On XML Objects

Martin Kempa
Universität zu Lübeck
Institut für Informationssysteme
Osterweide 8
D-23562 Lübeck, Germany
kempa@ifis.uni-luebeck.de

Volker Linnemann
Universität zu Lübeck
Institut für Informationssysteme
Osterweide 8
D-23562 Lübeck, Germany
linnemann@ifis.uni-luebeck.de

ABSTRACT

In todays web applications we face the problem that there is the world of HTML and XML on one side and the world of objects (primarily Java objects) on the other side. Programs generating XML and HTML, for example Java Servlets, either have to generate and analyze XML on a string-basis which is rather tedious or have to generate object structures in the document object model or in JAXB. This requires to switch between XML strings and corresponding objects manually by programming. Moreover, the object structure is not guaranteed to conform to an underlying document type definition DTD or XML schema. Although the goal of JAXB is to guarantee this validity, it is only achieved up to a certain extent. In many cases, expensive runtime testing of validity is necessary by using a validation method provided by JAXB. Moreover, in JAXB XML strings and XML objects are two different things requiring to switch between these two notions by methods called marshalling and unmarshalling.

In this paper we propose that in object oriented programming with XML there should be no distinction between XML documents and XML objects. In other words, XML in an object oriented program always denotes XML objects, i.e. generating and analyzing XML is done conceptually only on the basis of objects.

We propose, similarly to JAXB, to have a class for every element type of a DTD or an XML schema. In contrast to JAXB, these classes are defined such that the generation of XML objects is done in a syntax oriented manner allowing to check the validity of all generated XML structures, i.e. XML objects, statically by the compiler. We believe that by eliminating the difference between XML objects and XML documents and by introducing absolutely type safe tools for generating XML objects, programming of web applications, i.e. Java Servlets, is much easier, much safer and much less error-prone.

1. INTRODUCTION

Almost every web application is concerned with generating HTML or XML structures. In todays tools for this purpose, there is a serious mismatch between objects in an object oriented programming language and HTML- and XML-documents. At the time being, a programmer has to switch between these two worlds. He can either generate HTML or XML directly as strings which is rather tedious because the structure is buried in the markup tags or he can generate object structures facing the problem to manually generate a string afterwards.

When using current tools like Java Servlets [27], Java Server Pages [15, 7] or JAXB [18], the object structure is not guaranteed to conform to an underlying document type definition DTD [1] or XML schema [24]. Instead, the validity must be “proven” dynamically by appropriate test runs. The Java Server Page given in Listing 1 illustrates this problem.

```
<HTML>
  <HEAD> <TITLE> A Simple Server Page </TITLE>
  </HEAD>
  <BODY>
    <% if (timeOfDay() == "AM") { %>
      <UL>
        <LI> Good Morning </LI>
      </UL>
    <% } else { %>
      <UL>
        <LI> Good Afternoon </LI>
      </UL>
    <% } %>
  </BODY>
</HTML>
```

Listing 1: Java Server Page

Besides being quite cumbersome, the language mechanism of Java Server Pages does not guarantee the validity of the generated HTML phrase. For example, changing the program to the one given in Listing 2

```
<HTML>
  <HEAD> <TITLE> A Wrong Server Page </TITLE>
  </HEAD>
  <BODY>
    <% if (timeOfDay() == "AM") { %>
      <UL>
        Good Morning
      </UL>
    <% } else { %>
      <UL>
        Good Afternoon
      </UL>
    <% } %>
  </BODY>
</HTML>
```

Listing 2: Incorrect Java Server Page

still results in a correct Java Server Page in the sense that the Server Page processor and the Java compiler accept the program although the program does not generate correct HTML. When using Java Server Pages, problems of this kind have to be found dynamically by appropriate test runs.

Although the goal of JAXB [18] is to guarantee this validity, it is only achieved up to a certain extent. In many cases, expensive runtime testing of validity is still necessary. Moreover, in JAXB

XML strings and XML objects are two different things requiring to switch between these two notions by methods called marshalling and unmarshalling.

The main objective of the work presented in this paper is to overcome the difference between XML documents as strings and corresponding objects in an object oriented programming language like Java [3]. In other words, when using XML in an object oriented program, it always denotes XML objects, i.e. generating and analyzing XML is done conceptually only on the basis of objects. Nevertheless, this does not mean that the programmer has to generate the object structure of an XML document manually. He can use XML parts as texts with markup in his program and insert other XML parts in appropriate places. This text denotes objects very much like the string 10 denotes the natural number ten.

Similarly to JAXB, we introduce a class for every element type of a DTD or an XML schema. In contrast to JAXB, we do not generate these classes explicit as Java classes. Instead of that they can be used like built-in data types. They are defined in such a way that the generation of XML objects is done in a syntax oriented manner allowing to check the validity of all generated XML structures, i.e. XML objects, statically by the compiler. We believe that by eliminating the difference between XML objects and XML documents and by having absolutely type safe tools for generating XML objects, programming of web applications, i.e. Java Servlets, is much easier, much safer and much less error-prone.

This paper is organized as follows. In Section 2 we report on related work, i.e. we summarize the state of the art as far as generating XML and HTML is concerned. Section 3 introduces XML objects and corresponding operations for generating them. Several examples show that by using XML objects, the process of generating XML is much more precise and much clearer than by using conventional tools. Moreover, the validity of the generated XML structures is guaranteed statically by the compiler, which we present in Section 4. No test runs or special validation methods have to be used in order to check the validity at runtime. Section 5 gives some implementation details. Section 6 concludes the paper and gives an outlook on future work.

2. RELATED WORK

In this section we report on the state of the art as far as generating XML and HTML within programs is concerned.

String Operations

The most elementary way to deal with XML documents is to use the string operations which are provided by the programming language, i.e. XML documents are treated as ordinary strings without any structure. The most prominent representative of this technique is given by Java Servlets [27]. In former CGI Scripts [8] the programming language Perl [25] was used. The technique of using ordinary string operations of a programming language is rather tedious, especially when the XML document being generated is rather static, i.e. there are only few places where dynamic parts are inserted. String operations cannot guarantee the correctness of the generated XML documents according to a document type definition or an XML schema, i.e. the validity of a document must be checked dynamically by appropriate test runs.

