Proc. Estonian Acad. Sci. Eng., 2006, **12**, 1, 3–15
https://doi.org/10.3176/eng.2006.1.01

Automating the XML conversion and XSL formatting of textual legacy data

Heli Tervo, Pekka Kilpeläinen and Tommi Penttinen

Department of Computer Science, University of Kuopio, P.O. Box 1627, FI-70211 Kuopio, Finland; {heli.tervo, pekka.kilpelainen, tommi.penttinen}@cs.uku.fi

Received 4 August 2005, in revised form 27 October 2005

Abstract. XML technology supports multichannel publishing of data. Migration from legacy workflows to the use of XML technology requires not only the data to be converted to XML, the formatting processes need to be reimplemented, too. We address the problem of converting legacy data together with their formatting specifications to XML and XML stylesheets. We discuss a case study, based on simplified line-based data from a real production environment. We introduce a prototype architecture to automate the conversion and formatting of such data, based on its legacy formatting specifications. The prototype was implemented using freely available XML tools and a declarative XML conversion language called XW. The simplicity of XW makes it possible to control the somewhat complex processes by three relatively simple scripts.

Key words: XML, multichannel publishing, legacy data, stylesheet, automated conversion, XSL.

1. INTRODUCTION

XML techniques support multichannel publishing; different forms of the result for paper printing and digital media can be produced from a single XML document. Unfortunately, a lot of non-XML data is published using specialized techniques in a specific target form. Mass printing of phone bills is an example of such, often very efficient, activity. Multitargeting this kind of legacy data for different publication media is a challenge.

Applying XML technology to the publishing of legacy data requires the data first to be converted to XML. There are methods for this conversion [1], but the corresponding conversion of formatting instructions or style sheets has barely been examined. For example, XSL [2] is a powerful formatting language of XML

documents, but describing the processing of tens or hundreds of different element types requires manual coding of equally many formatting rules. If exact formatting specifications for the legacy data are available, it would be advantageous to avoid this tedious task by automating also the conversion of "legacy stylesheets" to corresponding XML stylesheets (XSL).

As an example of using XML techniques for automated formatting of legacy data, we examine a conversion of line-based text data to XML, and further to formatted result forms. We introduce a technique and a supporting tool set that automatically converts and formats the given legacy data with the help of an existing control file, which describes the structure and formatting of the data. The architecture of this tool set is based on an XML conversion language called XW (XML Wrapping) [3,4], which was developed in our research project. XW is a simple, yet powerful, declarative XML wrapper specification language. Using freely available XML tools and XW we developed this architecture that automates the generation of style sheets from legacy data and its control data. Section 2 introduces our example data and Section 3 presents the architecture. We report experience of using XML techniques in our prototype implementation in Section 4.

2. EXAMPLE DATA

We consider the possibility of automating the formatting of line-based textual data. As a sample scenario we consider simplified data, which is taken from a real printing environment [5]. In the scenario the data itself is given in one file and a control file specifies the positioning and formatting of the actual data fields.

In our experiment with five companies we found that they all dealt with line-based text data with fields. There were data with more or less exact schemas and formatting rules, but also data having no instructions at all, or only with informal instructions like notes for humans. Obviously, to automate the formatting task, instructions have to be in a format understandable to computers. From these legacy stylesheets and instructions we formed a simplified sample legacy stylesheet for our example data.

As an example we consider line-based text data of phone invoices. The data consists of invoices, each having three parts (Fig. 1): identifier data (A), specification data (B) and payment data (C). The payment data, for example, contains payer information, a reference number, the due date and the total sum.* The *row identifier* (A1, A2 etc.) at the beginning of each row indicates the content and the meaning of the row. A vertical bar "|" is used as a separator of data fields within one row.

* Real invoice data includes a greater amount of additional detailed information.

