Declarative XML Wrapping of Data

Merja Ek, Heli Hakkarainen,
Pekka Kilpeläinen, Tommi Penttinen

Report A/2002/2

ISBN 951-781-262-0

UNIVERSITY OF KUOPIO

Department of Computer Science and Applied Mathematics

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

Declarative XML Wrapping of Data

M. Ek, H. Hakkarainen, P. Kilpeläinen, T. Penttinen
Department of Computer Science and Applied Mathematics
University of Kuopio, Finland

{ekmerja, hihakkar, kilpelai, tpenttin}@cs.uku.fi

ABSTRACT

XML provides a standard technology for archiving information and for transferring it between co-operating systems as well-formed documents. Translation of legacy data to an XML-based representation, often called "XML wrapping", is a recurring practical problem in the utilization of XML. We attack this problem by describing a declarative XML wrapper description language called XW (XML Wrapper). XW is designed to be a convenient language for describing typical XML wrapping of data. The design of the language is influenced by a number of XML technologies, such as XML Namespaces, XML Schema, and XSLT. We are currently applying XW for automating the conversion of medical messages and mass-printing material to XML documents. We discuss the implementation techniques and principles of executing declarative XW wrapper descriptions. The XW implementation provides a SAX (Simple API for XML) interface, which means that it can be easily and efficiently used within other applications to access non-XML data sources as if they were XML documents seen through an XML parser.

Categories and Subject Descriptors

I.7.2 [**Document and Text Processing**]: Document Preparation—*languages and systems, markup languages, standards*; J.7 [**Computer Applications**]: Computers in Other Systems—*publishing*

General Terms

Design, Languages

Keywords

XML, wrapping, data translation, conversion

1. INTRODUCTION

XML provides a standard technology for archiving information and for transferring it between co-operating systems as well-formed documents. Exchange formats based on XML are used in several application areas, especially in e-commerce. (See, e.g., [7]). XML is also used as a standard structured-document representation in document production systems. Despite the increasing use of XML, many non-XML data sources do persist.

Translation of legacy data to an XML-based representation, often called "XML wrapping", is a recurring practical problem in the utilization of XML. Recent versions of many commercial systems, e.g. Oracle [14], provide built-in interfaces for externalizing their data as XML documents. Such built-in interfaces are still relatively rare, though, and often ad-hoc interfaces and translators are written for translating results of queries or data files to XML documents. It is both tedious and costly to develop and to maintain such ad-hoc wrappers. Converting and collecting data into structured XML documents is estimated to often be the most costly area in the entire document production process [19].

We address the problem of XML wrapping by describing a declarative XML wrapper description language called *XW (XML Wrapper)*. XW is designed to be a convenient language for describing typical XML wrapping of both textual and binary data formats. It is based on describing the structure of the input data and the output document with an XML document template, which should be an intuitive and familiar model for users of XML. XW is designed for relatively simple initial conversion of data to XML – it does not try to be a complete document transformation language like, say, XSLT [5]. An initial XW wrapping may be followed by additional steps using standard XML processing technologies like SAX (Simple API for XML) [16], DOM (Document Object Model) [9] or XSLT, if needed. On the other hand, an initial conversion to XML is a necessary prerequisite for applying any XML technologies.

We have developed a prototype implementation of XW. It provides a SAX interface, which means that XW can be easily and efficiently used within other applications to access non-XML data sources as if they were XML documents seen through an XML parser.

The rest of the paper is organized as follows. The design of XW is discussed in Section 2. We are currently applying XW for automating the conversion of medical messages and mass-printing material to XML documents. Section 3 discusses these applications of XW. Interesting opportunities of automating the processing of different mass-printing materials by applying XW wrappers to their control files are outlined in Section 3.2. The implementation techniques and principles of executing XW wrappers are discussed in Section 4. Section 5 briefly reviews related work, and Section 6 is a short conclusion.

2. DESIGN OF XML WRAPPER

XW is the combination of a declarative wrapper description language and an implementation, an *XW processor*, that interpretes and executes XW wrapper descriptions. XW can be used to describe the conversion of both non-XML text and binary data into XML. An XW wrapper description is itself an XML document and acts as a template for the resulting XML document. It also embeds a description of the structure of the input data. The actual conversion is done by the XW processor that reads source data and converts it into XML as per the instruction of the wrapper description. Below, XW wrapper descriptions will simply be called *XW wrappers*.

