Generation of Java Beans to Access XML Data

Carl Binding, Daniela Bourges-Waldegg, Stefan G. Hild
IBM Zurich Research Laboratory
8803 Rüschlikon, Switzerland
{cbd, dbw, sgh}@zurich.ibm.com

Abstract

This paper describes a technique to automatically generate a set of Java Beans accessing data encoded according to some XML document type definition. This technique does not support any XML based mark-up, but can be applied to a sufficiently large class of XML compliant mark-up language to make the approach of practical interest. The generated Java Beans can be used as data accessors to deliver dynamic or static content for mark-up pages used in an internet style application. We briefly describe the technology environment into which our solution is embedded and provide an in-depth treatment of the mapping from XML data onto a flattened relational data model, including an example to illustrate our approach.

1 Introduction

The ever increasing popularity of the World-Wide Web (WWW) [12] has generated a variety of technologies to build WWW enabled applications. Generally these applications consist in a set of inter-related (linked) HTML pages which contain data to be rendered to the application's user. Output data can be of *static* or *dynamic* nature and one of the issues in building WWW applications is to access and include application data dynamically[1] from a specific data source. In the early days of the WWW, *cgi* scripts were used as dynamic data accessing entities where the scripts relied upon an appropriate mechanism to access the relevant data and insert it into the HTML document generated on-the-fly. Hence, *cgi* scripts trigger the execution of arbitrary processes implemented via a scripting language or some other programming formalism.

In the more recent evolution of WWW technology, the *cgi-bin* mechanism becomes replaced with Java *servlet* technology [1] which essentially consists in Java classes implementing the *servlet* interface. Such servlets are invoked from an HTTP server's servlet execution engine. HTTP request arguments are passed into the servlet as an instance of Java class *HTTPRequest* and the servlet's *process()* method is free to use any Java language construct and API to generate the HTTP response, represented as an instance of Java class *HTTPResponse*.

A further Java based technology used for WWW style applications is the Java Server Pages (JSP) mechanism [2]. JSPs essentially embody mark-up templates which contain an escape syntax to include Java code fragments, amongst which the most relevant ones are Java Beans property access methods [5]. The JSP page-compiler translates these templates into Java servlets whose generated output consists of the mark-up data into which the *String*-typed values of the referenced Java Beans properties are inserted. Since the Java Beans' property access methods can perform arbitrary computations, the output thus generated from a JSP is dynamic in nature.

From an application developer's point of view, JSPs combined with Java Beans divide the application into the user interface aspects embodied in the JSP's mark-up and the data access encapsulated in the Java Beans components. A more modular design of the application thus becomes possible with the well-known advantages of such modularization such as re-use of components, ease of maintenance, etc.

JSPs are not restricted to the Internet's mark-up language, HTML. The mechanism is sufficiently generic to handle other mark-up languages; in particular we have used it to build a Wireless Application Protocol (WAP) [10] application using WAP's Wireless Markup Language (WML) [11] as the mark-up language to create the application's user interface. The overall architecture of such an application can be represented as in figure 1. We have shown a WAP application architecture which is similar to any internet application when considering the mark-up data generation.

One noteworthy aspect of this application architecture is the division between the user interface components respon-

[1] By *dynamic* we mean that the evaluation of the data value is triggered by the HTTP request and not performed during creation of the mark-up document.

0-7695-0577-5/00 $10.00 © 2000 IEEE

sibles for data rendering and input gathering embodied by the JSP templates and the data collection and preparation aspects encapsulated within the Java beans components. The application logic is present in both components; however the user interface aspects are distinct from the required data processing aspects.

In our sample application, Java Beans are used to read data from a file based repository and to interface into a relational database. In the future, more and more WWW data will be represented in an Extensible Markup Language (XML) [8] complying format, with XML acting as an unifying data representation formalism for internet application data. It thus becomes of importance to access XML formatted data from within JSPs through the Java Beans or some other technology.