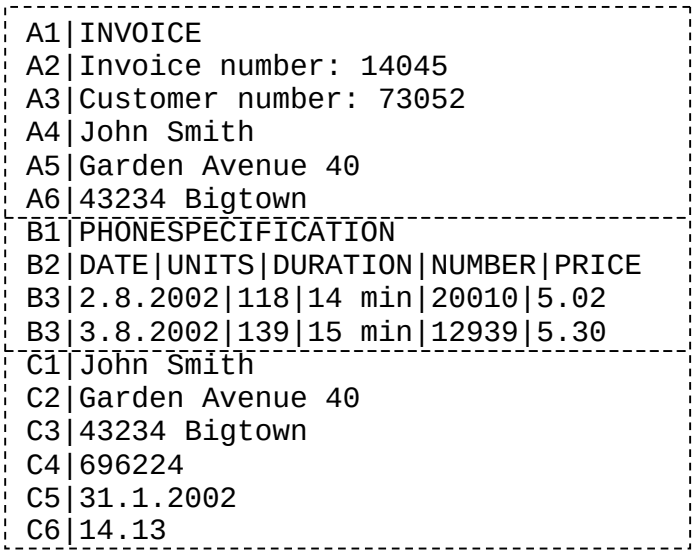


Fig. 1. An example of the phone invoice data.

The *control (data) file* (Fig. 2) describes the formatting and the structure of the actual data. The control file contains parts of the identifier data, the specification data and the payment data, respectively. Each row of the control file specifies the name and formatting properties, like font family, point size and coordinates, of the corresponding data element.

```
A1|header INVOICE|Helvetica|12|76|65
A2|invoice number|Helvetica|10|432|65
A3|customer number|Helvetica|10|432|80
...
B1|header PHONESPECIFICATION|headcell1
B2|headrow|headcell1|headcell2...
B3|specrow call|cell1|cell2|cell3|cell4|cell5
...
Cells of specification:
headcell1|Helvetica|12|40|left
headcell2|Helvetica|12|120|right
cell1|Helvetica|10|40|left
cell2|Helvetica|10|120|right
```

Fig. 2. Part of a control file for the phone invoice data.

3. AN AUTOMATED CONVERSION AND FORMATTING ARCHITECTURE

Applying XML technology to the publishing of legacy data requires the data first to be converted to XML. In a real-life case the XML wrapper for the invoice data as well as the formatting script may consist of hundreds or even thousands of rows. It would be advantageous to avoid the tedious task of writing them by generating conversion and formatting scripts automatically. We show that with the help of existing specifications this may be possible.

Next we describe how we were able to automate the XML conversion and the formatting of the invoice data. The control file has a central role in the process since it specifies the structure of the data, names of fields and the formatting of the data. Figure 3 illustrates the automated formatting process. First, the control file (1) is converted to XML (3). This allows the actual wrapper of the XML conversion (5) for the invoice data (6) to be generated directly from the XML control file with XSLT. With this resulting wrapper we can then convert the invoice data to XML (7). The formatting is done with a generic formatting script (8). This formatting script can be used with various data because the implementation of formatting objects is guided by the data-specific control file (3). The resulting invoice data in XSL-FO format (9) is then ready to be formatted to PDF or PostScript, for example.

Next we describe various phases of the conversion and formatting process in greater detail.

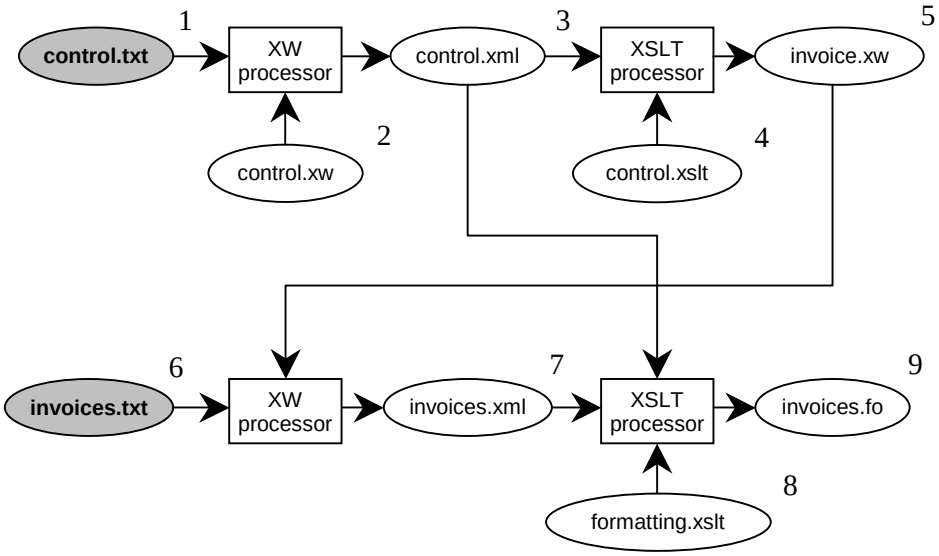


Fig. 3. Automatization of the conversion and formatting.

