Describing XML Wrappers for Information Integration

M. Ek, H. Hakkarainen, P. Kilpeläinen,
E. Kuikka, T. Penttinen

Report A/2001/2

ISBN 951-781-619-7
ISSN 0787-6416

UNIVERSITY OF KUOPIO

Department of Computer Science
and Applied Mathematics

P.O.Box 1627, FIN-70211 Kuopio, FINLAND

Describing XML Wrappers for Information Integration

Merja Ek, Heli Hakkarainen, Pekka Kilpeläinen,
Eila Kuikka and Tommi Penttinen
University of Kuopio
Department of Computer Science and Applied Mathematics

Abstract

XML-based data formats are actively being developed for standard-based exchange of data between heterogeneous co-operating systems. For this end we need transformation programs called wrappers, which are able to expose system-specific data using the chosen XML-based representation. Wrappers can be written using practically any general purpose programming language but ad hoc solutions are tedious both to develop and to maintain. To alleviate the problem of implementing XML wrappers we introduce a simple yet powerful wrapper specification language called XW. Using XW one can describe the structure of serialized input data through a simple declarative specification, which acts as a template for the automatic generation of a corresponding structured XML representation. We introduce the language through example specifications of real-life wrappers. We also sketch its implementation principles, emphasizing the use of standard XML techniques.

1 Introduction

XML-based data formats are actively being developed for standard-based exchange of data between co-operating information systems. For this end we need transformation programs called wrappers, which are able to expose system-specific data using the chosen XML-based representation.

We are currently running a project supported by TEKES (the National Technology Agency of Finland) and seven supporting companies and organizations to study and to develop the methodology of defining, developing and utilizing XML-based interfaces. As an initial phase of the project we have studied sample data of our project partners, including billing invoices, medical laboratory requests and responses, protocol messages and remote procedure calls. We have defined XML representations for these data and implemented ad hoc wrappers for translating each of them to the corresponding XML format. However, internal and external data formats tend to evolve, and maintaining such ad hoc transformation programs rapidly becomes a burden. Therefore a higher-level solution to defining and implementing XML-wrappers is needed.

In this paper, we introduce a declarative and concise wrapper specification language called XW (XML Wrapper), which will support automatic generation of wrappers. XW provides a loose-coupling framework for transforming serialized data into a hierarchical XML representation. That is, the language and its implementation will support the translation of data files that are external to the information system that produced the data. A tightly coupled wrapper that retrieves and wraps data, for example from a database system using its query language, may be more efficient in some situations, but we believe that loosely coupled wrappers are often both easier to use and sufficient in practise.

The idea of wrappers as a means of extracting information from databases was introduced by Wiederhold [Wie92]. Most wrapper applications discussed in the literature have been built for extracting data from Web sources [HGM97] [AsK97] [Ade98] [LHB99][LPH00] [SaA99]. The wrapper specification for HTML pages is either a set of rules or a query in some query language designed for wrapping. The wrappers are built semi-automatically or automatically from these specifications using a special tool or a programming language. For heterogenous data sources systems like TSIMMIS [GMP95] and MIX [BGL99] have been developed with the approach of using a query language for the wrapper specification. The work of Nakhimovsky [Nak01] is perhaps the closest to our view of describing and implementing wrappers: while he uses formal grammars and parser generators to create a skeleton parser for XML generation, we write a template document for the XML result and embed in it partitioning strings that break down the input data into its parts. Opposed to Nakhimovsky's approach, our wrapper specification is declarative.

The rest of the paper is organized as follows. Section 2 describes the main features of our wrapper specification language using three typical examples taken from the applications considered in our project. The implementation principles are sketched in Section 3, and Section 4 concludes the paper.

2 Declarative specification of wrappers

This section presents the design principles and a set of features of the declarative wrapper specification language XW through three typical examples. In the first example the text data flow is partitioned by positional information, and in the second one by separator strings. The third example considers binary data consisting of consecutive typed fields whose lengths are defined by their types.