Java Server Pages

Java Server Pages [15] are translated by a preprocessor into Java Servlets. They allow to switch between XML parts and Java parts for generating HTML or XML. This switching is done by the special markings <% and %>. An example was given in the introduction of this paper. Compared to string operations, this technique provides some progress especially when the XML document being generated is rather static. Java Server Pages share with string operations the disadvantage that the validity of the XML documents has to be checked dynamically by appropriate test runs.

XSLT

XSLT [10] is used primarily to generate HTML out of XML. By pattern matching, parts of an XML document are addressed and corresponding HTML parts are generated. This process takes into account only well-formed documents. The validity according to a DTD or an XML schema has to be checked dynamically.

JavaScript

JavaScript [14] is embedded in HTML and runs on the browser side. It allows, among others, to generate HTML parts dynamically. This can be done either on a pure string basis or on the basis of the document object model DOM [20]. There is no static checking of validity.

Document Object Model (DOM)

The main purpose of the document object model DOM [20] is to avoid generating and manipulating XML documents on a string basis by providing classes for the nodes of an XML document tree thus allowing to manipulate XML documents by object-oriented programming. In DOM, there is a clear distinction between XML objects and XML documents on a string basis. This means that the programmer has two choices: One choice is to program XML applications in a pure object-oriented manner. This requires to generate the XML structures manually by generating every node in the object explicitly by program. This is rather tedious especially in cases where large XML objects are to be generated. The second choice is to switch between XML strings and XML objects.

Because there is only one `Element` class for all different nodes in an XML tree, no validity of the XML objects can be guaranteed at compile time.

Validating Document Object Model

Validating DOM (VDOM) [12] is an extension of DOM. It provides a new distinct class for every element of a DTD. Hereby the validity of the generated structures can be guaranteed statically at compile time. Parameterized XML (P-XML) provides a mechanism to generate XML on a string-like basis allowing to statically guarantee the validity.

JAXB

JAXB [18] is Sun's proposal for the integration of XML in Java. Although there are classes for the elements of a DTD, the validity of the generated object structures cannot be guaranteed statically in all cases. Therefore, runtime validation is necessary. By so called marshalling and unmarshalling a switching between string-based XML-documents and XML object structures is necessary. Using a slightly modified picture from [17], the main aspects of JAXB can be illustrated by Figure 1.

Although JAXB is an important step in the direction of an object-based generation of XML-structures, there are still a number of disadvantages:

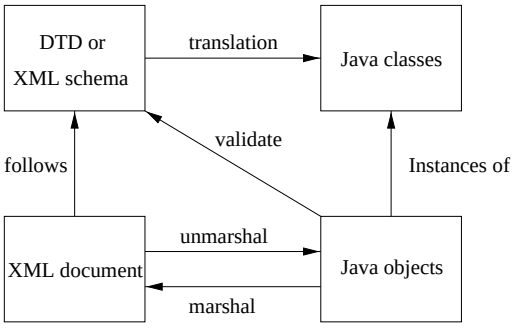

Figure 1: JAXB

- Although the idea of JAXB is to guarantee validity of object structures statically, this is accomplished only to a certain extent. For lists of elements Java's general `LIST`-class is used representing lists of arbitrary objects. This means that lists of arbitrary objects can be used in XML object structures. Therefore, `marshal`- and `validate`-methods may run into problems which can be treated only at runtime.

- The frequent case of a relatively static document with only a few dynamically generated parts can be solved only in a rather cumbersome way. Basically, there are two alternatives for this case:

  – Generate the whole object tree manually by program which basically means that the programmer has to parse the object structure manually.

  – Write a document template into a string with dummy values for the dynamically generated parts and generate the object structure by using the `unmarshal`-method. Although the template is given statically and although it could be validated statically at compile time, it has to be validated at runtime. This means runtime delay for the web application and a higher programming overhead, because appropriate exceptions have to be designed and programmed.

CASTOR

The CASTOR project [6] has many similarities to JAXB. In contrast to JAXB, in CASTOR there is only one final class `AnyNode`. Therefore, in CASTOR there is no compile time validation at all. As in JAXB, switching between string-based XML documents and `AnyNode`-objects is done by `marshal` and `unmarshal`. Because of the similarities between JAXB and CASTOR, we will not go into the details of CASTOR in this paper.

XDuce

The XDuce language [9] is a functional language developed as an XML processing language. It introduces so called regular expression types the values of which are comparable to our XML objects. Elements are created by specific constructors and the content can be accessed through pattern matching. XDuce supports type inference for patterns and variables and performs a subtyping analysis to ensure validity of regular expression type instances at compile time. The algorithm is based on regular tree automata and operates on a further internal representation for regular expression types.

BigWig

BigWig [4] is a programming language for developing interactive web services. It compiles BigWig source code into a combination of standard web technologies, like HTML, CGI, applets, and JavaScript. Typed XML document templates with gaps are introduced. In order to generate XML documents dynamically, gaps can be substituted at runtime by other templates or strings. For all templates BigWig validates all dynamically computed documents according to a given DTD. This is done by two data flow analyses constructing a graph which summarizes all possible documents. This graph is analyzed to determine validity of those documents. In comparison to our approach templates can be seen as methods returning XML objects. The arguments of the methods correspond to the gaps of the templates.

2.1 Evaluation

The main observation is that current tools for integrating XML into programming languages, except XDuce and BigWig, either use only a string-based representation of XML documents or make a strict distinction between the string representation and the object representation of an XML document. The programmer is forced to switch between both representations by marshalling and unmarshalling.

If compared to numerical computing, this would mean that the programmer of numerical applications would be forced to switch between the string representation of numbers and the corresponding abstract numbers. Of course, this is not the case in numerical applications. There is only the concept of abstract numbers, the string representation is used only to denote a number, there is no string representation of a number in the running program.

In the following section, we will show that, in analogy to numbers, it is possible to eliminate the difference between the string representation of an XML document and the XML document itself represented as an object. For the programmer, there are only XML objects. Like decimal notations of numbers and operations on numbers, there are notations for creating XML objects and for constructing new XML objects out of existing ones. As for numbers, the string representation of an XML object is needed only for communicating with the outside world. Guaranteeing the validity of XML objects according to an underlying DTD or XML schema is an important consequence of our proposal.

3. XML OBJECTS: THE XOBE PROJECT

3.1 Overview and Simple Examples

The main aim of the **XOBE**-project (**X**ML **OBJE**CTS) is to eliminate the distinction between the string representation and the object representation of XML documents. A web service generating XML or HTML conceptually works only with XML objects. XML structures in a program always denote object structures. It is important that these concepts guarantee validity statically at compile time.