XW has been influenced by several XML technologies. XML namespaces are used to separate XW elements and attributes, the "reserved words", from result elements and attributes. XW elements and attributes are defined in namespace `http://www.cs.uku.fi/XW/2001`. We normally use the prefix `xw` to denote this namespace. The way the result is described in XW through a template resembles that of XSLT. Element creation through `xw:ELEMENT` was borrowed from XSLT and XML Schema [18]. The types of fields supported in XW for binary input, expression of alternative parts through `xw:CHOICE` and occurrence indicators `xw:minoccurs` and `xw:maxoccurs` were also inspired by XML Schema.

An XW wrapper has the root element `xw:wrapper` that encloses the actual wrapper description. The type of input, either `text` or `binary`, is specified through the attribute `xw:sourcetype`. Character encoding for input and output can be specified through attributes `xw:inputencoding` and `xw:outputencoding`, respectively. An example is shown in Fig. 1.

```
<xw:wrapper xmlns:xw="http://www.cs.uku.fi/XW/2001"
            xw:sourcetype="text"
            xw:outputencoding="ISO-8859-1">
    ...
</xw:wrapper>
```

Figure 1: An example of `xw:wrapper` element

With XW, input is hierarchically divided into nested parts. Parts of textual input are identified by literal delimiting strings or by character positions. With binary input the parts are simply typed fields. Parts can be optional, alternative or repeated.

The structure of input data is modelled by the structure of the wrapper. For each part in input there is a corresponding element in the wrapper. Consecutive parts correspond to consecutive elements. Delimiting strings for a part can be specified in the corresponding element through XW attributes. Attributes `xw:starter` and `xw:terminator` specify strings that start and end a part, respectively. XW is designed for easy selection of relevant parts of input. For this reason non-matching parts of input are skipped when searching for delimiters specified in an XW wrapper.

```
Part A Words of comment
a1|a2|a3
a4
Part B
b1|b2
b3|b4
---
c1|c2
c3|c4
```

Figure 2: Sample data

For example, in Fig. 2 we have input data consisting of three parts starting with "`Part A`", "`Part B`" and "`---`". Each one of these delimiters appears at the start of a line, a condition that can be represented in XW by way of `\^` . An XW wrapper skeleton is presented in Fig. 3, with missing definitions marked with ellipsis.

```
<xw:wrapper xmlns:xw="http://www.cs.uku.fi/XW/2001"
            xw:sourcetype="text"
            xw:outputencoding="ISO-8859-1">
    <part-a xw:starter="\^Part A"> ... </part-a>
    <part-b xw:starter="\^Part B"> ... </part-b>
    <part-c xw:starter="\^---"> ... </part-c>
</xw:wrapper>
```

Figure 3: XW wrapper skeleton for the sample data

Nested parts in input are described by nested elements in the XW wrapper. That is, for a part and its corresponding element, the sub-parts correspond to the child elements. If the same delimiter starts, ends or separates all the sub-parts of a part, the delimiter can be defined in the element corresponding to the enclosing part through `xw:childstarter`, `xw:childterminator` or `xw:childseparator`, respectively. In the sample data all the three parts are divided into lines. For this, XW provides a platform-independent notation `\n` for the end of a line. The second line of the first part is further divided into sub-parts by the separator '`|`'. The starter of the first part has a comment following it that is of no interest to us. In XW a part always begins right after its starter, if one is specified, and thus the comment is considered the first sub-part. A part can be ignored, by using the element `xw:ignore`, in the sense that the part is read from the input but no output is produced.

Description of the first part, taken from the complete XW wrapper, is shown in Fig. 4. For the result, refer to the whole result XML document shown in Fig. 5. The definition `xw:childterminator="\n"` in element `part-a` states that the child elements `xw:ignore`, `line-1` and `line-2` correspond to parts that end with an end-of-line character, i.e., lines. Thus, `xw:ignore` corresponds to "` Words of comment`", line-1 corresponds to "a1|a2|a3"and line-2 corresponds to "a4". Similarly, the three a elements correspond to "a1", "a2" and "a3".

```
<part-a xw:starter="\^Part A"
        xw:childterminator="\n">
    <xw:ignore/>  <!-- rest of "Part A" line -->
    <line-1 xw:childseparator="|">
        <a/> <a/> <a/>
    </line-1>
    <line-2/>
</part-a>
...
```

Figure 4: XW wrapper for the first part of the sample data

The main focus in the design of XW has been to convert data, as opposed to prose, into data-oriented XML, i.e. XML with element content only. Pure element content follows naturally from the way text content in an input part is divided among the child elements of the corresponding element: only empty wrapper elements are left with text to output. Mixed content, that is, text content between elements, can be produced, if desired (see xw:collapse below).

