;
  public void setAge(int age){...}
  public int getAge(){...}
  public void setName(Name n) {..}
  public Name getName(){...}

  public void load(int id){/*load the person
with this id */

    /*query DB for the value of age and assign
it by calling 'setAge()' */
    name = new Name(); /*initialize 'name'.
Association happens */
    name.load(id); //load it
  }
         ...
}
```

Example 8. Generated Classes with 'load()' methods

If 'Person' is the root element in the schema, then the test code just needs to initialize the 'Person' object and call 'load()' method on it. Thus, the test code is just two lines long:

```
Person p = new Person();
p.load(personID);
```

Here, the 'load()' method on 'Person' will load all the properties of 'Person' from the database. If the type of one property is again a schema-generated class, then it will also have load method, which will be called in turn.
For instance, Person.load() calls Name.load().

Thus, it can be seen that all the 'load()' methods will be called in chain from the root of the hierarchy downwards. Eventually all the classes will be initialized and associated. (see Figure 4)

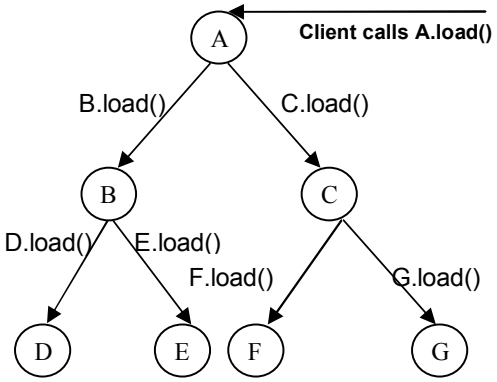

Figure 4. Chain Object Loading

6. Comparing two objects

Now that the data in the XML and the database are converted into a set of common related objects, the final step is to compare them to validate the functional accuracy of the system.

One simpler case is when all the information in the XML is persisted into the database. In this case, our process will result in two same sets of objects, which will be easy to compare with one another. However, most of the times, all the data in input XML might not be persisted in the database. For example, some elements are used for header information or processing instructions. So this information is lost in the database. So, the objects constructed from XML will have that information but objects constructed from database will not have them, thus making the two sets of objects different. So, we need control on when to say two set of objects equal. We can achieve this through defining 'equals()' method in each of the classes generated from the XML schema. This enables a fine control over asserting the objects.

Notice that asserting will also be a chain call similar to the chain initialization though the 'load()' method.
To illustrate, let us take a simple case as shown in Example 9.

```
Class A{
  int i; //simple type
  B b; //complex type

  public boolean equals(Object obj){

          if (obj == null) {
              return false;
          }
          if (obj.getClass() !=
this.getClass()) {
              return false;
          }

          A a=(A)obj;

          boolean result = (i==a.i)  &&
(b==null ? a.b==null : b.equals(a.b))  && ...;