In order to achieve these goals, we use a DTD or an XML schema directly as a definition of classes for XML objects. Putting in another way, this means that XML objects are instances of a DTD or of an XML schema. For every element of a DTD or of an XML schema there is a corresponding class. XML objects as instances of these classes are XML documents or a part thereof having the corresponding element as the root and being valid according to the DTD or XML schema. New XML objects can be generated by constructors which we call parameterized XML expressions. These expressions basically consist of valid XML where other valid XML

objects can be inserted in places which are allowed according to the underlying DTD or XML schema. These values are separated from the surrounding XML by braces. These XML constructors guarantee validity statically at compile time. Figure 2 shows the relationships between DTD or XML schema and XML objects in XOBE.

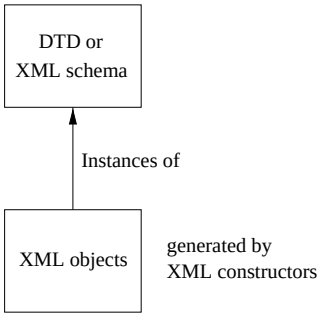

Figure 2: XOBE

We will explain our concepts by examples. The first examples deal with HTML and assume an XML schema for XHTML as given in [22].[1]

The following Java method generates an HTML object containing a counter being taken from an `int`-value. `int`-values are allowed in places where `String`-values are expected according to the underlying definition of XHTML. The `int`-value is converted automatically to its decimal notation as a `String`-value.

```
public HTML countHTML(int count){
   HTML h;

   h = <HTML>
         <HEAD><TITLE> Test Server Page </TITLE>
         </HEAD>
         <BODY>
            Hello World: You are number
               <B> {count} </B>
            to use this servlet.
         </BODY>
       </HTML>;

   return h;
} // countHTML
```

It should be obvious that `countHTML` is guaranteed statically to generate valid HTML.

The following method shows how to use different HTML classes. It is a XOBE solution of the Java Server Page given in Listing 1.

```
public HTML timesofday(){
    HTML html;
    LI listelement;

    if (timeOfDay() == "AM")
        listelement = <LI> Good Morning </LI>;
    else
        listelement = <LI> Good Afternoon </LI>;

    html = <HTML>
             <HEAD>
                <TITLE> Times Of Day </TITLE>
             </HEAD>
             <BODY>
               <UL> {listelement} </UL>
             </BODY>
           </HTML>;

    return html;
} // timesofday
```

Obviously, only valid HTML is generated.

For example, the following piece of program is **syntactically wrong** because a `TITLE` object is not allowed within an `UL`-object according to the underlying definition of XHTML [22].

```
...
HTML html;
TITLE title;

title = <TITLE> Wrong Program </TITLE>

html = <HTML>
         <HEAD>
            <TITLE> Times Of Day </TITLE>
         </HEAD>
         <BODY>
            <UL>{title}</UL>
         </BODY>
       </HTML>;
...
```

In JAXB, for example, in general errors of this kind can be found only at runtime. Moreover, JAXB requires to explicitly convert a template given in a string to objects at runtime (`unmarshal`-method).

A XOBE program generates XML only by XML constructors, i.e. there is no string representation during generation. String generation is necessary only when the document is communicated to the outside world, for example as the result of a Java Servlet. Only for this purpose a method `toString` is provided for XML objects.

For generating lists by using a loop construct or recursion, the classes defined so far are not sufficient. We need classes for defining variables acting as a container for a list. A class for this purpose for lists the elements of which are `e`-elements is called `e*` group.[2] With `<>` we denote the empty list. The concatenation operator `+` can be used with the obvious semantics. Moreover, there is a `getLength`-method for these classes. Of course, an `e*` object can be inserted only in places where a group of the elements is allowed according to the underlying DTD or XML schema.

The following example shows how to generate lists by using Java's loop constructs. We assume a nonempty array of strings. We want to generate list elements out of the array elements.

```
public HTML dynamic_page(String[] values){
    HTML html;

    LI* valuelist;

    valuelist = <>;

    for (int i=0; i < values.length; i++) {
       valuelist = valuelist +
                      <LI> {values[i]} </LI> ;
    } // for

    html = <HTML>
             <HEAD>
                <TITLE> List of Strings </TITLE>
             </HEAD>
             <BODY>
```

[1] In contrast to [22], we use upper case letters for HTML tags.

[2] It should be noted that an extra `e*` group is not necessary if there is an explicit type name in the underlying XML schema for the group.

```
                <UL>{valuelist}</UL>
              </BODY>
            </HTML>;

    return html;
} // dynamic_page
```

In this example, JAXB would require to use the Java `LIST`-class for defining variable `valuelist`. Because `LIST`-variables allow to store list elements belonging to an arbitrary class, correct HTML is not guaranteed statically.

Although XOBE can ensure most predicates of the property validity statically, runtime checks are necessary in some exceptions. Similar to an array of constant length, where the index has to be in the declared range, the number of occurrences of specific element has to be between the values set by the attributes `minOccurs` and `maxOccurs` in the defining XML schema. Additional runtime checks are necessary for identity constraints, restricted string types and facets on numeric types.

The impact of our approach on the host language Java is an XML consistent extension of the type system. Because of the outstanding role of XML in the web application and web services programming world, and may be the whole software development world in the future, this seems to be a consequent step. We believe that the tradeoffs between the extension of an existing programming language on one hand and the approach of defining a new programming language around XML on the other are minimal. Instead the benefits of using already developed code are significant.

3.2 An Illustrative Example Based on XML Schema

This section introduces a scenario of a web application using our non-standard language extension. The language we use is an extension of Java that we described in Section 3.1.

The example presented in this section is based on a web application implementing a shopping portal for antiquarian books and records. The user of this application can search different book and record catalogs, can buy books and records or can contact the antiquarian book seller offering a particular item for further information. Many different antiquarian book and record sellers support the online antiquarian book seller application.