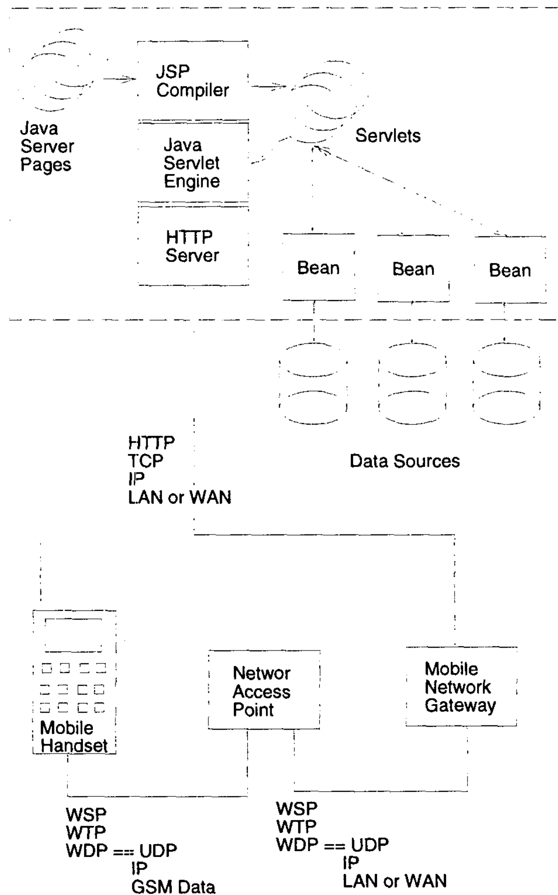

Figure 1. WAP Application Architecture

Our approach differs from other XML related technologies, such as XSLT [9] and DOM [7]. DOM provides a programming interface to hierarchically structured markup language in which the hierarchical nature of the data remains exposed. Thus, an application needs to explicitly navigate through the DOM tree to locate the information contained in an XML document. This approach appears complex for the application programmer, requires reprogramming of the application when a DTD's structure evolves, and cannot easily be integrated into the JSP mechanism which supports only Java Beans style interfaces. Although some layer of Java Beans code could provide the linkage between JSP and DOM this layer still requires the mentioned tree navigation and information location logic.

XSLT does not provide a programmatic interface to the XML data; it is a transformation mechanism which maps one XML structure onto some other XML structure. The mapping is controlled through the *XSLT templates*; however the result of the mapping remains a mark-up document. Thus, integration into JSPs to generate dynamic mark-up documents cannot be achieved through the usage of XSLT.

The XML data-binding facility proposed in [3] exhibits certain similarities with our approach as it maps XML elements into Java classes. These classes are not strictly conforming to the Java Beans framework[2]. The hierarchical nature of XML data is not fully addressed: although nesting of elements can be handled by using *get* and *set* methods which return arbitrary Java objects and not just *Strings* as in our scheme, it is unclear how repeated occurrences of nested elements are addressed. Furthermore the proposal requires the availability of an XML schema to support data integrity validation whereas in our method we avoid this dependency and operate only with generic *String* values.

The remainder of this paper proposes our methodology which enables the automatic generation of Java Beans the *get* methods of which allow the read-access of XML formatted data. Section 2 reviews the XML formalism and is

[2] Although exhibiting a very similar interface.

followed by a cursory presentation of the Java Beans technology (Section 3). The procedure for flattening hierarchically structured XML data onto a relational data model on which the generated Java Beans rely is described in section 4 and an explanatory example is given in section 5. The paper concludes with a discussion of the proposed method in section 6.

2 The Extensible Mark-up Language

The Extensible Mark-up Language (XML) [8] is a meta-language which allows to specify a concrete mark-up language[3]. The syntax of the concrete mark-up language is contained in the so-called *document type definition* (DTD). Various XML compliant mark-up languages have been defined, such as WML, VoiceML, SDML, etc.

XML supports the definition of mark-up *elements* which represent structuring entities identified by an opening and closing tag. Elements can be augmented with *attributes* which may have default values and whose presence can be required or optional. Elements in turn can contain other elements and XML allows to specify hierarchical nesting using *sequencing* and *choice* constructors.

The document's information is present in the form of an element's *attributes* as well as *character data* contained within an element. Thus, from an information encoding point of view, the data content is only present in attributes and as character data. The nesting of elements reflects the data's structure.