2.1 General ideas of XW

Our design principle has been that simple and often occurring cases should be simple to specify. Our wrapper specification language does not try to be a full-fledged transformation language. Instead, the goal is to provide a language that is easier to use than a full transformation language like XSLT yet powerful enough to most typical needs to wrap data. We expect XW wrapping to be mainly used as the initial step in the XML processing of data. If more complex XML manipulation is called for, the result of wrapping can be processed further by applying any appropriate XML technologies like SAX [SAX00], DOM [ABC98] or XSLT [Cla99].

An XW wrapper specification is a well-formed XML document composed of output elements as well as elements and attributes belonging to the XW namespace `http://www.cs.uku.fi/XW/2001`. Elements outside the XW namespace produce elements required by the generated XML representation. The XW language has been influenced by a number of XML technologies. The template-based way of generating output elements is inspired by XSLT, and the ways to model repetitive and alternative input structures have been influenced by XML Schema [TBM01].

The specification defines a template for the XML-formatted output document with embedded instructions describing how the input document is divided into parts and how its structure is to be altered. Basically, the wrapper specification describes the structure of the output document and how input data is arranged into its elements. Thus, the structure of the wrapper specification is implied by the structure of the input data. Consecutive parts in input data are described by consecutive output elements in the wrapper specification and subparts of a part are described by child elements of the corresponding element. Description of how a part is divided into subparts is placed either in the corresponding element or in its child elements. Elements outside the XW namespace produce elements to the output data with the the corresponding part of input data as its content, either as straight text or as child elements given in the wrapper specification.

The structure of input data can be modified, with some restrictions. Instead of replacing a part of the input by an output element, an XW element can be used to process it differently. For instance, all the child elements or the entire text content of a text element can be taken out of that element replacing it in the output. Also, an element can be created to enclose a group of elements or the text content of sibling text elements. Any part of the input data can also be discarded. However, the order of the parts in input data cannot be changed.

2.2 Positional text data

This example deals with a phone invoice that is divided into three parts: identifier data, specification and invoice data. The actual content is identified by headings and by positional information. For example, the heading `INVOICE` starts the identifier data part. These three parts are divided into subparts, which are rows in this case. The data on the rows is indentified by positions. For example, the customer number is defined to be located on the second row starting at position 65 and ending at position 76.

```
INVOICE                                           INVOICE NUMBER: 44196
                                                 CUSTOMER NUMBER: 25272
                                              PERSONAL REFERENCE: WORK


John Smith
Garden Avenue 40
43234 Bigtown


PHONE SPECIFICATION

DATE            UNITS    DURATION    NUMBER    PRICE
11.1.1992           5     307 min     37126    50.00
23.6.1995          10     193 min     53829   122.00
22.11.1992         10       6 min     51866   231.00
18.5.1995          18     316 min     21488   272.00
24.6.1995          13      16 min     42378   232.00
----------------------------------------------------------------------
John Smith
Garden Avenue 40
43234 Bigtown

                             595324
                             17.8.1996   907.00
```

Each XW wrapper specification has the root element `xw:wrapper` having the attributes `xw:name` to define an optional name for the wrapper, `xw:sourcetype` to tell the type of the input data and **xmlns:xw** to specify the XW namespace. The wrapper for phone invoices is as follows.[1]

```
<xw:wrapper xw:name="phone-invoice" xw:sourcetype="text"
            xmlns:xw="http://www.cs.uku.fi/XW/2001" >
   <invoice  xw:occurs="unbounded">
      <identifierdata xw:starter="\^INVOICE"
                      xw:childterminator="\n"
                      xw:ignoreemptysubpart="true">
         <invoicenumber xw:position="58 69"/>
         <customernumber xw:position="65 76"/>
         <personalreference xw:position="65 76"/>
         <name xw:position="1 22"/>
         <streetaddress xw:position="1 22"/>
         <postoffice xw:position="1 22"/>
      </identifierdata>
      <specification xw:starter="\^PHONE SPECIFICATION"
                     xw:childterminator="\n"
                     xw:ignoreemptysubpart="true">
         <xw:ignore/>
         <specificationrow xw:occurs="unbounded">
            <date xw:position="1 12"/>
            <units xw:position="14 22"/>
            <duration xw:position="24 33"/>
            <number xw:position="35 43"/>
            <price xw:position="45 52"/>
         </specificationrow>
      </specification>
      <invoicedata xw:starter="\^----------"
                   xw:childterminator="\n"
                   xw:ignoreemptysubpart="true">
         <xw:ignore xw:occurs="4"/>
         <reference xw:position="30 48"/>
         <xw:collapse>
            <duedate xw:position="30 39">
            <totalsum xw:position="42 50">
         </xw:collapse>
      </invoicedata>
   </invoice>
</xw:wrapper>
```