They send their current offerings to the antiquary web site in an XML data format. This format is called Antiquary Offerings Markup Language, or AOML, and is defined as follows.

```
<schema>
  <element name="aoml" type="aoml"/>

  <complexType name="aoml">
    <sequence>
      <element name="antiquary" type="antiquary"
          />
      <element name="offerings" type="offerings"
          />
    </sequence>
    <attribute name="date" type="date"/>
  </complexType>

  <complexType name="antiquary">
    <sequence>
      <element name="name" type="string"/>
      <element name="address" type="string"/>
      <element name="email" type="string"/>
    </sequence>
  </complexType>

  <complexType name="offerings">
    <group ref="item" maxOccurs="unbounded"/>
  </complexType>

  <group name="item">
    <choice>
      <element name="book" type="book"/>
      <element name="record" type="record"/>
    </choice>
  </group>

  <complexType name="book">
    <sequence>
      <element name="author" type="string"
          minOccurs="0"/>
      <element name="title" type="string"/>
      <element name="condition" type="string"/>
      <element name="price" type="price"/>
    </sequence>
    <attribute name="catalog" type="string"/>
  </complexType>

  <complexType name="record">
    <sequence>
      <element name="artist" type="string"/>
      <element name="title" type="string"/>
      <element name="condition" type="string"/>
      <element name="price" type="price"/>
    </sequence>
    <attribute name="catalog" type="string"/>
  </complexType>

  <complexType name="price">
    <simpleContent>
      <extension base="decimal">
        <attribute name="currency" type="string
            "/>
      </extension>
    </simpleContent>
  </complexType>

  ...
</schema>
```

The root element of AOML is the `aoml` element. The type of the `aoml` element is a sequence of one `antiquary` element and one `offerings` element as well as an attribute `date`. The `antiquary` element consists of information about the offering antiquary kept in the elements `name`, `address`, and `email`. In the `offerings` element the offered books and records are specified. Therefore the type of element `offerings` is an unbounded choice of `book` and `record` elements. Books are described by a sequence of elements qualifying the author, which is optional, title, condition, and price. Analogously the records are characterized by artist, title, condition, and price.

The following document is an example of a book list sent from one book seller to the web application.

```
<aoml date="2/20/2002">
  <antiquary>
    <name>St. Juergen Antiquary</name>
    <address>Ratzeburger Allee 40, 23562 Luebeck
        </address>
    <email>st.juergenantiquariat@t-online.de</
        email>
  </antiquary>
  <offerings>
    <book catalog="literature">
      <author>Thomas Mann</author>
      <title>Lotte in Weimar</title>
      <condition>binder blotched, pale back</
          condition>
      <price currency="EUR">8.00</price>
    </book>
    <book catalog="literature">
      <author>Thomas Mann</author>
```

```
        <title>Buddenbrooks</title>
        <condition>binder faded, owner qualifier
            at prefix</condition>
        <price currency="EUR">25.00</price>
      </book>
    </offerings>
</aoml>
```

The antiquarian book seller `St. Juergen Antiquary` offers two books in the catalog `literature`.

A second XML format called Antiquarian Catalog Markup Language, or ACML, is used to present the current offerings to the customer. In ACML the offerings of all sellers are joined and categorized by different catalogs. In the following the definition of ACML is given.

```
<schema>
    ...

    <element name="acml" type="acml"/>

    <complexType name="acml">
      <group ref="catalogs"/>
    </complexType>

    <group name="catalogs">
      <sequence maxOccurs="unbounded">
        <group ref="cat_item"/>
      </sequence>
    </group>

    <group name="cat_item">
      <sequence>
        <element name="catalog" type="catalog"/>
      </sequence>
    </group>

    <complexType name="catalog">
      <sequence>
        <element name="title" type="string"/>
        <group name="entries"/>
      </sequence>
    </complexType>

    <group name="entries">
      <sequence maxOccurs="unbounded">
        <group ref="cat_entry"/>
      </sequence>
    </group>

    <group name="cat_entry">
      <sequence>
        <element name="entry" type="entry"/>
      </sequence>
    </group>

    <complexType name="entry">
      <sequence>
        <choice>
          <element ref="book"/>
          <element ref="record"/>
        </choice>
        <element ref="antiquary"/>
      </sequence>
    </complexType>
</schema>
```

The root element of ACML is the `acml` element. The type of the `acml` element is an unbounded sequence of `catalog` elements. The `catalog` element consists of information about the catalog title kept in the `title` element and the entries of the catalog, contained in an unbounded sequence of `entry` elements. In each `entry` element the offered book or record is specified together with its offering antiquary.

The following document is an example of the interface to the user of the web application where all the books and records are categorized by catalogs. In this instance only the books from the St. Juergen Antiquary are considered.

```
<acml date="2/20/2002">
  <catalogs>
    <catalog>
      <title>literature</title>
      <entry>
        <book>
          <author>Thomas Mann</author>
          <title>Lotte in Weimar</title>
          <condition>binder blotched, pale
              back</condition>
          <price currency="EUR">8.00</price>
        </book>
        <antiquary>
          <name>St. Juergen Antiquary</name>
          <address>Ratzeburger Allee
              40, 23562 Luebeck</address>
          <email>st.juergenantiquariat@t-
              online.de</email>
        </antiquary>
      </entry>
      <entry>
        <book>
          <author>Thomas Mann</author>
          <title>Buddenbrooks</title>
          <condition>binder faded, owner
              qualifier at prefix</condition>
          <price currency="EUR">25.00</price>
        </book>
        <antiquary>
          <name>St. Juergen Antiquary</name>
          <address>Ratzeburger Allee
              40, 23562 Luebeck</address>
          <email>st.juergenantiquariat@t-
              online.de</email>
        </antiquary>
      </entry>
    </catalog>
  </catalogs>
</acml>
```

As shown the books are now assigned to their book catalog and each entry includes the corresponding antiquarian book seller.

In the following we present the algorithm which transforms the AOML documents transmitted from the various connected antiquarian book sellers to the web application into the ACML document. This document is the format which is presented to the user of the application to search through the offered items. We assume that the AOML documents exist already as `aoml` objects and the system processes the resulting `acml` object subsequently to our algorithm.

