Processing Data-Oriented XML
Using a Simple and Unified Structure

Csaba Vég, Zsolt Lencse
veg.csaba@xsys.hu, lencse@inf.unideb.hu

XML[1] (Extensible Markup Language), appeared in 1998, has achieved a wide success. After nearly a half decade, XML has become the sole general description format. The designers wanted to make it suitable for describing both data and text. However today XML is used only for describing data, as a resort for storing data and interprocess communication.
The data-oriented XML has a very simple structure. Since the tools available for processing XML are for processing general XML format, it makes complicated to handle the data described by XML. A suitable model and inner structure to represent this sort of XML would exploit the advantages of data-oriented format and simplify data processing.
This paper describes a simplified and unified finite tree model of the data-oriented XML and an inner representation that makes the data processing simpler and faster.

1. Data-Oriented XML

The need for a general descriptor (markup) format has already appeared at the end of the 60's. However, only SGML[2] has become well-known, which was standardized by ISO in '86. The success of SGML is due to the penetration of HTML[3], an SGML format used on web pages. The means of SGML are very complex which made hard to create parser programs. This led to a need for a simplified version. The W3C proposal for the Extensible Markup Language (XML) appeared in '98, which was followed by a fast spread. During the design the most important principles were the textual form, the simple processing and the compatibility with SGML.

The designers wanted to make XML applicable for describing both documents and data. Based on the application, two variants of XML become evolved: a document oriented and a data-oriented form.

The document oriented XML represents the more general form: within an element there could be both text and subelements. In the following example, there is an "i" element between two text elements:

```
<note>
 <author>Joe</author>
 <title>Writing <i>document oriented</i>  XML</title>
</note>
```

With the spreading of XML, the data-oriented XML documents become widely used and are created by or for programs (e.g. for interprocess communication). Below is an example of a data-oriented XML document:

```
<catalog>
  <book id="1091">
    <title>XML - everywhere</title>
    <author>John Smith, Joe Johnson</author>
    <price>27.95</price>
  </book>
  <book id="1341">
    <title>Besides</title>
    <author>George Black</author>
    <price>29.95</price>
  </book>
</catalog>
```

1.a : Data-oriented XML

The data-oriented XML has a simpler structure and conforms to stricter rules.

1. The succeeding text elements (including entities and CDATA sections which are for describing texts) are considered as a single text element.
2. The whitespace characters following element markups are to be ignored.
3. An element can have only one (non-empty) text element or can have only subelements.
4. The comments and the processing instructions are to be ignored during parsing.

It can be observed that the general rules of the document oriented XML are for making documents easier "by hand". Every document-oriented XML can be transformed into data-oriented form easily by the following method, for example:

– text elements and comments, processing instructions etc. can be represented by specially named elements or
– text pieces containing formatting can be represented as literals like this: `<note><![CDATA[Joe is <i>very</i> tall]]></note>`

There is also a W3C proposal called DOM (Document Object Model) [4] for storing XML documents. An arbitrary XML structure can be represented by the DOM structure and it can also be used for storing HTML documents (which can be the result of an XSL[5] transformation). The generality of DOM (see details later), however, makes it extremely hard to process simple data-oriented XML documents and requires a lot of redundant storage space.

The example below is from the tutorial[7p3,7p11] of the Java Web Services Developer Pack® made for the Java[6] language. Consider the price list of coffee beans described by the XML-fragment below:

```
<priceList>
  <coffee>
    <name>Mocha Java</name>
    <price>11.95</price>
  </coffee>
  <coffee>
    <name>Sumatra</name>
    <price>12.50</price>
  </coffee>
</priceList>
```