[1]The line numbers in these examples have been added to wrapper specifications only for presentational purposes.

The multiple root elements of the output data are called `invoice` (line 3). Since invoices can occur any number of times in the input data, the element `invoice` has the attribute `xw:occurs` with the value `unbounded`. For the three parts of the invoice there are elements `identifierdata` (lines 4-13), `specification` (lines 14-25) and `invoicedata` (lines 26-35). Each of these has the three attributes `xw:starter`, `xw:childterminator` and `xw:ignoreemptysubpart`. The attribute `xw:starter` tells the starting string in the input data for the current element. For example, for `identification` the attribute `xw:starter` (line 4) has a value `\^INVOICE` (`\^` denotes the beginning of a row). This means that the string `INVOICE` at the beginning of a row of the invoice indicates the starting of the `identifierdata` part. The attribute `xw:childterminator` (lines 5, 15, 27) specifies the terminating string of the subparts. In this example it is the end-of-the-line character (denoted by `\n`) for all the three parts. So the parts are divided into subparts that are in fact rows. The third attribute `xw:ignoreemptysubpart` (lines 6, 16, 28) with the value `true` indicates that any empty subparts, empty lines in this case, should be ignored while splitting the input data into parts.

Both the starting and ending positions of the actual data to be added to the output data are given on each element with the attribute `xw:position`. It specifies the bounding positions of the text inside the enclosing part. The first position inside a part is one. When defining a part with the `xw:position` attribute the wrapper removes all whitespace characters from the beginning and the end of the text. The invoice number, for example, is added into `invoicenumber` element (line 7) that has the attribute `xw:position` with the value `58 69` for starting and ending positions. Since the end of the `invoicenumber` field is empty, the length of the output element's content will be in this case only five characters instead of twelve. A careful reader notices that the starting positions of the invoice number and the custom number differ although they both start at the same place in the example invoice. The reason for the difference is the following. The positions are counted starting *after* the end of the `xw:starter` string, which is in this case the string `INVOICE` of seven characters.

The element `xw:ignore` (line 17) is used for removing recognized parts of the input data. This element is used to remove the heading row containing headings `DATE`, `UNITS`, `DURATION`, `NUMBER` and `PRICE`.

The element `xw:collapse` (lines 31-34) reduces the hierarchical structure of the input data. In this example invoice, the row of the input data containing the date and the total sum of the invoice is collapsed. As a result, two elements will be created from a single data row and placed at the same hierarchy level with the preceding `reference` element (line 30).