Figure 2 shows a typical XML document structure as a tree of attributed elements, some of which contain character data. A potential syntax for the XML tree shown in Figure 2

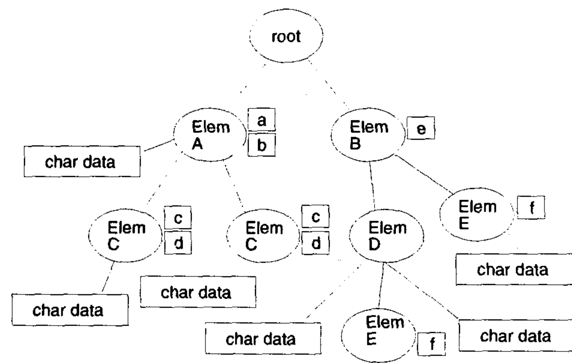

Figure 2. XML Data Structure

[3]XML thus is similar to Extended Backus-Naur Format (EBNF) for programming language specification.

would be[4]:

$$\begin{array}{rcl} root & = & (A, B) \\ A & = & (PCDATA \mid C)\star \\ B & = & (D, E) \\ C & = & (PCDATA) \\ D & = & (PCDATA \mid E)\star \end{array}$$

and the elements have following attributes:

$$\begin{array}{rcl} A & : & \{a, b\} \\ B & : & \{e\} \\ C & : & \{c, d\} \end{array}$$

Element D has no attributes.

3 Java Beans

Java Beans are Java components which provide a codified set of access methods to *set* and *get* the value of *properties*. A bean's properties have *names* and the naming convention for access methods is to concatenate the operation with the property name. For example, a bean property *prop* would be accessible through the access methods *getProp()* and *setProp(String value)*[5]. Properties can also represent array of values; this is termed *indexed* properties for which the access methods take an additional indexing parameter. Indices out of bound are indicated by raising the Java exception *ArrayIndexOutOfBoundException*.

Within the JSP context, Java Beans form the link between the mark-up template and some, possibly dynamically changing, data source through the convention-based Beans interfaces. How a given beans obtains the actual value for its properties is entirely open to the beans implementor; the entire Java language and its supporting APIs can be used to fill the bean's properties.

4 Mapping XML data onto Java Beans

XML data is hierarchical in nature; the hierarchy expresses a containment relationship. The class of Java Beans used within JSPs and in which we are interested in here is limited to beans which have a set of *String* typed properties and thus are similar to data *tuples* as known in relational data models and where each tuple element is of type *String*[6].

Our approach of flattening the hierarchical data has been driven by the desire to keep the structure of the Java Server Pages that use the generated beans simpler. Therefore we did not chose to reflect the data's hierarchy also in the beans

[4]We use the XML notation *PCDATA* to denote (parsed) character data.

[5]Assuming the property value is read/write.

[6]Clearly all data types can be represented as textual *String* data, thus generalizing our approach to all data types.

hierarchy, where nested XML elements would correspond to Java Beans typed properties.

We therefore have chosen to map the hierarchically structured data onto a flat, tuple like data model. In addition, this flattening should be performed automatically, i.e. generation of Java Beans must be an automated process driven by the structural definition of the XML data contained in the DTD associated with the actual data. Generating XML accessing beans by hand represents a cumbersome, error-prone, and thus inefficient method; automatic beans generation is thus highly desireable.

Our method to perform this task is based on the following observations:

- The XML data model can be split into *structure* and *information*. The structure is given by the nesting of the elements, whereas the information content is contained in attributes and character data.
- We can consider a nested element to inherit all information content from its containing element[7]. This is particularly true for XML DTDs which implicitly represent a hierarchical data structure. We use this observation, to flatten the XML structure in establishing a *join* in the relational calculus sense between the nesting element and its nested elements [6].
- We restrict ourselves to XML structures which do not contain mixed content, i.e. an element either contains only character data or only one or several other elements, but not both *PCDATA* and other elements. Elements can be attributed, except *PCDATA* only elements. (Note that these restrictions are not fundamental in nature; they merely support a somewhat simpler naming scheme for bean properties.)
- A further restriction consists in disallowing recursion in the element structure. That is, a given XML element E may not nest another element from which E can be derived: a flattening on such recursive structures would be unbounded.

Formally, let $A = \{a_1, a_2, \ldots, a_n\}$ be the set of attributes for an element E. Furthermore, let the element E be an XML element containing elements $E_1, E_2, \ldots, E_m$ consisting of only character data (PCDATA). No beans are created for $E_1, E_2, \ldots, E_m$; instead we create properties with the element name to represent the element. For E's attributes we create one property per attribute. Thus, the bean corresponding to element E will have properties

1. $a_1, a_2, \ldots, a_n$
2. $E_1, E_2, \ldots, E_m$

[7] We use the term *inherit* similarly to its usage in attribute grammars [4]

all of which are of Java type *String*.