3.1. XML conversion of the control file

The automatization of the formatting process is based on an XML conversion of the control file. We describe this rather straightforward process using an XML wrapper description language called XW [3,4]. XW is a declarative, lightweight language for translating legacy data to XML. Figure 4 represents an XW wrapper for converting the control file to XML. The result of this conversion is shown in Fig. 5.

An XW wrapper specification is syntactically a well-formed XML document. The wrapper specification defines a template for the input document and describes also the structure of the output document and how input data is arranged into its elements. Sequential parts in the input data are described by sequential output

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xw:wrapper xmlns:xw="http://www.cs.uku.fi/XW/2001" ...>
<controlfile>
   <identifierdata xw:childterminator="\n">
      <row xw:starter="\^A"...>
      <identifier>A<xw:collapse/></identifier>
      <name xw:childseparator="\s">
         <xw:collapse xw:maxoccurs="unbounded">
            <xw:collapse/>_
         </xw:collapse>
      </name>
      <font/> <size/> <x/> <y/>
      </row>
   </identifierdata>
   <specification xw:childterminator="\n">
      <specificationrow xw:starter="\^B"...>
      <identifier>B<xw:collapse/></identifier>
      <name xw:childseparator="\s">
         <xw:collapse xw:maxoccurs="unbounded">
            <xw:collapse/>_
         </xw:collapse>
      </name>
      <cell xw:maxoccurs="unbounded"/>
      </specificationrow>
   </specification>
   <paymentdata xw:childterminator="\n">
      <row xw:starter="\^C"...>
      ...
      </row>
   </paymentdata>
   <spec_cells xw:starter="\^Cells of specification:\n"...>
      <cell xw:maxoccurs="unbounded" xw:childseparator="|">
      <name/> <font/> <size/> <x/> <a/>
      </cell>
   </spec_cells>
</controlfile>
</xw:wrapper>
```

Fig. 4. An XW wrapper for converting the control file to XML.

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <controlfile>
3     <identifierdata>
4        <row>
5        <identifier>A1</identifier>
6        <name>header_INVOICE_</name>
7        <font>Helvetica</font> <size>12</size>
8        <x>76</x> <y>65</y>
9        </row>
10       ...
11    </identifierdata>
12    <specification>
13       <specificationrow>
14       <identifier>B3</identifier>
15       <name>specrow_call_</name>
16       <cell>cell1</cell> <cell>cell2</cell>
17       <cell>cell3</cell> <cell>cell4</cell>
18       <cell>cell5</cell>
19       </specificationrow>
20       ...
21    </specification>
22    ...
23    <spec_cells>
24       <cell>
25       <name>headcell1</name>
26       <font>Helvetica</font> <size>12</size>
27       <x>40</x> <a>left</a>
28       </cell>
29       ...
30    </spec_cells>
31 </controlfile>
```

Fig. 5. The control file translated to XML.

elements in the wrapper specification (Fig. 4: the identifier data part (lines 4–14), the specification data part (15–25), the payment data part (26–30) and the description of specification rows (31–35)). Subparts of a part are described by child elements of the corresponding element. For example, a `row` consists of parts like `identifier` and `name` (Fig. 4, lines 5–13). Elements outside the XW namespace like `row` and `name` produce elements to the resulting XML file. Content characters in wrapper elements (for example, "_" on line 9 of Fig. 4) produce corresponding characters in the resulting XML elements. Elements and attributes belonging to the XW namespace are instructions for processing the input data.

Attributes `xw:starter` and `xw:terminator` identify a starting or terminating string of the corresponding part in the input document. For example, a row in the identifier data part is recognized from the string "A" at the beginning of a line (Fig. 4, line 5). Alternatively, the start or the end of a part could be described in the parent element with attributes `xw:childstarter` and

`xw:childterminator`. For instance, the end-of-line character terminates the subparts of the identifier data part (Fig. 4, line 4). Repetitions are described as in XML Schema [6] with attributes `xw:minoccurs` and `xw:maxoccurs`. An element `xw:collapse` produces the content of a corresponding part to the result, without producing a corresponding element. With `xw:collapse` we generate (Fig. 4, lines 7–11), for example, a content to the resulting element `name` (Fig. 5, line 6) from the name field of the input file (Fig. 2, line 1). The generated names like `header_INVOICE_` will be used as element names in the XML version of the invoice data file. The translation of the data file to XML with the help of the XML control file is discussed in the next section.