The wrapper produces a phone invoice in XML form as follows:

```
<invoice>
   <identifierdata>
      <invoicenumber>44196</invoicenumber>
      <customernumber>25272</customernumber>
      <personalreference>WORK</personalreference>
      <name>John Smith</name>
      <streetaddress>Garden Avenue 40</streetaddress>
      <postoffice>43234 Bigtown</postoffice>
   </identifierdata>
   <specification>
      <specificationrow>
         <date>11.1.1992</date>
```

```
        <units>5</units>
        <duration>307 min</duration>
        <number>37126</number>
        <price>50.00</price>
      </specificationrow>
      <specificationrow>
        <date>23.6.1995</date>
        <units>10</units>
        <duration>193 min</duration>
        <number>53829</number>
        <price>122.00</price>
      </specificationrow>
      ...
    </specification>
    <invoicedata>
      <reference>595324</reference>
      <duedate>17.8.1996</duedate>
      <totalsum>907.00</totalsum>
    </invoicedata>
</invoice>
```

2.3 Separator-delimited text data

This example processes a data exchange format of Health Level Seven (HL7) version 2.3 [HLS01] messages. The highest level of the hierarchical structure is the message itself. The second-level structure parts, rows, are separated by end-of-the-line characters. Rows are divided into fields separated by pipe characters (|), and their subfields are separated by carets (^). On each row, the first field contains a three-letter identification of its content. Some rows have fixed places inside the message while others can occur in mixed order with other rows. As an example we use the following message which is a response to a clinical laboratory request.

```
MSH|^~\&|KL-Lab||CCIMS|RDNT01|200001071300||ORU^R01|T12345|P|2.3||||AL
PID|||311244A0112|ExamMod1|Smith^John||19441231|M|||Street
11 A 11 ^^City^^12345||099-9876543|099-56473|British|S|N||311244A0113
OBR||76551|Res_01||||20000107060000||||||||||||||||CH|C
OBX||NM|1535^aB-pO2^||11||||||F
NTE|||This is a comment for aB-pO2.
NTE|||Another comment for aB-pO2.
OBX||NM|1026^S -ALAT^||61|||*|||F
```

An HL7 message can be read as follows. The first row begins with the identifier `MSH` which denotes the beginning of the message. Its next field introduces separators that are used in the message. The seventh field contains the date and time of creating the message. The second row (written on two lines for presentational reasons) has an identifier PID and consists of patient information. Its fourth field contains the social security number and its sixth field the name of the patient. The ninth field informs the sex of the patient. The third row having an identifier OBR contains the response information from the laboratory. The next four rows have the identifier OBX or NTE. Each OBX row contains a result for the test. The NTE rows comment the result of the previous OBX row.

An XW wrapper specification is presented below for transforming the previous message to an XML document instance

```
1  <xw:wrapper xw:name='HL7' xw:sourcetype='text'
2              xmlns:xw='http://www.cs.uku.fi/XW/2001'>
3     <response xw:starter='\^MSH'  xw:childseparator='\n'
4              xw:occurs='unbounded' xw:discardemptyelement='true'>
5        <xw:ignore/>
6       ...
7        <xw:CHOICE xw:occurs='unbounded'>
8           <xw:collapse xw:starter='\^OBX' xw:childseparator='|'>
9              <xw:ignore xw:occurs='3' />
10             <observation/>
11             <xw:ignore/>
12             <result/>
13             <xw:ignore xw:occurs='2' />
14             <flag/>
14             <xw:ignore xw:occurs='2' />
16             <responsetype/>
17          </xw:collapse>
18          <xw:ELEMENT xw:name='comment'>
19             <xw:collapse xw:starter='\^NTE'  xw:childseparator='|'
20                          xw:occurs='unbounded'>
21                <xw:ignore xw:occurs='3'>
22                <xw:collapse/>
23             </xw:collapse>
24          </xw:ELEMENT>
25       </xw:CHOICE>
26    </response>
27 </xw:wrapper>
```

The root element of the output data is called `response`. The creation of `response` elements is controlled by four attributes (lines 3-4). The position that starts the part of the input data to be made into a `response` element is identified by the attribute `xw:starter`. Its value in this example is the string `\^MSH`. The separator of the next hierarchical level (row) is identified by using the attribute `xw:childseparator` with the value `\n`. Attribute `xw:occurs` expresses the number of repetitions of the response element. Attribute `xw:discardemptyelement` with the value `true` defines that only elemetns that have some content shall be created to the output.