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<part-a>
  <line-1>
    <a>a1</a> <a>a2</a> <a>a3</a>
  </line-1>
  <line-2>a4</line-2>
</part-a>
<part-b>
  <line>
    <b>b1</b> <bb>b2</bb>
  </line>
  <line>
    <b>b3</b> <bb>b4</bb>
  </line>
</part-b>
<part-c>
  <c>c1</c> <cc>c2</cc>
  <c>c3</c> <cc>c4</cc>
</part-c>
```

Figure 5: Result XML document converted from the sample data

Repetition of a part is described in the corresponding element through the attributes xw:minoccurs and xw:maxoccurs. The three a elements in Fig. 4 could have been expressed simply by <a xw:minoccurs="3" xw:maxoccurs="3"/>. Optionality can be specified through xw:minoccurs="0". Both attributes default to 1 if unspecified. In the sample data, the second part has one or more lines. An unspecified number of occurrences, "as many parts as there are in the input", is specified through unbounded. Description of this part is shown in Fig. 6 and the result can be found in Fig. 5.

```
...
<part-b xw:starter="\^Part B\n"
        xw:childterminator="\n">
    <line xw:childseparator="|"
          xw:maxoccurs="unbounded">
        <b/> <bb/>
    </line>
</part-b>
...
```

Figure 6: XW wrapper for the second part of the sample data

The structure of the resulting XML document can be modified in two ways. Levels of hierarchy can either be removed or added. Naming an element xw:collapse will produce elements for the sub-parts of the corresponding part instead of the part itself. If an xw:collapse element is empty, only the corresponding text content is produced to output. This could be used to produce mixed content. In the sample data, the last part is divided into lines, but we only want to create elements out of their fields separated by '|'. Thus, instead of naming a result element for the lines, we use xw:collapse. See Fig. 7 for the wrapper and Fig. 5 for the result.

```
...
<part-c xw:starter="\^---\n" xw:childterminator="\n">
    <xw:collapse xw:childseparator="|"
                 xw:minoccurs="0"
                 xw:maxoccurs="unbounded">
        <c/> <cc/>
    </xw:collapse>
</part-c>
...
```

Figure 7: XW wrapper for the last part of the sample data

As an inverse operation of xw:collapse, a group of elements can be enclosed in an element even if no corresponding part can be found for it. The group is simply enclosed in xw:ELEMENT and the name of the enclosing element is specified through the attribute xw:name. If we wanted to enclose the elements part-b and part-c in an element body, we would write the XW wrapper as shown in Fig. 8.

```
<xw:wrapper xmlns:xw="http://www.cs.uku.fi/XW/2001"
            xw:sourcetype="text"
            xw:outputencoding="ISO-8859-1">
    <part-a xw:starter="\^Part A">...</part-a>
    <xw:ELEMENT xw:name="body">
        <part-b xw:starter="\^Part B\n">...</part-b>
        <part-c xw:starter="\^---\n">...</part-c>
    </xw:ELEMENT>
</xw:wrapper>
```

Figure 8: XW wrapper skeleton for the sample data

Textual input can also be divided into parts by character positions, usually in combination with delimiters. These positions are measured from the end of the last delimiter found, starting from 1. Positions are specified through the attribute `xw:position` as a space-separated pair of start and end positions. For example, for lines of names, with the surname between the first and the 20th character and the given name between the 22nd and the 31st, we could use the XW wrapper in Fig. 9.

```
<xw:wrapper xmlns:xw="http://www.cs.uku.fi/XW/2001"
            xw:sourcetype="text"
            xw:outputencoding="ISO-8859-1">
    <names xw:childterminator="\n">
        <name xw:maxoccurs="unbounded">
            <surname xw:position="1 20"/>
            <given xw:position="22 31"/>
        </name>
    </names>
</xw:wrapper>
```

Figure 9: XW wrapper for lines of names

Alternative parts that can be identified through a starting string can be represented with `xw:CHOICE`. For a discussion, see the example on HL7 messages in Section 3.

The elements `xw:ELEMENT` and `xw:CHOICE` behave differently from other elements with regard to parent element's `xw:childstarter`, `xw:childseparator` and `xw:childterminator` attributes. Since these two elements have no corresponding part in input, the delimiter does not affect them, but their child elements instead.[1] For an example, see Section 3.1.

A current limitation of XW is that the order of input parts cannot be changed. Any two input parts, or rather their content, must be in the same order in output as they were in input. Work is under way to enable the reordering of data without sacrificing the simplicity and efficiency of XW.

3. APPLICATIONS OF XW

We have applied XW to XML wrapping of medical messages and mass printing material. We first discuss the translation of medical messages into XML, which provides a rather extensive example of the features of XW.

3.1 XML wrapping of medical messages

Health Level Seven (HL7) is an organization that develops widely used standards for interchanging medical information between healthcare applications. Earlier versions of HL7 standards use field-based messages while the newest versions use an XML encoding [8]. In this example we consider XML wrapping of HL7 version 2.3 messages, which is needed for upgrading legacy systems to use the newer XML-based exchange formats [6].

The source of the wrapping consists of a number of messages divided into lines. Lines are divided into fields by pipe characters (`|`). On each line, the first field contains a three-letter identification of its content. Some lines have fixed places inside the message while others can occur in mixed order. Fig. 10 shows fragments of an HL7 message that represents a response to a clinical laboratory request.