Listing 3 shows the implementation of the method `asCatalogs` as a XOBE program transforming an array of `aoml` objects into one `acml` object.

```
1  acml asCatalogs(aoml[] input) {
2     int i;
3     String title;
4     String[] titles;
5     CatMap m;
6     aoml a;
7     catalogs cats;

9     // insert all offerings of one antiquary into
          the map m
10    m = new CatMap();
11    for (i = 0; i < input.length; i = i + 1) {
12       m = addToMap(m, input[i]);
13    } // for

15    // generate the catalog out of the map m
16    cats = <>;
```

```
17      titles = m.titles();
18      for (i = 0; i < titles.length; i = i + 1) {
19         title = titles[i];
20         cats = cats + <catalog>
21                         <title>{title}</title>
22                         {m.get(title)}
23                      </catalog>;
24      } // for

26      return <acml>{cats}</acml>
27   } // asCatalogs
```

Listing 3: Method `asCatalogs`

First the method iterates over the array of `aoml` objects received as input parameter (`input`). For each `aoml` object the method `addToMap` is called, which adds the offerings of the current object to the map `m` of class `CatMap`. The class `CatMap` is defined similar to the Java class `Map` but uses the specialized classes `String` and `entries` instead of the general `Object` for the operations `get`, `put`, `titles`, and `containsTitle`. Using `CatMap` the map `m` maps the offered books or records to catalog titles, which are the keys of the map. The implementation of `addToMap` can be seen in Listing 4. Second the method generates an object `catalog` for every catalog registered in the map `m`. All `catalog` objects are added to the `catalogs` group `cats`. At the end the method returns the complete document in `acml` format.

In the method `asCatalogs` we use XOBE program constructs for creating the group of catalogs and the `acml` object. First an empty `catalogs` group is assigned to variable `cats` (16) and second `cats` is extended by a `catalog` object for each map entry (20-23). Therefore XML constructors and the operation `+` is used. At the end of the method yet another XML constructor is called returning the final `aoml` object.

The method `addToMap` in Listing 4 adds the offerings of an `aoml` object to a given map.

```
28   CatMap addToMap(CatMap m, aoml input) {
29      String ctlg;
30      antiquary an;
31      offerings of;
32      item itm;

34      an = input/antiquary;
35      of = input/offerings;

37      for (int i = 0; i < of/*.getLength(); i++) {
38         // itm points to the current item
39         itm = of/*[i];

41         // get value of attribute catalog
42         if (itm@catalog == null)
43            ctlg = "misc";
44         else
45            ctlg = itm@catalog;

47         // remove attribute catalog
48         itm@catalog = null;

50         // add entry to the catalog
51         if (m.containsTitle(ctlg))
52             m.put(ctlg , (m.get(ctlg) +
53                         <entry>{itm}{an}</entry>);
54         else
55             m.put(ctlg , <entry>{itm}{an}</entry>);
56      } // for

58      return m;
59   } // addToMap
```

Listing 4: Method `addToMap`

The method iterates through the offerings of the `aoml` object `input` given as parameter. For each offered item the method tries to determine the corresponding catalog name given by the attribute `catalog`. If the attribute is present the item is registered with the value of the attribute as catalog name into the map `m`. If the attribute `catalog` is absent the item is sorted into catalog `misc`. The attribute `catalog` is removed before the item is added to the map values. The items of one catalog are stored in an entries group.

In the example we use XPath expressions [21] to select the contents of the `aoml` element (34,35) and the content of one offered item (39,42,45). For the termination of the for-loop the number of items is needed, which we receive by calling the `getLength`-method (37). Further the attribute `catalog` is removed (48). In the if-condition the values of the map are extended by `entry` objects (52-53,55) using operation `+` and XML constructors.

4. THE XOBE TYPE SYSTEM

As seen in the last section XOBE allows XML syntax in expressions, assignments and method parameters. Consider the following assignment as example where an expression in AOML syntax is assigned to variable `page`.

```
aoml page;
offerings offers;

 :
page = <aoml>
          <antiquary>
             <name>St. Juergen Antiquary</name>
             <address>Ratzeburger Allee 40, 23562
                  Luebeck</address>
             <email>st.juergenantiquariat@t-online.
                  de</email>
          </antiquary>
          {offers}
       </aoml>;
```

During compilation the XOBE system verifies the correctness of the assignment in two steps. First it determines the types of the right and left hand side using type inference. Secondly, the subtype relationship of the inferred types is checked by a subtyping algorithm.

In XOBE we formalize and represent types as regular hedge expressions representing regular hedge languages [5]. Consequently an XML schema is formalized and represented internally by a regular hedge grammar.

Definition 1 (regular hedge grammar)
A regular hedge grammar is defined by $G = (T, N, s, P)$ with a set $T = B \cup E$ of terminal symbols, consisting of simple type names B and a set E of element names (Tags), a set N of nonterminal symbols (names of groups and complex types), a start expression s and a set P of rules or productions of the form $n \rightarrow r$ with $n \in N$ and r is a regular hedge expression over $T \cup N$.[3] □

We define the regular hedge expression, referred in short as regular expression, similar to the notation used in [23].

Definition 2 (regular hedge expression)
Given a set of terminal symbols $T = B \cup E$ and a set N of nonterminal symbols, the set Reg of regular hedge expressions is defined

[3] We restrict r to be recursive in tail position only. This ensures regularity.

recursively as follows:

$$\begin{aligned}
\emptyset &\in Reg && \text{the empty set,}\\
\epsilon &\in Reg && \text{the empty hedge,}\\
b &\in Reg && \text{the simple types,}\\
n &\in Reg && \text{the complex types,}\\
e[r] &\in Reg && \text{the elements,}\\
r|s &\in Reg && \text{the regular union operation,}\\
r,s &\in Reg && \text{the concatenation operation, and}\\
r* &\in Reg && \text{Kleene star operation.}
\end{aligned}$$

for all $b \in B$, $n \in N$, $e \in E$, $r, s \in Reg$. □

As an example we formalize the XML schema from the last section as regular hedge grammar as $G = (B \cup E, N, aoml[aoml], P)$ with:

$$\begin{aligned}
B &= \{string\},\\
E &= \{aoml, antiquary, offerings, name, address,\\
&\quad email, \dots\},\\
N &= \{aoml, antiquary, offerings, item, \dots\}, \text{ and}\\
p &= \{ \quad aoml \rightarrow (antiquary[antiquary],\\
&\qquad\qquad offerings[offerings]);\\
&\quad antiquary \rightarrow (name[string], address[string],\\
&\qquad\qquad email[string]); \dots\}.^{4}
\end{aligned}$$

As in XML Schema we do not demand that the set of element names E and the set of complex types N have to be disjunct.