This algorithm can be applied recursively moving downwards the XML hierarchy. Assume a child node C of E. We create a beans E_C which has all the properties of E as well as the properties derived from XML attributes and *PCDATA* sub-elements of C. This process can be repeated until the leaves of the XML hierarchy are reached.

If E contains nested elements of different types say F and G, we furthermore create one bean type per child node type, i.e. E_F, E_G, etc.

The above algorithm is equivalent to performing a *join* in a relational database sense over the nested XML data. All descendants of a given XML elements, inherit the data properties consisting of the element's attributes and *PCDATA* sub-elements, i.e. we implicitly build the cartesian products of $\mathit{Properties}(E) \times \mathit{Properties}(C)$.

Our restriction of allowing non-mixed XML elements which nest both, *PCDATA* and other elements is not strictly necessary if one were to use a slightly more complex naming scheme for *PCDATA* only elements. This would also avoid the restriction that *PCDATA* only elements do not have attributes.

The properties of the created Beans are indexed in order to support repeated occurrences of identically labelled paths through the XML data hierarchy. As an example, refer to figure 2, in which two instances of bean type *A_C* occur. The access to property *d* would thus be *A_C.getD(int i)* and reflects our approach to *join* the data elements along paths of the XML DTD hierarchy. Each uniquely labelled path corresponds to one type of data tuple of which there can be zero or multiple occurrences in the actual XML data instantiation.

In our approach thus, the JSP which makes use of the XML data accessed through the generated beans looks as follows:

```
<repeat index=i>
  ...
  <%= a_c.getD(i) %>
  ...
</repeat>
```

There is only one iteration over all the tuples of beans type *A_C*.

The alternative approach mentioned above, i.e. to have bean properties returning instances of other bean types would require the introduction of additional, nested iteration loops. The above example to iterate over all the instances of *A_C* would be

```
<repeat index=j>
  <% C c = a.getC( j); %>
  <repeat index=i>
    <insert bean=c property=D(i)></insert>
  </repeat>
</repeat>
```

Forming the *join* over the XML element hierarchy is an algorithm with exponential complexity over the depth of the XML hierarchy. We do use constraints to cut unnecessary branches in the XML element tree: the constraints are formulated as a conjunction of attribute values, i.e. $a_1 = v_1$ AND $a_2 = v_2 \ldots AND$ $a_n = v_n$. Branches in the XML hierarchy that do not match these attributes are then pruned from the *join* operation described above. A more complex constraint language can be envisioned, but is currently limited by the format of the URL identifying the XML data: we use the query arguments to create above filtering constraint and append it to the URL when querying the data source.

5 Example

To illustrate our approach for automatically generating Java Beans from an XML DTD, consider the following DTD:

```
<!ELEMENT AvOut ((Itinerary)* | Error)>
<!ELEMENT Itinerary (Time, Flight+)>

<!ELEMENT Flight (Flightnbr, Departure, Boardpoint,
                  Offpoint, Arrival, Airline,
                  Equipment, Class+)+>
<!ELEMENT Class (Designator, Status)>

<!ELEMENT Time (#PCDATA)>
<!ELEMENT Airline (#PCDATA)>
<!ELEMENT Flightnbr (#PCDATA)>
<!ELEMENT Boardpoint (#PCDATA)>
<!ELEMENT Offpoint (#PCDATA)>
<!ELEMENT Departure (#PCDATA)>
<!ELEMENT Arrival (#PCDATA)>
<!ELEMENT Equipment (#PCDATA)>
<!ELEMENT Designator (#PCDATA)>
<!ELEMENT Status (#PCDATA)>
<!ELEMENT Error (#PCDATA)>
```