```
1 MSH|^~\&|KL-Lab||CCIMS|RDNT01|200001071300|| ...
2 PID|||311244A0112|ExamMod1|Doe^John||19441231|...
3 OBR||76551|Res_01||||20000107060000|| ... |CH|C
4 OBX||NM|1535^aB-pO2^||11||||||F
5 NTE|||Comment.
6 NTE|||Another comment.
7 OBX||NM|1026^S -ALAT^||61|||*|||F
```

Figure 10: Fragments of an HL7 message

We only explain some central parts of the message, concentrating on the observations reported in a message. The first line begins with the message-begin identifier `MSH`. The third line, with identifier `OBR`, carries general information about the observation request. The next four lines (4–7) are identified by either `OBX` or `NTE`. Each `OBX` line contains a result for the test, and the `NTE` lines provide additional comments to the previous `OBX` line.

```
<xw:wrapper xw:sourcetype='text'
        xmlns:xw='http://www.cs.uku.fi/XW/2001'>
   <response xw:starter='\^MSH'
             xw:maxoccurs='unbounded'
             xw:childseparator='\n' >
      ...
      <xw:CHOICE xw:minoccurs='0'
                 xw:maxoccurs='unbounded'>
         <xw:collapse xw:starter='\^OBX'
                      xw:childseparator='|'>
            <xw:ignore xw:minoccurs='3'
                       xw:maxoccurs='3 />
            <observation/>
            <xw:ignore/>
            <result/>
            <xw:ignore xw:minoccurs='5'
                       xw:maxoccurs='5' />
            <responsetype/>
         </xw:collapse>
         <xw:ELEMENT xw:name='comment'>
            <xw:collapse xw:starter='\^NTE'
                         xw:childseparator='|'
                         xw:minoccurs='0'
                         xw:maxoccurs='unbounded'>
               <xw:ignore xw:minoccurs='3'
                          xw:maxoccurs='3 />
               <xw:collapse/>
            </xw:collapse>
         </xw:ELEMENT>
      </xw:CHOICE>
   </response>
</xw:wrapper>
```

Figure 11: Fragments of a wrapper for HL7 messages

An XW wrapper specification for transforming such messages to XML documents is shown in Fig. 11. The root el-

[1] To make the distinction clear these elements are written in capitals.