The purpose of the following example is to add a coffee bean named "Kona" with a price of 13.50 after the "Mocha Java" element. In order to achieve this, first the coffee bean "Mocha Java", that is, a `coffee` element where the first `name` subelement has this value, has to be found.

```
Node rootNode=document.getDocumentElement();
NodeList list=document.getElementsByTagName("coffee");
// Loop through the list.
for (int i=0; i < list.getLength(); i++) {
  thisCoffeeNode = list.item(i);
  Node thisNameNode = thisCoffeeNode.getFirstChild();
  if (thisNameNode == null) continue;
  if (thisNameNode.getFirstChild() == null) continue;
  if (! thisNameNode.getFirstChild() instanceof
                                     org.w3c.dom.Text) continue;
  String data = thisNameNode.getFirstChild().getNodeValue();
  if (! data.equals("Mocha Java")) continue;
```

(Note that the double test at `thisNameNode.getFirstChild()` is unnecessary since the result of the `instanceof` operator is always false for a `null` value.)

At this point the variable `thisNameNode` points to the "`Mocha Java`" coffee bean. Now the new nodes have to be created and appended after the previously found node.

```
  Node newCoffeeNode=document.createElement("coffee");

  Node newNameNode = document.createElement("name");
  Text tnNode = document.createTextNode("Kona");
  newNameNode.appendChild(tnNode);

  Node newPriceNode = document.createElement("price");
  Text tpNode = document.createTextNode("13.50");
  newPriceNode.appendChild(tpNode);

  newCoffeeNode.appendChild(newNameNode);
  newCoffeeNode.appendChild(newPriceNode);
  rootNode.insertBefore(newCoffeeNode,thisCoffeeNode);
  break;
}
```

The code already contains significant simplifications as it is mentioned in the tutorial[7p11,7p231]:

"To be more robust, the sample code described in The DOM API […]*, would have to do these things:*

1. When searching for the <name> element:

a. Ignore comments, attributes, and processing instructions.

b. Allow for the possibility that the <coffee> subelements do not occur in the expected order.

c. Skip over TEXT nodes that contain ignorable whitespace, if not validating.

2. When extracting text for a node:

a. Extract text from CDATA nodes as well as text nodes.

b. *Ignore comments, attributes, and processing instructions when gathering the text.*

c. *If an entity reference node or another element node is encountered, recurse. (That is, apply the text-extraction procedure to all subnodes.)"*

Actually, the processing becomes even more complicated if we need the value of the price, for example to find one of the cheapest coffees. In this case the text representations have to be parsed into numbers.

Due to its generality, the DOM structure makes the processing of XML very hard. It seems to be practical to create an inner structure to exploit the benefits of the data-oriented XML and to simplify the data processing while requiring less storage space.

There are several DOM implementations to support data-oriented XML. The best-known solution is JDOM[8]. JDOM simplifies the processing of XML, though it was designed to process the general XML format, so it cannot fully exploit all the features of the data-oriented XML.

Using JDOM, our example will be more simple, as follows:

```
Element rootNode=document.getRootElement();
List list=rootNode.getChildren("coffee");
// Loop through the list.
for (int i=0; i < list.size(); i++) {
  coffee = (Element)list.get(i);
  if(!"Mocha Java".equals(
        coffee.getChild("name").getText()) continue;

  Node newCoffee=new Element("coffee");
  newCoffee.addContent(new Element("name").setText("Kona") );
  newCoffee.addContent(new Element("price").setText("13.50") );
  list.insert(i+1, newCoffee);
  break;
}
```

2. Finite Tree Model For Data-Oriented XML

The XML format is basically for describing a hierarchy (element-subelements), so it is obvious to model an XML document as a finite tree, that is a finite, directed, acyclic graph. The DOM standard of the internal representation of an XML document follows a tree-like model, too. An XML document is represented as a system of nodes and subnodes in this model. A subelement of an XML element is represented as a subnode. Text elements, CDATA sections, comments and processing instruction are also subnodes.

A DOM node has the following basic properties:

– Name that is fixed for special elements. In case of text elements, the name is `#text`, while it is `#cdata-section for` CDATA sections, and so on.
– A text value. For text elements, it is the text itself, while the value of the attribute for attributes etc.
– Attributes, which can also be regarded as nodes.
– Subelements.