          return result;
    }
}
```

Example 9. Sample class with 'equals()' method

Here, 'A.equals()' calls 'B.equals()' in turn.

For asserting two objects we can make use of existing frameworks, e.g. JUnit framework [13], so that one line of code is required: *Compare(output, input)*.
Figure 7 summarizes the proposed method for automated testing of the XML/database interfaces.

7. Industry Application

The proposed method can be applied to assure quality of any product that satisfies following properties:

1. The product stores data in database in relational format.
2. The product uses XML representation of the data at various interfaces.

For testing large-scale enterprise products where XML schema is complex and testing needs to be done for large volume of data, the proposed method is particularly suitable because it results in generation of common set of classes only once. Once this common set of classes is generated the procedure (converting and comparing) will be same for all kinds of test data.

This method was applied successfully for testing one specific implementation of EPCIS (Electronic Product Code Information Services) enterprise application. EPCIS is industry standard for sharing EPC related data, both within and across enterprises. This is aimed at giving visibility of EPC-tagged objects to the participants of EPCglobal network.

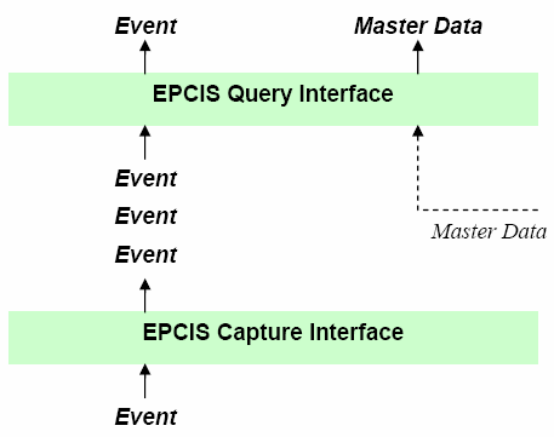

Figure 5. EPCIS capture/query interface[4]

EPCIS standard [4] defines 'Capture Interface' which captures 'Events' from RFID (Radio Frequency Identification) middleware and persists into EPCIS repository. The 'Query Interface' can be used by EPCIS accessing applications for querying 'Events' or master data. So 'EPCIS Event' is input/output for both these Capture/Query interfaces.

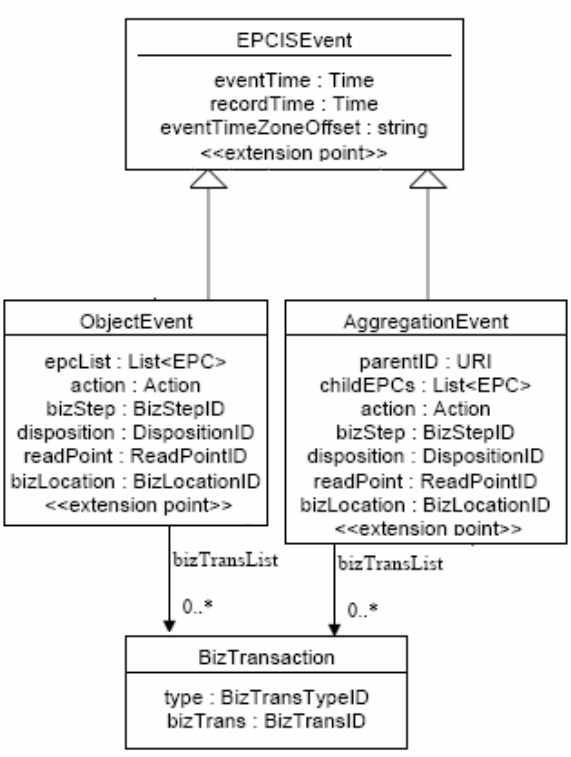

Figure 6. EPCIS Events[4]

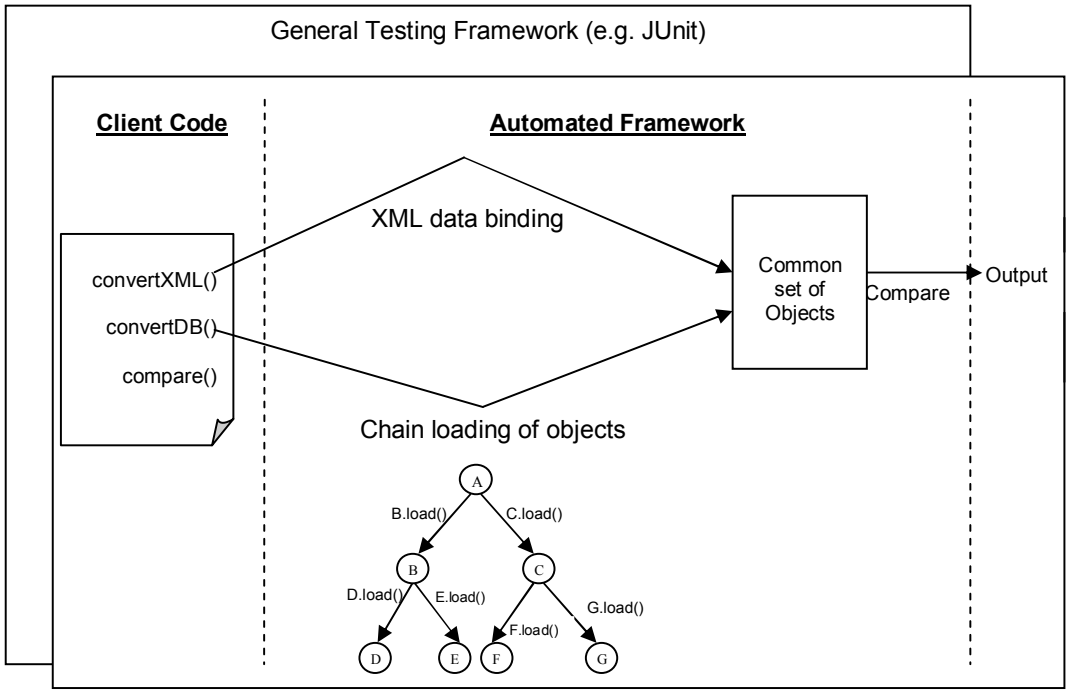

Figure 7. Architecture

EPCIS specification defines XML schema for core event types. We also decide to use relational database model to persist 'Event' data in the system (although EPCIS specification does not mandate that).

Thus, we see that EPCIS application satisfies two basic conditions stated above for applying this automation method for testing the application ('Event' data is stored as relational format in the repository, and same event data is used in capture/query interfaces as XML format).

Now let us look at 'Events'. Figure 6 shows two core event types. These events will be captured and persisted into database using 'Capture Interface' and can be queried via 'Query Interface'. So one interest for the quality assurance team will be to verify accuracy of data between two end points, e.g. XML events captured via 'Capture Interface' and event data persisted into relational database. As this is a known pattern as described in this paper, we can decide on applying this automation method for testing the product.

We used this automation method to verify the capture of approximately 300 events and uncovering around 30 defects. Table 1 below gives more information on a single pass of test execution.

Table 1: Test Execution Report

Number of test cases = 75 Execution time using the automation method = 15 minutes (approximately) Number of defects produced = 30 The above execution was done on the following platform and hardware: Operating system = RedHat Enterprise Linux AS4 Architecture = i386 Java version used = 1.5 RAM = 2 GB Disk Capacity = 80 GB