```
<response>
   ...
   <observation>1535^aB-pO2^</observation>
   <result>11</result>
   <responsetype>F</responsetype>
   <comment>Comment.Another comment.</comment>
   <observation>10269^S -ALAT^</observation>
   <result>61</result>
   <responsetype>F</responsetype>
</response>
```

Figure 12: The result of wrapping an HL7 message

ement of the output data is named `response` (beginning on line 2). The creation of `response` elements is controlled by the XW attributes of the element. The part of input used for creating this element is specified to begin with string `MSH` at the beginning of a line. The content of the part is divided into lines to be matched by sub-elements of `response` by specifying the end-of-line to be a `childseparator`. The value `unbounded` for attribute `maxoccurs` specifies that an arbitrary number of parts beginning with `MSH` are accepted and transformed to `response` elements.

The element `xw:CHOICE` (beginning on line 4) is used for describing alternative parts of the input. Its occurrence attributes are used to inform that the selection can be repeated an unlimited number of times (`xw:maxoccurs='unbounded'`) and that the content described by this element is optional (`xw:minoccurs='0'`). XW attribute `starter`, which is used for identifying the matching part, is mandatory for all descendant elements of `xw:CHOICE` that describe some initial fragment of the alternative parts of input; in this case these are the `xw:collapse` elements on lines 5–12 and 14–17 used for matching lines beginning with `OBX` or `NTE`.

In this example, the comment text on any adjacent `NTE` lines is grouped together in a `comment` element. This is done by introducing a new hierachy level around the `NTE` lines using `xw:ELEMENT` (lines 13 and 18). The `NTE` lines are processed on lines 14–17 with `xw:collapse`, which is replaced in the result by the results of its child elements. Out of these the `xw:ignore` element on line 15 is used for discarding three non-interesting (empty) subfields of a line, and the `xw:collapse` on line 16 is used for including the actual comment text from the fourth field of the line.

The specified wrapper will convert the sample input data into XML form shown in Fig. 12.

3.2 Automating the control of processing

Describing document conversions and transformations is often tedious even with powerful languages like XW or XSLT. The descriptions grow large if there are tens or hundreds of parts to identify. Fortunately the generation of these descriptions can be automated in some cases. As an example we are considering mass printing of data files consisting of invoices. Invoices are divided into lines that start with identifiers, and lines are divided into fields. There is a control file for this invoice data file that describes the names and the meaning of the identifiers and delimiters, as well as the formatting. A program for processing the data is written manually based on the control file. Because the control file already contains all the information needed to identify and to format the data, it would be practical to automate the generation of wrapper specifications and stylesheets from the control file.

In Fig. 13 we can see how processing of the data file can be fully automated. First we write an XW wrapper (1) for translating the control file (control.txt) to an XML form. This allows the control file to be processed with various XML technologies, for example XSLT. Next we write an XSLT script (2) which processes the XML version of the control file (control.xml) and generates an XW wrapper (3) for converting the data file (invoices.txt) to XML (invoices.xml). Third we write a formatting script (4) which produces a formatted version (invoices.fo) of the data file by retrieving for each of its elements the appropiate formatting instructions from the control file.

Once the first XW wrapper (1) and the two XSLT scripts (2 and 4) have been written, the conversion and the formatting are automated. After that it is possible to deal with a number of similar materials simply by replacing the control file.

As of this writing we have tested the initial part (scripts 1 and 2) of the process outlined above, succesfully generating an XW wrapper (3) that converts given invoice data to well-formed XML.

4. IMPLEMENTATION PRINCIPLES

In this section we discuss the implementation principles of XW. We have developed a prototype implementation of XW using Java. The implementation uses Apache Xerces [2] as an XML parser to read in the wrapper description. The wrapper description is represented internally as a tree structure called a *wrapper tree*. The wrapper tree is used as an automaton that guides the parsing and translation of the input data.

The wrapper tree consists of labeled nodes, each corresponding to an element of the wrapper description. The nodes are used for matching parts of input and for generating corresponding result elements. A node may match a number of parts of the input based on the occurrence indicator attributes `minoccurs` and `maxoccurs` given in the wrapper description.[2]

Parsing of delimiter-separated textual data is based on strings that XW looks for in the input data. We attach four sets of strings to each node of the wrapper tree, denoted by S (Starters), B (Begin strings), T (Terminators), and F (Follow strings). The design of these sets was inspired by the *First* and *Follow* sets used in LL(k) parsing [17]. The S set contains starter strings that are specified for an element in the wrapper description, and the T set contains the specified

[2]Generation of static text content is also supported by including fragments of literal text in a wrapper description. They are implemented by *text nodes*, which simply generate the corresponding text to the result whenever encountered during the interpretation of the wrapper tree. We ignore further treatment of text nodes in the following discussion of wrapper execution.

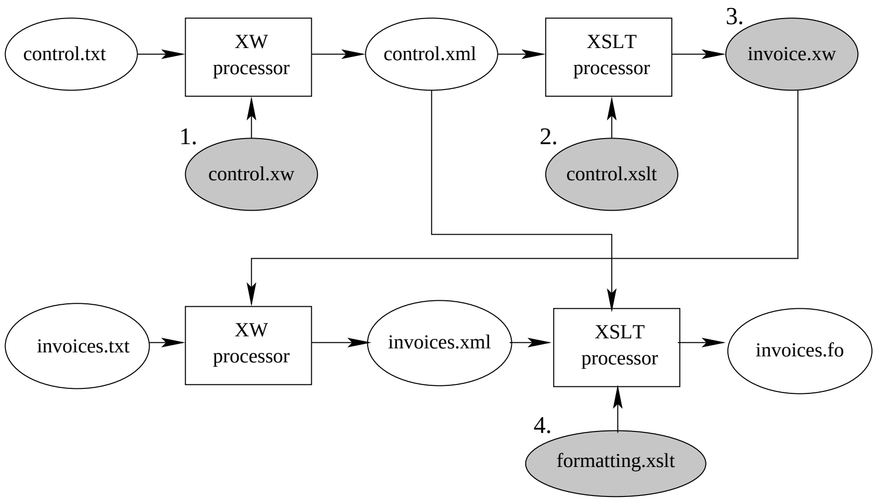

Figure 13: Automated conversion and formatting of invoices

terminator strings. Sets S and T are used for finding and skipping delimiters of input parts.[3] The B set for a node consists of strings that can begin a part of input matched by the node. The B sets are used for choosing between alternative branches of the wrapper tree based on initial fragments of input. Finally, the F set for a node consists of those strings that can begin a part of input that *follows* a part matched by the node.

The delimiter sets are computed with rather straightforward traversals of the wrapper tree. We omit a formal description of the preprocessing of the delimiter sets. Instead, we give an example which hopefully conveys the main ideas.

Consider the wrapping of simplified troff-like documents, which consist of an optional header followed by a number of paragraphs. The start of the header and the paragraphs are denoted by single-line commands `.H` and `.P`, respectively. Paragraphs consist of a number of lines. Translation of such an input to XML could be specified by an XW wrapper shown in Fig. 14. The wrapper tree created from this wrapper description is shown in Fig. 15. Occurrence indicators (`minoccurs`, `maxoccurs`) are shown in parentheses after the name of the node. The default value of both is 1; the value that denotes ”`unbounded`” is shown as ’*’.

The S sets contain the starters given explicitly in the wrapper specification (for `header` and `para`). If the S set is non-empty, then the B set contains exactly the same strings. Otherwise it contains the strings that can begin a part matched by the node: for `wrapper` and `docu` this means the starter `"\^.H\n"` of `header`, and, because `header` is optional, also the starter `"\^.P\n"` of `para`.

If no starters are specified for a node, its input part is assumed to start immediately after the previously matched