2.a : Modeling DOM as a finite tree

The definition of DOM refers to a finite tree model, where the nodes are assigned the element names. and may have text values, attributes and subelements as well[7p216, 7p242] (see also: [12, 13]).

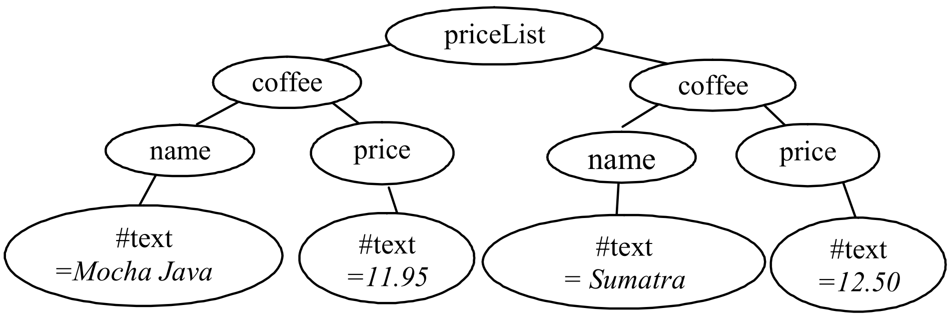

Figure 1. An XML document as a tree

The value assigned to the node is written after the equation sign (=) on the figure, while text elements containing only whitespaces are suppressed.

As data-oriented XML has a simpler structure, we can introduce a simpler model for its representation. For a data-oriented variant, the comments and processing instructions are ignored, yielding in that those will not appear as nodes. Since the adjacent text elements, entities and CDATA sections are regarded as a single text element, an element may have at most one textual element that can be stored in the node of that element.

In the followings, we will suggest a finite tree model for the data-oriented XML.

2.b : Properties of a data-oriented XML element/attribute

* As we've mentioned earlier (see 1.a), az adatorientált XMLszorosabb szabályoknak felel meg, mint az általános ("dokumentumorientált") XML, ill. DOM. A szabályok alapján egy adatorientált XML dokumentum eleme a következő property-kkel írható le:
- név: az elnevezés
- attribútumok listája
- szöveges érték
- alemek listája

Ahol az attribútum a következő property-kkel adható meg:
- név: az elnevezés
- szöveges érték

A két elnevezés összevonható egyetlen node definícióba. Ebben az esetben az "attribútumok listája" és az "alelemek listája" is öszevonható egyetlen "alcsomópontok listája" tulajdonsággá, mindössze biztosítanunk kell, hogy az attribútumok megkülönböztethetők az alelemektől. A megkülönböztetés érdekében az attibútumok nevét kiegészítjük egy kezdő '@' jellel. Note that, the '@' prefix is used by the W3C XPath[9] proposal for attribute names.

2.c : Properties of a data-oriented XML node (element or attribute)

Ekkor a csomópont property-ei a következők:
- név: a csomópont (elem vagy attribútum) elnevezése
- érték: az elemhez rendelt szöveges érték vagy az attribútum értéke
- alcsomópontok (attribútumok vagy alelemek) listája

Az egyesített definíció alapján egy rendkívül egyszerű véges fa modellt alkothatunk az adatorientált XML-ek szerkezeti leírására. A véges fa modellben a DOM-struktúra modelljétől a következő alapkoncepciókban térünk el:
- a név nem a csomóponthoz, hanem a csomópontba vezető élhez van rendelve
- a szöveges érték nem külön alcsomópont(ok), hanem a csomóponthoz van rendelve
- az attribútumok is alcsomópontok,
- attribútumok esetén a név (az él cimkéje) megkülönböztetett ('@' prefix),

Ez az egyszerűsített véges fa modell a következőképpen definiálható.