This method can be applied effectively for all the systems that meet the criteria mentioned in the beginning of this section irrespective of their application in a specific business or industry.

8. Conclusion

An automated way for assuring data accuracy at XML/DB/XML layers has been presented. We made use of one existing XML technology, called XML data binding technology for solving the problem. We converted both the XML and data stored in database into a set of objects and compared them. The set of

common classes were generated in three steps. First, we generated basic classes from the XML schema. Then we added *'load()'* method to all the generated basic classes for chain loading of the objects. Finally, we added *'equals()'* method for fine control over comparing objects.

When these common classes and test data are ready, the test code can initiate the process (feed 'system to test' with XML input or invoke XML output, which can be done programmatically, not described in this paper) and compare output (DB/XML) with input (XML/DB). We can save our test code in some test suite to run the tests again in the future automatically.

9. Trademarks

- Java™ and all Java-based marks are a trademark or registered trademark of Sun Microsystems, Inc, in the United States and other countries.
- Red Hat® is a registered trademark of Red Hat, Inc.
- Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

10. References

[1] Worldwide Web Consortium (W3C), *Extensible Markup Language 1.0 (Fourth Edition)*, Aug, 2006, http://www.w3.org/XML/

[2] World Wide Web Consortium (W3C), *XML Schema*, Oct, 2004, http://www.w3.org/XML/Schema/

[3] Dennis Sosnoski, *XML and Java technologies: Data binding, Part 1: Code generation approaches - JAXB and more*, IBM developerWorks, 2003, http://www-128.ibm.com/developerworks/library/x-databdopt/index.html.

[4] EPCglobal Inc, *EPC Information Services (EPCIS) Version 1.0 Specification*, April 2007, http://www.epcglobalinc.org/standards/epcis/epcis_1_0-standard-20070412.pdf

[5] Joseph Fong, Francis Pang, Chris Bloor, Converting Relational Database into XML Document, In *Proc. of 12th International Workshop on Database and Expert Systems Applications*, 2001.

[6] Haifeng Jiang, Hongjun Lu, Wei Wang and Jeffrey Xu Yu, XParent: An Efficient RDBMS-Based XML Database System, In *Proc. of the 18th International Conference on Data Engineering (ICDE)*, 2002.

[7] Jinyu Wang, Kongyi Zhou, K. Karun and Mark Scardina, Extending XML Database to Support Open XML, In *Proc. of the 20th International Conference on Data Engineering (ICDE)*, 2004.

[8] Ntima Mabanza, Jim Chadwick, and G.S.V.R.Krishna Rao, Performance evaluation of Open Source Native XML databases – A Case Study, *The 8th International Conference on Advanced Communication Technology*, 2006.

[9] Cynthia M. Saracco, Don Chamberlin, and Rav Ahuja, *DB2 9: pureXML Overview and Fast Start*. IBM Redbooks, June 2006.

[10] Sandeepan Banerjee, Vishu Krishnamurthy, Muralidhar Krishnaprasad, and Ravi Murthy, Oracle8i - The XML Enabled Data Management System, In *Proc. of the 16th International Conference on Data Engineering*, 2000.

[11] Muralidhar Krishnaprasad, Viswanathan Krishnamurthy, Ravi Murthy, and Visar Nimani, *Apparatus and method for mapping relational data and metadata to XML*, United States Patent No. US7260585, Aug, 2007.

[12] SourceFourge, Inc., *Hibernate - Relational Persistence for Idiomatic Java*, http://sourceforge.net/projects/hibernate/

[13] Kent Beck and Erich Gamma, *JUnit Test Framework,* www.junit.org/