```
<xw:wrapper xmlns:xw="http://www.cs.uku.fi/XW/2001"
            xw:sourcetype="text" >
  <docu><header xw:starter="\^.H\n"
                xw:minoccurs="0" />
        <para xw:starter="\^.P\n"
              xw:minoccurs="0"
              xw:maxoccurs="unbounded">
            <line xw:terminator="\n"
                  xw:minoccurs="0"
                  xw:maxoccurs="unbounded" />
        </para>
  </docu>
</xw:wrapper>
```

Figure 14: XW wrapper for simplified `troff`

part of input (if any). This is achieved by including in the B set of such a node (e.g. `line`) only an empty string, which technically appears at any position of input (e.g., at the very beginning of a `para` and immediately after the end of a preceding `line`).

Element `line` has the only explicitly specified terminator `"\n"`, which is stored in the T set of the corresponding node.

The contents of the F (Follow) sets are motivated as follows. By default, the entire input stream is processed by the wrapper. For this reason an end-of-file indicator (EOF) is included in the F set of the root of the wrapper tree. Follow-delimiters are propagated to the F sets of any descendant nodes that may match some terminal fragment of the input: in this case from `wrapper` to `docu`, `para`, `line`, and `header`. Repeating `para` parts follow each other in sequence, which is why the B set of node `para` is included in its F set. A similar reasoning applies to node `line`. Because node `line` also matches terminal parts of paragraphs, its F set contains also the follow-strings of its parent node `para`. Since a `header` can be followed by `para` parts, its F set contains in addition the begin strings of node `para`.

[3] Sub-part delimiters specified using `childstarter` and `childseparator` are included in the S sets of the nodes that correspond to the sub-parts of the element, and similarly the `childterminators` are included in the T sets of these nodes.

A node is called *nullable* if it matches an empty part of input (without any delimiters or content). A node is nullable if its S and T sets are both empty and if all of its children are optional or nullable. In the example wrapper tree nodes `wrapper` and `docu` are nullable.

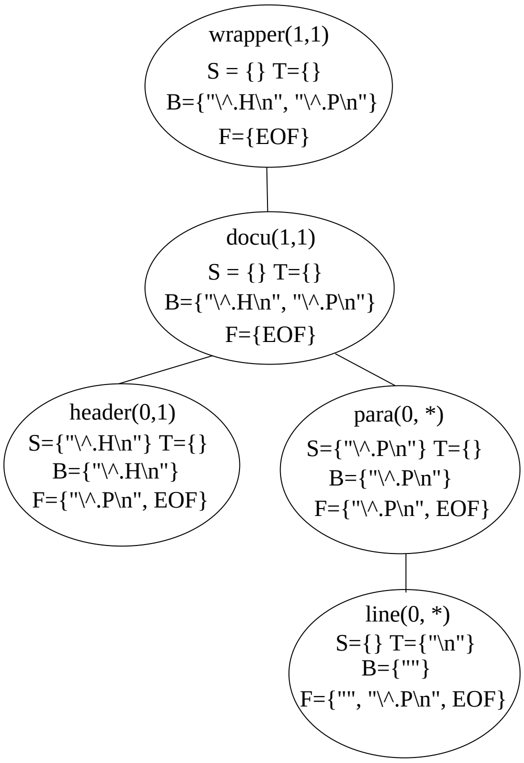

Figure 15: An XW wrapper tree

Execution of a wrapper is implemented as a traversal of the wrapper tree, which guides the parsing of the input. Fig. 16 shows a simplified version of the XW execution algorithm represented as a recursive procedure `processNode`. An XW transformation is initiated by passing the root of the wrapper tree as an argument to `processNode`.

Parsing of the input is based on two scanning functions. Function `scanUntil`(**in** D: set of strings; **out** s: string) advances the input cursor until the first occurrence of any delimiter string in D is found. When such an occurrence is found, the function assigns it to s and returns the scanned part of the input, excluding the terminating string s. In case of multiple strings occurring at the same input position, the longest match is selected and assigned to s. This is done to choose the most specific one out of conflicting delimiters. For example, in the execution of the wrapper tree in Fig. 15 this rule would lead to selecting the starter "`.P`" of a new paragraph instead of an empty-string starter of a new line. Function `scanOver`(**in** D: set of strings) similarly advances the input cursor past the first occurrence of any string in D and returns the part of input that precedes the occurrence. Function `scanOver` is used in the XW execution algorithm for skipping starters (line 10) and terminators (line 18). If none of the given delimiters is found, the function raises an