Első lépésben egy adatorientált XML tartalmát definiáljuk:
The *content* of a data-oriented XML can be specified by a recoursive definition: The content is either an atomic value (e.g. `Mocha Java` or `11.95`) or a list of labeled contents where the names of the individual labels are the names of the corresponding subelements or attributes; and the attributes have a special name format (the '`@`' prefix for example). See the following figure as an example:

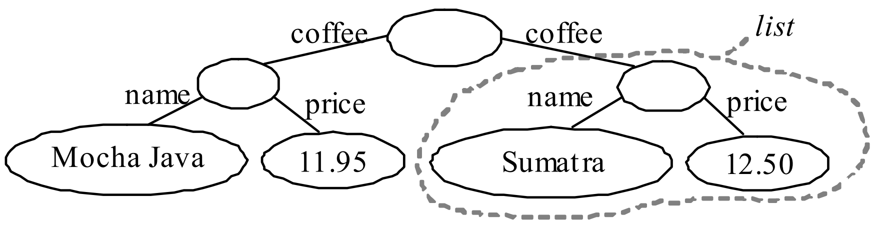

Figure 2. Data-oriented XML as a finite tree

The data-oriented XML and its content is defined as follows:
- A data-oriented XML element is the name (of the top level element) and its content together.

The finite tree model of the data-oriented XML allows both the subelements and the attributes to be considered as the single nodes in the finite tree model. This model extends the rules of the data-oriented XML as follows:
- An element may have an optional text element **and/or** may have one or more optional subelements. For attributes, the text element is the value of the attribute.
- The values of the attributes are optional.
- Attributes may have further attributes or subelements.

3. Complex Values

The best-known model for data modeling is the relational model, however, it features significant restrictions. The extension of the relational model to complex values proposed by Abiteboul, Hull and Vianu [10p508] is suitable for comparison with the model for the data-oriented XML.

To model a complex value, first we define a set of atomic values (called dom, for short of domain) and the constructors of *set* and *tuple*, which can be applied recursively. See the following examples of the authors:

Set	*Complex Value*
dom	A
{dom}	{a,b,c}
<A:dom, B:dom>	<A:a, B:b>
{<A:dom, B:dom>}	{<A:a, B:b>,<A:b, B:a>}
{{dom}}	{{a,b}, {a}, {}}

Using the model of the complex values, the data structure of the relational data model (a relation) can be seen as a result of the *tuple* and *set* constructors taken on atomic values. This model is suitable to describe other extensions of the relational model, such as the nested relational model. In that extension, it is compulsory to alternate the two constructors, the *tuple* and the *set* constructor.

A complex value also can be regarded as a finite tree[10p509], where the leaves are atomic values, the respective constructors are assigned to the subtrees, the edges are the attribute names for the *tuple* constructors, and they are unnamed for *set* constructors. Using the notation of Abiteboul et al., our previous example can be described as follows:

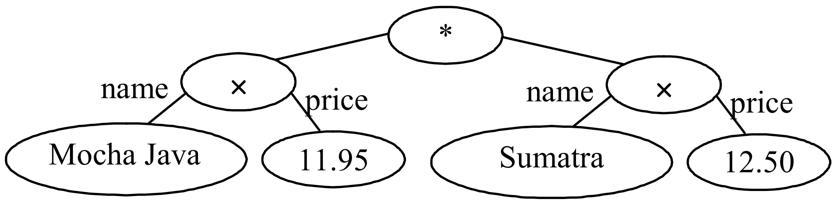

Figure 3. Representation as complex value

Note that, in case of data-oriented XML, multiple edges may have the same name, and it is compulsory to label all edges. In case of elements constructed by the set constructor, we can use arbitrary names, so we can say that:

The data-oriented XML is suitable to represent complex values.

In the practice of using data-oriented XML, these "arbitrary" names (in sets) are repeated names (the `coffee` in our example), wich are the names of the elements of a collection.