3.2. Automatization of the wrapper specification for the legacy data

The invoice data has to be converted to XML as well if we want to process it with XML techniques. The resulting invoice data in XML is presented in Fig. 6. A simple way to do the conversion is to use the XW language. An XW wrapper for converting the invoice data to XML is presented in Fig. 7. For example, the first row of the invoice data (Fig. 1)

```
A1|INVOICE
```

is converted to the form

```
<header_INVOICE_>INVOICE</header_INVOICE_>.
```

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <data>
3     <invoice>
4        <identifierdata>
5        <header_INVOICE_>INVOICE</header_INVOICE_>
6        <invoicenumber_>invoicenumber: 14045</invoicenumber_>
7        <customernumber_>customernumber: 73052</customernumber_>
8        ...
9        </identifierdata>
10       <specification>
11       ...
12       <specification_phone_>
13          <cell1>2.8.2002</cell1> <cell2>118</cell2>
14          <cell3>14 min</cell3> <cell4>20010</cell4>
15          <cell5>5.02</cell5>
16       </specification_phone_>
17       ...
18       </specification>
19       <paymentdata>
20       <payer_name_>John Smith</payer_name_>
21       <payer_streetaddress_>Garden Avenue 40</payer_streetaddress_>
22       ...
23       </paymentdata>
24    </invoice>
25 </data>
```

Fig. 6. Phone invoice data translated to XML.

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <xw:wrapper xmlns:xw="http://www.cs.uku.fi/XW/2001"
3        xw:inputencoding="ISO-8859-1" xw:outputencoding="ISO-8859-1"
4        xw:sourcetype="text">
5     <data>
6        <invoice xw:maxoccurs="unbounded">
7           <identifierdata>
8              <xw:collapse xw:starter="\^A1|" xw:childseparator="|">
9                 <header_INVOICE_/>
10             </xw:collapse>
11             ...
12          </identifierdata>
13          ...
14       </invoice>
15    </data>
16 </xw:wrapper>
```

Fig. 7. A part of an XW wrapper for translating invoice data to XML.

For identifying and converting the current row we use the row identifier "A1" in the wrapper of conversion (Fig. 7, line 8), and the element name header_INVOICE_ to create and name the element (line 9). Other rows in the invoice data are identified and converted to XML in the same way, only changing the row identifier and the name of the element to be created.

Invoice data may have a large amount of specified information. That makes it laborious to write the specification of the conversion (for example, an XW wrapper). Writing the specification is not necessarily complicated, but requires often a lot of careful routine work. The conversion could be automated using the XML control file as a parameter file. All the elements to be created can be generated in a uniform way if variable data – row identifiers and names of the XML elements – are retrieved from the parameter file.

The control file includes all information, required for converting the invoice data to XML. Therefore an XW wrapper for the invoice data can be generated directly from the control file after the control file is converted to XML. This phase was implemented with XSLT [7], which is a convenient transformation language for XML documents. We were able to generate the conversion scripts with XSLT rather easily, because of the XML syntax and the declarative nature of the XW language.

The XML control file is an input file to the XSLT script (4 in Fig. 3). The script creates an XW wrapper specification for the invoice data. The example of the XW script considered above (Fig. 7, lines 7–12) is generated with XSLT as follows (see also Fig. 5, the XML control file):

```
<xsl:template match="row">
<xw:collapse xw:starter="\^{identifier}|"
             xw:childseparator="|">
   <xsl:element name="{name}"/>
</xw:collapse>
</xsl:template>
```

Every `row` element from identifier data and payment data in the XML control file is processed with the same XSLT template. In the control file the content of the child element `identifier` of the `row` element (Fig. 5, line 5) is selected for the value of the attribute `xw:starter` by the template. The child element `name` of the `row` element (Fig. 5, line 6) generates a name for the element to be created.

This way all the information required to create the wrapper is received from the control file and the wrapper can be generated automatically for the whole invoice data. Then the resulting XW wrapper (Fig. 7) converts the actual invoice data to XML (Fig. 6). By writing an XW wrapper for the control file we were able to generate the XW wrapper for invoice data automatically with two simple conversions.