The element `xw:CHOICE` (lines 7-25) is used for choosing between alternative parts of the input. It does not create a new element to the output data but chooses the first subelement that matches the input data. Its attribute `xw:occurs` (line 7) is used to inform that the selection can be repeated several times. Attribute `xw:starter` that is used for identifying the matching part is mandatory for all the childelements (`xw:collapse` on lines 8-17 and `xw:ELEMENT` on lines 18-24) of the `xw:CHOICE` element. Since an `xw:ELEMENT` describes output structures rather than input parts, the rule applies to its child elements instead. In this example there are two kinds of rows to choose, identified by starters `OBX` (line 8) or `NTE` (line 19).

If a new hierarchical level that does not occur in the input data should be added to the output, the designer can use the element `xw:ELEMENT`. This element adds a new hierarchy level and names it according to the value of attribute `xw:name`. In this example an element called `comment` is added to the output data (lines 18-24).

The attribute `xw:childseparator` (line 3) is inherited through the elements `xw:CHOICE` and `xw:ELEMENT` to their children because these elements describe parts of input data indirectly by their child elements.

The specified wrapper will convert the previous input data into the following output data:

```
<response>
   ...
   <observation>1535^aB-pO2^</observation>
   <result>11</result>
   <responsetype>F</responsetype>
   <comment>This is a comment for aB-pO2.Another comment for aB-pO2.</comment>
   <observation>1026^S -ALAT^</observation>
   <result>61</result>
   <flag>*</flag>
   ...
</response>
```

2.4 Binary data with typed fields

In this example the input is binary data of a fictional IP-based data communications protocol consisting of packets of consecutive typed fields. A packet contains a segment that is either a whole datagram or a part of one. Most of the fields in a packet are of simple types but the actual data, the payload, is an array of bytes. The input data, a single packet, is depicted in Figure 1. The first row describes the lengths, the second row the names and the third row the types of the sequential fields. The sizes and domains of the types used in XW are the same as those in XML Schema Definition Language (XSD) [BiM01] and Java [GJS96]. The type defines the length of the field and the interpretation of its bits. The fields are as follows: `len` means the length of the whole packet in bits, `chk` is a checksum, `id` identifies the datagram, `off` is the segment offset describing the position of this segment in the whole datagram, `src` is a 32-bit IP-address for the source, `dst` is a 32-bit IP-address for the destination and `pay` is the actual data, the payload.

length	16b	16b	16b	16b	4*8b	4*8b	varies
name	len	chk	id	off	src	dst	pay
type	short	short	short	short	4*byte	4*byte	array of bytes

Figure 1: A sample datagram structure

The wrapper specification is as follows:

```
1  <xw:wrapper xw:name="IP-like-protocol" xw:sourcetype="binary"
2              xmlns:xw="http://www.cs.uku.fi/XW/2001">
3   <datagram>
4     <xw:ignore xw:name="total-length" xw:type="short"/>
5     <checksum xw:type="short"/>
6     <id xw:type="short"/>
7     <segment-offset xw:type="short"/>
8     <xw:ELEMENT xw:name="source-address">
9       <a xw:type="byte"/>
10      <b xw:type="byte"/>
11      <c xw:type="byte"/>
12      <d xw:type="byte"/>
13    </xw:ELEMENT>
14    <xw:ELEMENT xw:name="destination-address">
15      <a xw:type="byte"/>
16      <b xw:type="byte"/>
17      <c xw:type="byte"/>
18      <d xw:type="byte"/>
19    </xw:ELEMENT>
20    <xw:ELEMENT name="payload">
21        <xw:collapse xw:occurs="total-length - 16" xw:type="byte"
22                     xw:numeric-output-format="hexadecimal"/>
23     </xw:ELEMENT>
24   </datagram>
25  </xw:wrapper>
```

Since the input data is binary, the wrapper specification has its `xw:sourcetype` attribute set to `binary` (line 1). The first field `len` is ignored but its value is stored in the variable `total-length` for later use (line 4). For fields `checksum`, `id` and `segment-offset` the corresponding elements need only be named and their types described with the `xw:type` attribute (lines 5-7).