```
procedure processNode(Node n):
 1. for r:=1 to n.maxoccurs do
 2.     if r ≤ n.minoccurs and n is not nullable then
 3.         void:= scanUntil(n.B, s);
 4.     else void:= scanUntil(n.B ∪ n.F, s);
 5.     if r > n.minoccurs and s ∉ n.B then return;
 6.     if n is xw:CHOICE then
 7.         Let c be the first child of n s.t. s ∈ c.B
 8.         processNode(c);
 9.     else
10.         if n.S ≠ ∅ then void:= scanOver(n.S);
11.         if n is xw:ELEMENT or a result element then
12.             startElement(...);
13.         if n has children then
14.             c := n.firstChild();
15.             while c ≠ null do
16.                 processNode(c);
17.                 c := c.nextSibling();
18.         if n.T ≠ ∅ then content:= scanOver(n.T);
19.         else content:= scanUntil(n.F, s);
20.         if n is a leaf, other than xw:ignore then
21.             characters(content);
22.         if n is xw:ELEMENT or a result element then
23.             endElement(...);
24. endfor
```

Figure 16: XW wrapper tree execution algorithm

error.

The execution algorithm may iterate at each node based on its occurrence indicators. Required occurrences of non-nullable nodes are processed by first scanning the input for a string that begins the input part of the node (lines 2–3). Optional occurrences of a node are processed until a follow-delimiter is found instead of a begin-delimiter for the node (lines 4–5). The result of the transformation is formed by generating SAX events for the start, for the text content, and for the end of an element (lines 12, 21 and 23).

It is relatively easy to see that the wrapper tree execution algorithm runs in linear time with respect to the length of the input; each character needs to be inspected only once (and possibly reported as the content of the result document).[4]

Wrapping of positionally defined textual fields or binary data is different but also simpler than parsing of delimiter-separated textual input, which we discussed above. For reasons of space we omit the discussion of translation principles of these cases.

The execution of a wrapper, as discussed above, scans some strings in the input and writes some other strings to the result on the basis of the current node in the wrapper tree traversal. So, basically, XW wrapping corresponds to the execution of a *finite state transducer*. (See, e.g., [15, Chap. 2].) That is, XW is capable of performing translations from a

[4]The scanning functions can be made to inspect each input character only once by implementing their argument sets as Aho-Corasick automata [1].

regular language to another regular language.[5] We could also extend XW to enable the recognition and translation of recursive or self-similar nesting input structures. Representations of nested objects or lists are examples of such structures. This extension would make XW wrapping to correspond to translation of LL(1) languages. We haven't considered this extension in depth; it would complicate the implementation, while it is not clear to us whether it is really needed for XML wrapping of real-life data representations.

5. RELATED WORK

Extraction of data from web pages has been studied extensively. The irregularity of web pages often requires rather complex methods for wrapping data. These include, for example, heuristics for identifying relevant tokens, sections and objects on HTML pages [3, 11], and machine learning of wrappers [10]. An XML-based Data Extraction Language (DEL) has also been proposed [12]. DEL uses an XML syntax to express a procedural sequence of commands for accessing a text source with regular expressions and for building a DOM representation of the extracted data. As opposed to these developments we are dealing with an easier problem. Data formats that we consider are quite regular when compared to typical HTML pages, which makes a declarative and simple language like XW sufficient to describe wrappers.

Extracting and loading relational data as XML documents has been considered by Bourret, Bornhövd and Buchman [4]. They use object-relational mapping techniques to develop a utility to transfer data between relational databases and XML documents. Their work is somewhat related to ours. XW is not tied to any specific application area or input data format, but it can be used, for example, for translating comma-separated database reports to XML. (Translation of XML documents to a format used for loading data to a database is normally easy with standard XML processing techniques.)