4. XML Nodes – The Framework

The finite tree model of the data-oriented XML makes it possible to consider the elements and the attributes in a uniform way: as nodes of a finite tree model.

An XML node can be specified as having the following properties:

– `parent`: `XmlNode`; the node to which this node belongs
– `name`: textual; the label of the edge leading to this node
– `attributes`: `XmlNodeList`
– `value`: optional text
– `subnodes`: `XmlNodeList`

The unified structure makes it possible to handle the attributes and elements in a uniform way. Storage of a reference to the parent makes some of the operations on the structure, like searching the namespace definitions, easier to perform.

In the followings we will show the representation based on the finite tree model by the tools of a programing language to make testing and verification of the model possible in practice. Let's choose Java as the programing language for this representation.

The basic methods to access the node (`XmlNode`) grouped by topic are the followings:

Checking a node's existence:
```
exists(): boolean        // is it an existing node?
```

Parent:
```
parent(): XmlNode        // provides the parent of the node
setParent(x:XmlNode)     // sets the parent of the node
```

Name of the element/attribute:
```
name(): String                      // name of the element
hasName(name :String): boolean//  checks the name against the parameter
```

Collection of attributes:
```
attrs: XmlNodeList                  // provides attributes as a collection
attr(i:int): XmlNode                // provides i-th attribute
attrCount(): int                    // number of the attributes
```

Value:
```
value(): String                     // provides the value of the node
set(value: String): XmlNode!        // sets the value of the node
```

Collection of the subelements:
```
items: XmlNodeList                  // provides the subelemets as a collection
get(i: int): XmlNode                // provides the i-th subelement
count(): int                        // number of subelemets
```

Insertation and remove:
```
addAttr(x: XmlNode): XmlNode!  // appending an attribute
add(x: XmlNode): XmlNode!      // appending a subnode
```

```
remove(x: XmlNode): boolean    // removing a subnode
```

The first method, `exists` allows to handle a non-existing node (let's call it `Nil`) – typically the result of an unsuccesful search – as an ordinary node. The `Nil` has neither attributes, nor textual value and subelements.

The methods which has the return type `XmlNode!` are simple modifiers. These returns a reference to the underlaying object (`this`), so those can be "chained" each after, for example:

```
x.set("12.50").add(y);
```

The attributes and subelements can be handled in a similar way due to the unified model. In order the achieve this, we use the "`@`" symbol as the first character of the attribute name. (This notation is common in the world of the XML technologies.)

The following convenient methods can be easily defined using the basic methods:

`add(name: String): XmlNode`
: Creates and appends an attribute or subelement with the given name. The return value is the new node.

`get(name: String): XmlNode`
: Searches for the attribute or subelement with the given name. If it does not exist, the method returns `Nil`.

`let(name: String): XmlNode`
: Searches for the attribute or subelement with the given name. If it does not exist, it will create a new node with the given name. The return value is the node found or the new node created.

`valueOf(name: String): String`
: Searches for the attribute or subelement with the given name, and returns its value (same as `get(name).value()`).

`set(name:String, value:String): XmlNode!`
: Searches for or creates the attribute or subelement with the given name and sets its value (same as `let(name).set(value)`).

With these methods supporting the processing of data-oriented XML, the code for founding "`Mocha Java`" is less than the half of the original code:

```
for(int i=0; i<priceList.count(); i++){
  XmlNode coffee=list.get(i);
  if(coffee.hasName("coffee")
     && "MochaJava".equals(coffee.valueOf("name")){
```

Since similar searches based on the element name and value combination are frequent, it is useful to define three additional methods:

`get(name: String, String[]prop): XmlNode`
: Searches for the attribute/subelement with the given name and properties. Shouldn't such node exist, the method returns the `Nil` node. The property values are passed in a list of names and values, where the name, of course, can be an attribute name.