Both the source address and the destination address are 32-bit fields. The content of each is not used as such but rather as a collection of four bytes because the different parts of the address need to be specified separately in the output. Since XW does not recognize structure in binary data other than consecutive fields, no corresponding element can be named for a collection of fields. Thus, an element needs to be created with `xw:ELEMENT` to enclose the four parts of the source address as well as the destination address (lines 8-13, 14-19). Similarly, an element needs to be created for the bytes of the payload (lines 20-23). No elements should be created for the individual bytes, though, and that is why `xw:collapse` is used for the iteration of the bytes (lines 21-22).

Variables as well as numeric literal constants can be used in `xw:occurs` and simple arithmetic can be performed with them (line 21). Such a variable is created by naming it with the attribute `xw:name`. Its value is the content of the part corresponding to the element that has the attribute. In our wrapper specification, the number of `bytes` in the `pay` field is calculated by subtracting the combined length of the other fields (16 bytes) from the value of `total-length` (line 21). The number is used as the number of repetitions of the `xw:collapse` element to process all the bytes of the payload (line 21-22).

The type of the fields in the payload is `xw:byte` which would produce decimal numbers into the output. In this example (line 22) a hexadecimal notation is required so an `xw:numeric-output-format` attribute is given the value `hexadecimal`. In the case of bytes it produces 2-digit hexadecimals, and in general the number of digits required for representing any value of the given type.

The input data is converted by the specified wrapper into the following XML form:

```
<datagram>
  <checksum>397485</checksum>
  <id>37</id>
  <segment-offset>0</segment-offset>
  <source-address><a>193</a><b>167</b><c>232</c><d>253</d></source-address>
  <destination-address><a>193</a><b>167</b><c>224</c><d>8</d>
  </destination-address>
  <payload>e6a9ff120a</payload>
</datagram>
```

3 Implementation plans

XW will be implemented using existing XML tools for loading the wrapper specification, and Java for realizing its input-parsing and output-generating behaviour.

The wrapper specification is read in using an XML parser, and manipulated internally as a DOM tree called a *wrapping tree*. The input data is divided hierarchically into nested parts, which are matched against the nodes of the wrapping tree. For this, a preprocessing phase computes sets of delimiting strings called the *start set* and the *end set* for the nodes of the wrapping tree. The sets are computed from partitioning information, for example, from the starter, separator and terminator strings introduced in the wrapper specification. The role of the start set and the end set of a node is as follows: While scanning the input data, the next part of input starting by any string in the start set and extending to the first occurrence of any string in the end set will be matched by the node being processed thus providing content for a result element corresponding to the node.

The process that matches the wrapping tree against the input data creates the output XML form through SAX events like `characters`, `startElement` and `endElement`. An implication of this design decision is that it is easy to provide different APIs to the wrapper: The specified wrapper could easily be integrated to a larger application as a component that provides the resulting XML document either more efficiently through SAX events or in a more programmer-friendly way as a complete DOM tree. It is equally easy to make the wrapper a stand-alone filter that generates a textual representation of the resulting XML document by simply interpreting the SAX events as straight-forward text generating commands.

4 Conclusion

We have described a declarative XML-based language called XW for specifying wrappers for non-XML data sources. XW is suitable for wrapping any kind of serialized data consisting of recognizable parts like text lines or binary records and fields. The described wrappers process the input data sequentially recognizing parts and arranging them to a hierarchical structure defined by the wrapper specification. As a minimal transformation, the wrapping transforms recognized parts of the input into XML elements as defined by the wrapper specification, preserving the order of the input data. In addition, it is possible to remove parts and hierarchy levels of the input data and to create additional hierarchy levels to the output data. Since the language is not designed for any specific application domain it should be easily applicable to wrap a wide variety of different serialized data.