The work of Nakhimovsky [13] is rather close to ours in the sense that he also applies a declarative formalism for specifying the translation of data to XML. He applies formal grammars and parser generators, while we apply declarative XW wrappers. On the other hand, generating XML with a parser generator requires hand-coding of the relevant semantic actions, which requires considerable skill. In contrast, the XW implementation automates the generation of the resulting XML document described by the wrapper.

6. CONCLUSIONS

We have described a declarative XML-based language called XW for describing translation of non-XML data sources to XML documents, or "XML wrapping". XW is suitable for wrapping any serialized data consisting of recognizable parts like lines or records and fields. The described wrappers process the input data sequentially, recognizing parts and arranging them as an XML document described by the wrapper. Since the language is not designed for any specific application domain it should be easily applicable for wrapping a wide variety of legacy data formats.

[5] The result of the translation is just chosen to be a string of SAX events that encodes a hierarchical XML structure.

We have developed a working prototype of XW, and are using it to automate the conversion of medical messages and mass printing data to XML documents. We also discussed possibilities of applying the wrapping technology to control files of mass printing data in order to fully automate their processing.

We also discussed the algorithmic principles of executing XW wrapper descriptions. We are considering various extensions to the basic features of XW discussed in this paper. Possible extensions include XSLT-like constructs for generating XML attributes using values drawn from the input data, and methods to change the order of parts of input. Such extensions should not weaken the declarative simplicity and efficient implementability of XW. On the other hand, they might eliminate the need of further document transformations in some typical wrapping applications, and thus both simplify the design and speed up the execution of XML wrappers.

7. ACKNOWLEDGMENTS

This work is supported by the Finnish National Technology Agency with funds provided by the Europeand Union, and the following organizations: Deio Corporation, Enfo Group Plc, JSOP Interactive, Kuopio University Hospital, Medigroup Ltd, SysOpen Plc, and TietoEnator Corporation.

We dedicate this paper to the memory of Prof. Eila Kuikka, the original leader of our project. We also acknowledge the support of Prof. Martti Penttonen and the collaboration of M.Sc. Sami Komulainen and Ph.Lic. Paula Leinonen.

8. REFERENCES

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Comm. of the ACM*, 18(6):333–340, June 1975.

[2] Apache Software Foundation. *Xerces2 Java Parser*, 2002. `http://xml.apache.org/xerces2-j/index.html`.

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

[4] R. Bourret, C. Bornhövd, and A. Buchmann. A generic load/extract utility for data transfer between XML documents and relational databases. In P. You, editor, *Proc. of the 2nd Intl. Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems*, pages 135–143. IEEE Comp. Soc., 2000.

[5] J. Clark, editor. *XSL Transformations (XSLT) Version 1.0.* W3C Recommendation, Nov. 1999.

[6] M. Ek, H. Hakkarainen, P. Kilpeläinen, E. Kuikka, and T. Penttinen. Describing XML wrappers for information integration. In *Proc. of XML Finland 2001*, pages 38–51, Tampere, Finland, Nov. 2001.

[7] Electronic business XML (ebXML) home page. `http://www.ebxml.org`, 2002.

[8] Health Level Seven. *HL7 Standards*, 2002. `http://www.hl7.org/Library/standards.cfm`.

[9] A. L. Hors et al., editors. *Document Object Model (DOM) Level 2 Core Specification*. W3C Recommendation, Nov. 2000.

[10] N. Kushmerick. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence*, 118(1-2):15–68, 2000.

[11] A. Laender, B. Ribeiro-Neto, and A. da Silva. DEByE – data extraction by example. *Data & Knowledge Engineering*, 40:121–154, 2002.

[12] E. Lampinen and H. Saarikoski, editors. *DEL - Data Extraction Language*. W3C Note, Oct. 2001.

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

[14] XML support in Oracle9i. `http://otn.oracle.com/tech/xml/techinfo.html`, Dec. 2000. An Oracle Technical Whitepaper.

[15] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*, volume 1 Word, language, grammar. Springer-Verlag, 1997.

[16] SAX 2.0: The Simple API for XML. `http://www.megginson.com/SAX/sax.html`, May 2000.

[17] S. Sippu and E. Soisalon-Soininen. *Parsing Theory*, volume I: Languages and Parsing. Springer-Verlag, 1988.

[18] H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn, editors. *XML Schema Part 1: Structures*. W3C Recommendation, May 2001.

[19] D. Waldt. Getting data into XML: Data collection and conversion techniques. In *XML Europe 2002*, Barcelona, May 2002. Abstract.