As mentioned above XOBE infers at compile time both types of the right and left hand sides of the assignment. Because all variables have to be declared, the type inference of variables is simple. In our example variable `page` is declared of type $aoml$ and variable `offers` of type $offerings$. Based on the variable types, the type of the whole XML constructor on the right hand side can be inferred. In the example above it is $aoml[antiquary[name[string], address[string], email[string]], offerings]$.

After inferring the types of the left and right hand sides, the XOBE type system checks if the type of the right hand side is a subtype of the type of the left hand side. For this example XOBE has to check the so-called *regular inequality*

$$aoml[antiquary[name[string], address[string], email[string]], offerings] \leq aoml$$

where $\leq$ stands for the subtype relationship. Note, that the name $aoml$ on the right hand side stands for the complex type $aoml$. The name $aoml$ on the left hand side is an element name.

Checking the subtype relationship between two regular hedge expressions is the main task in proving type correctness of XOBE programs. For this we adopt the Antimirov algorithm [2] for checking inequalities of regular expressions and extend it to the hedge grammar case. The idea behind Antimirov's algorithm is that for every invalid regular inequality there exits at least one reduced inequality which is *trivially inconsistent*. An inequality is trivially inconsistent if the empty word is in the language represented by the regular expression on the left hand side but not in the language represented by the right hand side regular expression.

The algorithm operates as follows: It takes the regular inequality to prove as argument and retrieves the leading simple type names, complex type names and element names from the left hand side regular expression using operation `leadingNames`. For each determined name the algorithm tries to reduce both sides of the inequality by this name. The resulting reduced inequalities are simpler than the starting inequality in the majority of cases and can be checked by a recursive application of the algorithm. The algorithm tracks already treated inequalities in a set of assumptions, which is empty in the beginning, as in standard algorithm for subtyping recursive types. This ensures termination if we encounter the same inequality later on. To avoid invalid proves of subtyping, we have to add an inequality to the set of assumptions after calculating its reduced inequalities.

There are two different results of our recursive algorithm. First the algorithm responds false if the inequality in question is trivially inconsistent. To verify this property the predicate `isNullable` checks the empty word inclusion. Secondly, the algorithm terminates with true when it processes an inequality which is already in the set of assumptions. This means that our algorithm cannot produce any new inequality in this branch of recursion.

Definition 3 (subtyping algorithm)
Given a set A of assumption the inequality to prove $r \leq s$ is defined by the following pseudo code:

```
bool prove(r ≤ s, A) {
  if ((isNullable(r) && ~isNullable(s))
      || s == ∅)
    return false;
  elsif ((r ≤ s) ∈ A)
    return true;
  else {
    ok := true;
    ns := leadingNames(r);
    forall (n ∈ ns)
       pd := pd ∪ partialDerivatives(n,r
             ≤ s)
    A := A ∪ {r ≤ s};
    forall ((r1 ≤ s1) ∨ (r2 ≤ s2) ∈ pd)
      ok := ok && (prove(r1 ≤ s1,A) ||
                   prove(r2 ≤ s2,A));
    return ok;
  } // else
} // prove
```

Listing 5: Subtyping Algorithm

□

Antimirov introduces so-called *partial derivatives of regular expressions* to express reduced regular expression. A partial derivative reduces a regular expression by a given type name or element name. In the hedge grammar setting we modify partial derivatives concerning type names and element names. For example, if we have the regular expression $(name[string], address[string])$ and calculate its partial derivative with respect to the given element name $name$, we receive the result $\{(string; address[string])\}$. The result is a set of type pairs in general, because we can get multiple derivatives for a given regular expression. The first type of the

[4] Because the comma (,) is used as concatenation operation in regular expression, we use the semicolon (;) as separator in sets where regular expressions appear as elements.

pair is the type of the content of element $address$. The second is the regular expression reduced by element $name$.

Additionally Antimirov introduces so-called *partial derivatives of regular inequalities* to express reduced inequalities. In the hedge grammar case these become more complicated. Because the partial derivatives of regular expressions are pairs we have to perform the set-theoretic observation of Hosoya, Vouillon and Pierce [9] for a recursive application of our `prove` procedure. This transformation simplifies the set of all partial derivatives of regular inequalities. We receive a set the elements of which have the form of two inequalities conjuncted with the boolean operation or.

In the remaining section we apply our subtyping algorithm to a small example. Consider the following hedge grammar as example $G = (B \cup E, N, (aq)*, P)$ with:

$$\begin{aligned} B &= \{str\}, \\ E &= \{n, a\}, \\ N &= \{aq\}, \text{ and} \\ P &= \{aq \to n[str], a[str]\}. \end{aligned}$$

For a concise description let $r \equiv (n[str], (a[str], n[str])*)$ and $s \equiv (aq*, n[str])$. We want to check the regular inequality $r \leq s$, for which we start the subtyping algorithm with an empty set of assumptions $A = \{\}$.

In the first recursion the algorithm calculates the sets of leading names $ns = \{n\}$. Additionally the set A of assumptions is enlarged by the inequality to check $A := A \cup \{r \leq s\}$.

During the calculation of partial derivatives of the inequality $r \leq s$ the partial derivatives $\{(str; (a[str], n[str])*)\}$ of the regular expression r and $\{(str; a[str], s); (str; \epsilon)\}$ of the regular expression s are determined. More precisely, this means

$$(str; (a[str], n[str])*) \leq (str; a[str], s) | (str; \epsilon)\ .$$

Because our algorithm is not suited for proving Cartesian product subtypes we apply the following set-theoretic transformation. For brevity, let the inequality above be

$$(a; b) \leq (c_1; d_1) | (c_2; d_2)\ .$$

At first we can rewrite this inequality to

$$(a; b) \leq (c_1; \tau) \cap (\tau; d_1) | (c_2; \tau) \cap (\tau; d_2)$$

because in general, $a \times b$ is equal to $(a \times \tau) \cap (\tau \times b)$ with τ being the set of all ground types. After this we apply distributivity of intersection over unions and turn the disjunctive form into the conjunctive form. This yields

$$\begin{aligned} (a; b) \leq\ & ((c_1; \tau) | (c_2; \tau)) \cap \\ & ((c_1; \tau) | (\tau; d_2)) \cap \\ & ((\tau; d_1) | (c_2; \tau)) \cap \\ & ((\tau; d_1) | (\tau; d_2)) \end{aligned}$$

where we can separate the c_i and d_i types in each disjunction:

$$\begin{aligned} (a; b) \leq\ & ((c_1 | c_2; \tau) | (\tau; \emptyset)) \cap \\ & ((c_1; \tau) | (\tau; d_2)) \cap \\ & ((c_2; \tau) | (\tau; d_1)) \cap \\ & ((\tau; \emptyset) | (\tau; d_1 | d_2)) \end{aligned}$$