The implementation of XW is under way. Parallel to implementing the wrapper specification language, we are considering various extensions to its basic features introduced in this paper. Possible extensions include XSLT-like constructs for generating attributes and content from the input data. Also, enhancements to the ability to recognize alternative parts will be considered. Such extensions should not weaken the declarative simplicity and efficient implementability of XW. On the other hand they might eliminate the need for further XSLT processing in some typical wrapping applications and thus both simplify the design and speed up the execution of XML wrappers.

Acknowledgements

This research was financially supported by TEKES (the National Technology Agency of Finland) and the following companies and organizations: Deio Corporation, Enfo Group Plc, JSOP Interactive, Kuopio University Hospital, Medigroup Ltd, SysOpen Plc and TietoEnator Corporation. The authors would also like to thank Professor Martti Penttonen and post-graduate students Sami Komulainen and Paula Leinonen for their help during this work.

References

[ABC98] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. Le Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, L. Wood (editors). Document Object Model (DOM) Level 1 Specification. W3C Recommendation 1 October, 1998. Available at http://www.w3.org/TR/REC-DOM-Level-1.

[Ade98] B. Adelberg. NoDoSE - A tool for semi-automatically extracting structured and semistructured data from text documents. *SIGMOD Record*, 27(2):283-294, 1998.

[AsK97] N. Ashish and C. A. Knoblock. Wrapper generation for semi-structured Internet sources. *SIGMOD Record*, 26(4):8-15, 1997.

[BGL99] C. Baru, A. Gupta, B. Ludäscher, R. Marciano, Y. Papakonstantinou, P. Velikhov, and V. Chu. XML-based information mediation with MIX. *SIGMOD Record*, 28(2):597–599, 1999.

[BiM01] P.V. Biron, A. Malhotra (editors). *XML Schema Part 2: Datatypes*. W3C Recommendation 02 May 2001. Available at http://www.w3.org/TR/xmlschema-2/.

[Cla99] J. Clark (editor). *XSL Transformations (XSLT) Version 1.0*. W3C Recommendation 16 November 1999. Available at http://www.w3.org/TR/xslt.

[GJS96] J. Gosling, B. Joy, G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[GMP95] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom. The TSIMMIS approach to mediation: Data models and languages. In Conference proceedings of the NGITS (*Next Generation Information Technologies and Systems*), Naharia, Israel, June 27-29 1995. Available at http://dbpubs.standford.edu/pub/1995-7.

[HGM97] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting semistructured information from the Web. In Proceedings of Workshop on *Management of Semi-structured Data*, pages 18-25, 1997.

[HLS01] Health Level Seven Home Page. HL7 standards, 2001. Available at http://www.hl7.org.

[LHB99] L. Liu, W. Han, D. Buttler, C. Pu, and W. Tang. An XML-based wrapper generator for Web information extraction. *SIGMOD Record*, 28(2):540-543, 1999.

[LPH00] L. Liu, C. Pu, and W. Han. XWRAP: An XML-enabled wrapper construction system for Web information sources. In Proceedings of the *International Conference on Data Engineering* (ICDE), pages 611-621, 2000.

[Nak01] A. Nakhimovsky. Using parser-generator to convert legacy data formats to XML. In *XML Europe 2001*, Berlin, Germany, 21-25 May 2001.

[SaA99] A. Sahuguet and F. Azavant. Building light-weight wrappers for legacy Web datasources using W4F. In Proceedings of the VLDB (*Very Large Data Bases*), pages 738-741, 1999.

[SAX00] SAX 2.0: The Simple API for XML. May 2000. Available at http://www.megginson.com/SAX/sax.html.

[TBM01] H.S. Thompson, D. Beech, M. Maloney, N. Mendelsohn (editors). *XML Schema Part 1: Structures*. W3C Recommendation 2 May 2001. Available at http://www.w3.org/TR/xmlschema-1/.

[Wie92] G. Wiederhold. Mediators in the architecture of future information systems. *Computer*, 25(11):38–49, 1992.