`add(name: String, String[]prop): XmlNode`
  Creates the attribute/subelement with the given name and properties.
`let(name: String, String[]prop): XmlNode`
  Searches for or creates the attribute or the subelement with the given name and properties. The return value is the node found or the new node created.

With the new `get` method, the previous search can be shortened to a sole instruction:

```
XmlNode coffee=priceList.get("coffee", new Object[]{
  "name", "Mocha Java"
});
```

We can check for the existence of the element with the given name and properties by the `coffee.exists()` call.

Elements can be created in a similarly simple way. For example, we can easily append a new element to the end of `pricelist`:

```
priceList.add("coffee")
            .set("name","Kona")
            .set("price","13.50")
;
```

We can also use the following form:

```
priceList.add("coffee", new String[]{
    "name", "Kona",
    "price", "13.50",
});
```

The previous example from the tutorial inserted the "`Kona`" coffee after the "`Mocha Java`". In order to achieve such functionality, some more extensions are necessary to access the attributes and subelements as collections. Since the similar tasks are rare using data-oriented XML, in this paper we disregard discussing that issue (the `XmlNodeList`).

5. Summary

The phrase XML nowadays usually refers to the so called data-oriented XML, which conforms to more restrictive rules than the document-oriented format. At the same time, the tools for internal representation (e.g. DOM) process the general XML format, which results in unnecessarily complex programs and unnecessarily large storage space for data-oriented XML.

This paper provides a finite tree model for data-oriented XML and is about an object-oriented framework. The framework exploits the specialities of data-oriented XML and decreases the size of the processing programs and the required storage space in a great extent. The model handles the subelements and attributes in a uniform way. This uniform handling makes the search and creation of elements with given name and property values simpler. Certain additional methods (search for an element with a given name and properties, and its creation if it does not exists) simplify the set-like operations, e.g. collecting information.

The model extends the possibilities of XML, since the attributes can have further attributes and subelements.

The expressive power of the framework built on this finite tree model is shown by the significant decrease of the required processing code in terms of LOC (lines of code).

6. References

[1] *Extensible Markup Language (XML) 1.1.* W3C Recommendation 04 February 2004, edited in place 15 April 2004.
http://www.w3c.org/TR/2004/REC-xml11-20040204/

[2] Charles Goldfarb: *The SGML Handbook.* (Oxford, 1990)

[3] *HTML 4.01 Specification.* W3C Recommendation 24 December 1999.
http://www.w3.org/TR/html4/

[4] *Document Object Model (DOM) Level 3 Core Specification Version 1.0.* W3C Recommendation 07 April 2004.
http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/

[5] *XSL Transformations (XSLT) Version 1.0.* W3C Recommendation 16 November 1999
http://www.w3.org/TR/xslt

[6] Ken Arnold, James Gosling, David Holmes: *The Java™ Programming Language, 3rd ed.* Addison Wesley Professional, 2000.

[7] *The Java Web Services Tutorial 1.3* (December 17, 2003)
http://java.sun.com/webservices/docs/1.3/tutorial/doc/JavaWSTutorial.pdf

[8] Jason Hunter: *JDOM and XML Parsing.* (Oracle Magazine 2002)
http://www.oracle.com/technology/oramag/oracle/02-sep/o52jdom.html

[9] *XML Path Language (XPath) Version 1.0.* W3C Recommendation 16 November 1999
http://www.w3.org/TR/xpath

[10] Abiteboul, Hull, Vianu: *Foundations of Databases.* Addison-Wesley, 1995.

[11] *XML Schema Part 2: Datatypes.* W3C Recommendation 02 May 2001.
http://www.w3.org/TR/xmlschema-2/

[12] Bert Bos: *The XML data model.* (W3C)
http://www.w3.org/XML/Datamodel.html

[13] Ping Guo, Julie Basu, M. Scardina, K. Karun: *Parsing XML Efficiently*. (Oracle TN)
http://www.oracle.com/technology/oramag/oracle/03-sep/o53devxml.html