Because τ is one argument in each clause this inequality is in turn equivalent to:

$$\begin{aligned} & (a \leq c_1 | c_2 \vee \quad b \leq \emptyset) \wedge \\ & (a \leq c_1 \quad \vee \quad b \leq d_2) \wedge \\ & (a \leq c_2 \quad \vee \quad b \leq d_1) \wedge \\ & (a \leq \emptyset \quad \vee \quad b \leq d_1 | d_2) \end{aligned}$$

According to this transformation the algorithm produces the following 8 inequalities, which are pairwise connected with the boolean operation or. This means that only one inequality of the pairs has to evaluate to true.

$$\begin{aligned} pd = \{ & (str \leq str | str \vee && (1) \\ & \quad (a[str], n[str])* \leq \emptyset); && (2) \\ & (str \leq str \vee && (3) \\ & \quad (a[str], n[str])* \leq \epsilon); && (4) \\ & (str \leq str \vee && (5) \\ & \quad (a[str], n[str])* \leq a[str], s); && (6) \\ & (str \leq \emptyset \vee && (7) \\ & \quad (a[str], n[str])* \leq (a[str], s) | \epsilon)\} && (8) \end{aligned}$$

The inequalities 1, 5 and 3 evaluate trivially to true, while inequalities 2, 6, 4 do not hold. Because inequality 7 is false, inequality 8 has to be checked in another recursion of our algorithm.

Let $r' \equiv (a[str], n[str])*$ and $s' \equiv (a[str], s) | \epsilon)$ for further description. For proving inequality 8 the algorithm calculates the sets of leading names $ns = \{a\}$. Again the set of assumptions is enlarged $A := A \cup \{r' \leq s'\}$. The following partial derivatives are generated during the algorithm:

$$\begin{aligned} pd = \{ & (str \leq str \vee && (9) \\ & \quad r \leq \emptyset); && (10) \\ & (str \leq \emptyset \vee && (11) \\ & \quad r \leq s)\} && (12) \end{aligned}$$

It is easy to see, that inequality 9 holds and inequalities 10 and 11 evaluate to false. The last inequality 12 is true, because it is already in our set of assumptions $r \leq s \in A$. This leads to the result, that inequality $r \leq s$ is accepted.

Because the algorithm did not derive any inconsistent partial derivative we can accept $r \leq s$ as correct.

5. IMPLEMENTATION ISSUES

The previous sections presented the representation for the code of XOBE programs. We now describe our XOBE architecture [13], which is shown in Figure 3. Although it is possible to integrate the functionality of XOBE into the Java compiler, we have chosen to implement XOBE as a Java preprocessor. The XOBE preprocessor consists of the following three components:

1. The XOBE parser,
2. the type checking analysis and
3. the transformation to standard Java code.

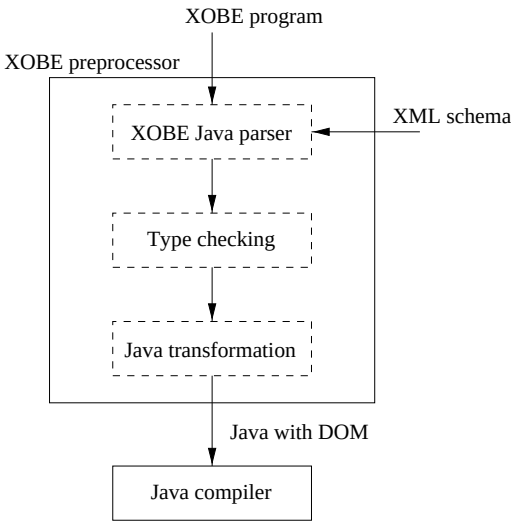

Figure 3: XOBE Java Architecture

The XOBE parser reads the XOBE program and converts the XML portions of the program to an internal representation. The parser includes in addition to a standard Java parser an XML Schema parser and a slightly modified XML Parser. The schema parser is necessary to scan the XML schemas imported by the XOBE program. The XML Parser is needed to recognize the XML constructors distributed over the program. Because the XML constructors can include XML variables we have to modify the standard XML Parser. In our implementation we utilize the Java compiler compiler JavaCC [26] to generate the XOBE parser. Additionally we use the XML parser xerces [16] to recognize the used XML schemas. The internal representation of the processed XOBE program is done with the Java tree builder JTB [19].

In the type analysis phase the preprocessor determines whether the parsed program is well typed or not. Well-typed in XOBE means that the processed XML objects are valid according to the declared XML schemas. At first the type analysis phase validates the imported schemas. Afterwards, the type check of Java expressions using XML objects, like assignments or method calls, is performed according to the description of the previous section. The type inference of XML constructors and XML variables is followed by subtyping proves to verify the expressions. Because the type system of standard XML is strict and can be formalized by restricted regular hedge expressions, we can use the regular hedge grammar based algorithm deciding the equivalence of regular expressions for that purpose. XML Schema weakens the strictness by introducing type extension and type restriction, which requires a more sophisticated type inference strategy. The detailed description of this extended algorithm will be introduced in thesis [11].

The last task the preprocessor performs is the transformation of the XOBE program into Java source code, which is accepted by the standard Java compiler. For this resulting Java code several implementation alternatives exists, depending on the XML representation. We chose the standard representation of the Document Object Model, or DOM [20], recommended by the W3Consortium. The transformation replaces the XML constructors and XPath expressions of the XOBE program with suitable DOM code. The exact transformation rules will be presented in [11]. Please note that even though DOM is an untyped XML implementation not guaranteeing validity statically, the transformed XML objects in the XOBE program are valid. This holds because our type-checking algorithm guarantees this property. In our implementation the transformation is performed on the internal JTB representation, where we replace the subtrees representing XOBE constructs by newly created subtrees, which represent the suitable DOM code.