The sample DTD describes the availability of flights for a given air travel itinerary and has been used in joint project between IBM Research Zurich and an airline IT provider. Our Java Beans generating tool can be used to start the flattening process described in section 4 at different levels in the XML hierarchy, i.e. the implicit *join* can originate at the XML elements *AvOut, Itinerary*, or *Flight.* The Java Beans code generated is shown below. First we show the code generated for a Java Beans *AvOut* followed by the code for Beans *Flight.*

```
package xmlBean;

import ....

public class AvOut extends XMLBeanBox
  implements java.io.Serializable {

    public static String[] atts =
      { "Itinerary_Time",
        "Itinerary_Flight_Flightnbr",
        ... };

    public AvOut () {
      super(
        "http://w3.zurich.ibm.com/~cbd/wdgw.xml",
        "http://w3.zurich.ibm.com/~cbd/" +
        "flights_availability.dtd", "AvOut");
      super.addAttribute(this, "AvOut", atts);
      super.setUseFullyQualifiedPropsName(false);
      super.setIncludePropsIntoReqURL(false);
      super.setProxy(null);
    }

    public void setItinerary_Time(String val) {
      super.setAttribute("Itinerary_Time", val);
    }
    public void setItinerary_Time(int i, String val) {
      super.setAttribute(i, "Itinerary_Time", val);
    }
    public String getItinerary_Time() throws .... {
      return super.getAttribute("Itinerary_Time");
    }
    public String getItinerary_Time(int i) throws .... {
      return super.getAttribute(i, "Itinerary_Time");
    }
    public void setItinerary_Flight_Flightnbr(String val)
    {
      super.setAttribute("Itinerary_Flight_Flightnbr",
                         val);
    }
    public void setItinerary_Flight_Flightnbr(int i,
                                              String val)
    {
      super.setAttribute(i,
                         "Itinerary_Flight_Flightnbr",
                         val);
    }
    public String getItinerary_Flight_Flightnbr()
    throws ....
    {
      return super.getAttribute(
        "Itinerary_Flight_Flightnbr");
    }
    public String getItinerary_Flight_Flightnbr(int i)
    throws .... {
      return super.getAttribute(i,
        "Itinerary_Flight_Flightnbr");
    }
}
```

A few observations are necessary to understand above code fragment in which we have removed most of the property accessing methods since they all are built using the same scheme as well as the declaration of thrown Java *Exceptions*:

1. Inheritance from class *XMLBeanBox*: we use this class to hold all data structures needed by the actual bean, as well as to support various methods used by all bean types we generate.

   Essentially the super-class holds references to all bean instances of a given type; each clone corresponding to one XML element encountered in the XML data.

2. The array *atts* holds the string name of all the properties for this beans and is used to create an array-like data structure holding attribute-value pairs in the bean's super-class *XMLBeanBox*.

3. The bean's constructor invokes the super-class' constructor to indicate the origin - i.e. the URL - of the XML data (the first argument) and also indicates the DTD of which the data is an instance (the second argument). Notice that the bean's data is fetched lazily and is requested over the internet[8] only once the appli-

[8] Via Java's well known *URL* class.

cation attempts to read-access one of the bean's properties via a *get*-property method.

Indication of the DTD's URL allows to validate the obtained XML data: it must match the structure specified by the DTD in order to "fit" a particular bean derived from the DTD.

The other methods contained in the constructor are used to control the shape of the HTTP request to fetch the data. The HTTP request is created from the data source's URL concatenated with the property-value pairs of those beans properties which have been *set* by the application in the originally instantiated *template bean*. (By *template bean* we mean an instance of a bean which does not contain any data from the source indicated via the above URLs; it only contains property values set by the application.). We thus build up an URL of the form

```
http://<host>/<path>?<propA=valA>
                    &<propB=valB>
```

assuming that the beans types has properties *propA* and *propB* set to the corresponding values which can be taken into account at the data source to filter out unnecessary data.

4. The *getProperty()* and *setProperty()* methods: these are methods to access the properties of a particular beans instance.

   When the application first instantiates the *template bean*, the *set...()* methods are used to set some property values which are subsequently used as parameters concatenated to the data's URL when requesting the actual XML data.

   After the data has been obtained, an XML parser traverses this data and creates one beans instance for each corresponding XML element; these clones cross-reference each other by using features of their superclass *XMLBeanBox*.

5. The *getProperty(int i, ...)* and *setProperty(int i, ...)* methods are used to traverse the set of bean clones; starting from the initial template bean to which the application holds a reference. For the application, these interfaces behave as indexed beans properties in-line with the Java Beans specifications [5].

   Individual property values are then read or written by using the non-indexed *getProperty()* respectively *setProperty()* methods of a specific beans instance.

The above beans contains all the data obtained by flattening the XML hierarchy. In reference to the relational data model, the *AvOut* beans contains a set of *Itinerary* elements or one *Error* element. The *Itinerary* is formed by the join of $Time \times Flight$ where the *Flight* elements in turn can be considered to be the *join* $(Flightnbr, Departure, \ldots, Equipment) \times Class$.