Using DOM is a straightforward implementation, but other implementations are possible as well. Especially we think that an implementation, which enables a more structured access to XML elements, would be useful. The idea of this structured access is that the user can select the content of an element by structure defined in the schema beside the element centric access of DOM. The structure of the content can be a nested sequence, choice or all groups. Even DOM permits this structured selection too, if the appropriate schema is available, expensive additional calculation effort is necessary. This calculation is required because the structure of an element isn't stored in the DOM implementation. If the implementation considers the structure of the element content, the content can be accessed in constant time. This accelerates the selection by magnitude, because the DOM selection of one child element is linear to the number of children.

Because XOBE is a statically typed system, XOBE programs have no runtime overhead when compared to regular Java programs. The included XML schemas are used only for compile time type checking and are not preserved at runtime. However, the extra type information in an XOBE program can be used to enable program optimizations. For example, the attribute and tag names of elements can be stored in the corresponding XML object classes once instead of storing duplicates for every XML object.

6. CONCLUDING REMARKS

This paper proposed to eliminate the distinction between XML documents in string form and in object form. Web applications generating XML or HTML should have to deal only with XML objects. A document type definition or an XML schema is used directly for defining new classes for XML objects. XML constructors allow to generate new XML objects either from scratch or by inserting existing XML objects in places, which are allowed according to the underlying DTD or XML schema. This mechanism guarantees statically by the compiler that XML objects always contain valid XML documents. There is no need for special runtime checking or for extra test runs to check the validity of the generated XML documents. These tools require to generate an XML document in string form only when the document is communicated to the outside world for example as the result of a Java Servlet.

In the future, a lot of research has to be done to integrate XPath and XQuery with XML objects smoothly. In this paper, only the basics concerning XPath and XML objects could be given and solved. A challenging question will be what kind of restrictions is necessary in order to be able to guarantee the type of the result of a XQuery statement statically and uniquely. Moreover, optimization issues have to be looked at. For example, the compiler can prepare the generation of XML objects out of existing ones by XML constructors such that at runtime only some pointers have to be set.

7. REFERENCES

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web, From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, San Francisco, California, 2000.

[2] V. Antimirov. Rewriting regular inequalities. In Reichel, editor, *Fundamentals of Computation Theory*, volume 965 of

LNCS, pages 116–125. Springer Verlag Heidelberg, 1994.

[3] K. Arnold and J. Gosling. *The Java Programming Language*. The Java series. Addison Wesley Longman Limited, second edition, 1998.

[4] C. Brabrand, A. Moller, and M. I. Schwartzbach. Static validation of dynamically generated html. In *Proceedings of Workshop on Program Analysis for Software Tools and Engineering (PASTE 2001), June 18-19, Snowbird, Utah, USA*. ACM, 2001.

[5] A. Brüggemann-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over unranked alphabets: Version 1. Technical Report HKUST-TCSC-2001-05, Hong Kong University of Science & Technology, April 3 2001. Theoretical Computer Science Center.

[6] ExoLab Group. Castor. ExoLab Group, `http://castor.exolab.org/`, 11 December 2001.

[7] D. K. Fields and M. A. Kolb. *Web Development with Java Server Pages, A practical guide for designing and building dynamic web services*. Manning Publications Co., 32 Lafayette Place, Greenwich, CT 06830, 2000.

[8] M. Gaither. *Foundations of WWW-Programming with HTML and CGI*. IDG-Books Worldwide Inc., Foster City, California, USA, 1995.

[9] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for xml. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada*, volume 35(9) of *SIGPLAN Notices*, pages 11–22. ACM, September 18-21 2000. ISBN 1-58113-202-6.

[10] M. Kay. *XSLT Programmer's Reference*. Wrox Press Ltd., Burmingham, 2000.

[11] M. Kempa. *Schema-abhängige Programmierung von XML-basierten Web-Anwendungen*. PhD thesis, Institut für Informationssysteme, Universität zu Lübeck, 2003. to appear, in german.

[12] M. Kempa and V. Linnemann. V-DOM and P-XML – Towards A Valid Programming Of XML-based Applications. In A. B. Chaudhri and A. Rashid, editors, *OOPSLA '01 Workshop on Objects, XML and Databases, Tamba Bay, Florida, USA*, October 2001.

[13] J. Kramer. Erzeugung garantiert gültiger Server-Seiten für Dokumente der Extensible Markup Language XML. Master's thesis, Institut für Informationssysteme, Universität zu Lübeck, 2002. in german.

[14] Netscape Communications Corporation. JavaScript 1.1 Language Specification. `http://www.netscape.com/eng/javascript/index.html`, 1997.

[15] E. Pelegrí-Llopart and L. Cable. Java Server Pages Specification, Version 1.1. Java Software, Sun Microsystems, `http://java.sun.com/products/jsp/download.html`, 30. November 1999.

[16] T. A. X. Project. Xerces Java Parser. `http://xml.apache.org/xerces-j/index.html`, 15. November 2001. Version 1.4.4.

[17] A. Renner. Xml data and object databases: The perfect couple? In *Proceedings IEEE International Conference on Data Engineering*, pages 143–148, Heidelberg, April 2001.

[18] Sun Microsystems, Inc. The Java Architecture for XML Binding, User Guide. `http://www.sun.com`, May 2001.

[19] K. Tao, W. Wang, and D. J. Palsberg. Java Tree Builder JTB. `http://www.cs.purdue.edu/jtb/`, 15. May 2000. Version 1.2.2.

[20] W3Consortium. Document Object Model (DOM) Level 1 Specification, Version 1.0. Recommendation, `http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/`, 1. October 1998.

[21] W3Consortium. XML Path Language (XPath), Version 1.0. Recommendation, `http://www.w3.org/TR/xpath`, 16. November 1999.

[22] W3Consortium. XHTML 1.0: The Extensible HyperText Markup Language, A Reformulation of HTML 4.0 in XML 1.0. Recommendation, `http://www.w3.org/TR/2000/REC-xhtml1-20000126/`, 26. January 2000.

[23] W3Consortium. XML Schema: Formal Description. Working Draft, `http://www.w3.org/TR/2001/WD-xmlschema-formal-20010925/`, 25. September 2001.

[24] W3Consortium. XML Schema Part 0: Primer. Recommendation, `http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/`, 2. May 2001.

[25] L. Wall and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., Sebastipol, California, 1992.

[26] WebGain. Java Compiler Compiler (JavaCC) - The Java Parser Generator. `http://www.webgain.com/products/java_cc/`, 2002. Version 2.1.

[27] A. R. Williamson. *Java Servlets by Example*. Manning Publications Co., Greenwich, 1999.