This structure of the *Flight* element is apparent in the below code fragment of the *Flight* beans. Its structure is similar to the one of *AvOut* and thus not commented further. During the beans data loading process the flattening merely is started at a lower level in the XML hierarchy when parsing the XML data.

```
package xmlBean;

import .....

public class Flight extends XMLBeanBox
implements java.io.Serializable {

    public static String[] atts = {
       "Flightnbr", "Departure",
       "Boardpoint", ..., "Class_Status" };


    public Flight () {
      super(
         "http://w3.zurich.ibm.com/~cbd/wdgw.xml",
         "http://w3.zurich.ibm.com/~cbd/"+
         "flights_availability.dtd", "Flight");
      super.addAttribute(this, "Flight", atts);
      super.setUseFullyQualifiedPropsName(false);
      super.setIncludePropsIntoReqURL(false);
      super.setProxy(null);
    }

    public void setFlightnbr(String val) {
      super.setAttribute("Flightnbr", val);
    }
    public void setFlightnbr(int i, String val) {
      super.setAttribute(i, "Flightnbr", val);
    }
    public String getFlightnbr() throws .... {
      return super.getAttribute("Flightnbr");
    }
    public String getFlightnbr(int i) throws .... {
      return super.getAttribute(i, "Flightnbr");
    }

    public void setClass_Status(String val) {
      super.setAttribute("Class_Status", val);
    }
    public void setClass_Status(int i, String val) {
      super.setAttribute(i, "Class_Status", val);
    }
    public String getClass_Status() throws .... {
      return super.getAttribute("Class_Status");
    }
    public String getClass_Status(int i)
    throws .... {
      return super.getAttribute(i, "Class_Status");
    }
}
```

6 Conclusion

We have presented a mechanism which supports the automatic generation of Java Beans code from an XML DTD. It is currently limited to a class of DTDs in which elements can nest PCDATA only elements or other XML elements, but not both. This limitation is due to JSP's preference of *String* valued properties for Java Beans. Our scheme can easily be extended to support properties which return an-

other Java Beans class which in turn correspond to a particular XML element.

The efficiency of the approach is limited through the necessity of creating the *join* over the XML hierarchy, thus replicating data unnecessarily.

The main advantages of the proposed methods are the ability to automatically derive the Java Beans code and the enablement of accessing XML data through a programming interface obeying the well-known Java Beans programming model.

7 Acknowledgment

We are grateful for discussions with our colleagues at IBM Research Zurich, in particular Andreas Schade has suggested various improvements for this paper.

References

[1] James D. Davidson and Suzanne Ahmed. *Java Servlet API Specification*. SUN Microsystems, November 1998. Version 2.1a, http://www.javasoft.com/products/servlet/index.html.

[2] E. Pelegri-Llopart, L. Cable, and S. Ahmed. *Java Server Pages Specification*. SUN Microsystems, November 1999. Version 1.1, http://www.javasoft.com/products/jsp/download.html.

[3] Mark Reinhold. An XML Data-Binding Facility for the Java Platform. http://web2.java.sun.com/xml/white-paper.html, July 1999.

[4] Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental Context-Dependent Analysis for Language-based Editors. *ACM Transaction on Programming Languages*, 5(3):449–477, July 1983.

[5] SUN Microsystems. *JavaBeans*, July 1997. Version 1.01, http://www.javasoft.com/beans/doc/spec.html.

[6] Jeffrey D. Ullman. *Principles of Database Systems*. Computer Science Press, 1982. Second edition.

[7] W3C. *Document Object Model (DOM) Level 1 Specification*, October 1998. Version 1.0, http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001.

[8] W3C. *Extensible Markup Language (XML)*, February 1998. Version 1.0, http://www.w3.org/TR/REC-xml.

[9] W3C. *XSL Transformations (XSLT)*, November 1999. Version 1.0, http://www.w3.org/TR/1999/REC-xslt-19991116.

[10] WAP Forum. *Wireless Application Protocol: Wireless Application Environment Overview*, August 1999. version 1.1.

[11] WAP Forum. *Wireless Application Protocol: Wireless Markup Language Specification*, August 1999. version 1.1.

[12] Erik Wilde. *Wilde's WWW: Technical Foundations of the World-Wide Web*. Springer Verlag, 1999.