3.3. Automating XSL formatting

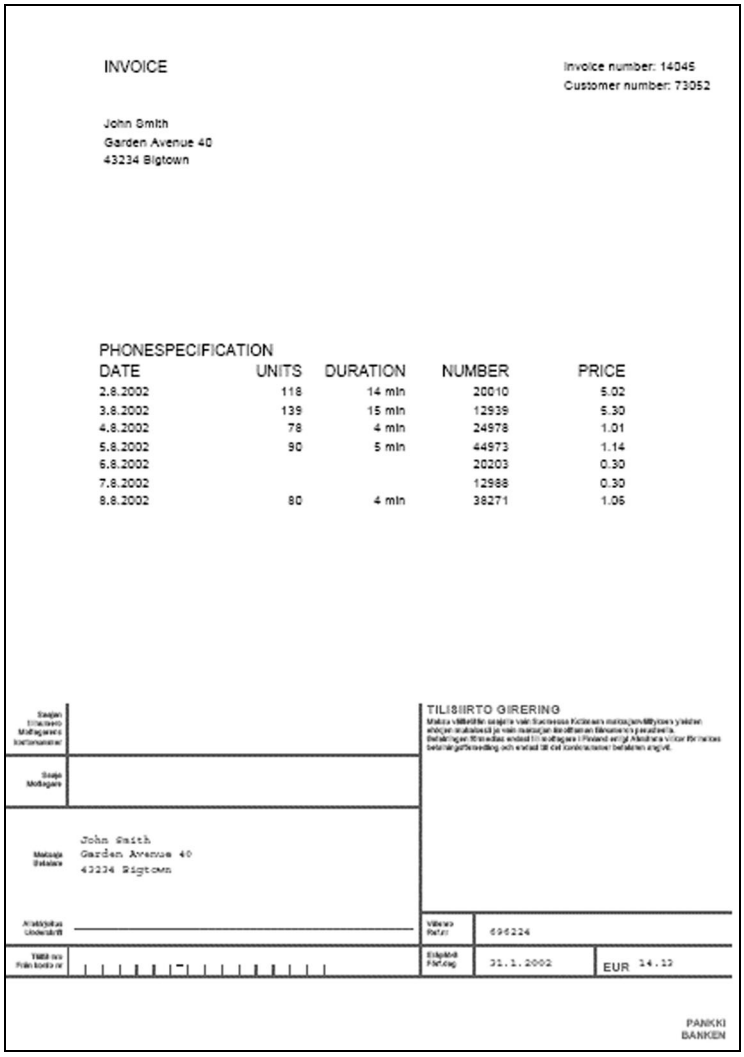
After the invoice data is converted to XML, we can use XML techniques for formatting the data. We wrote an XSLT script (8 in Fig. 3) for the invoice data, which produces the data in XSL-FO [2] format. This format includes formatting objects, created from the specifications given in the control file.

The XML invoice data is input for the formatting script. The script implements the formatting as follows: it gets the formatting information of elements (font, point size and coordinates) by element names from the XML control file (3 in Fig. 3), and uses them to generate appropriate XSL formatting objects. For example, the XSL formatting of child elements of the identifier data is described in Fig. 8.

We need to be able to place the data of single elements to the result file arbitrarily. For this arrangement we used the XSL `block-container` formatting object. To this object we can give the position of the block as coordinates on the page (Fig. 8, lines 10–11). Stacking of specification rows and placing their

```
 1   <xsl:template match="identifierdata/*">
 2      <xsl:variable name="elementname" select="name()"/>
 3      <xsl:variable name="row" select=
 4         "document('control.xml')/controldata/
 5         identifierdata/row[name=$elementname]"/>
 6      <xsl:variable name="font" select="$row/font"/>
 7      <xsl:variable name="size" select="$row/size"/>
 8      <xsl:variable name="x" select="$row/x"/>
 9      <xsl:variable name="y" select="$row/y"/>
10      <fo:block-container height="1cm" width="20cm" top="{$y}pt"
11         left="{$x}pt" position="absolute">
12         <fo:block font-family="{$font}" font-size="{$size}pt">
13            <xsl:apply-templates/>
14         </fo:block>
15      </fo:block-container>
16   </xsl:template>
```

Fig. 8. Formatting of the identifier data according to the control file.



INVOICE

Invoice number: 14045
Customer number: 73052

John Smith
Garden Avenue 40
43234 Bigtown

PHONESPECIFICATION

DATE	UNITS	DURATION	NUMBER	PRICE
2.8.2002	118	14 min	20010	5.02
3.8.2002	139	15 min	12939	5.30
4.8.2002	78	4 min	24978	1.01
5.8.2002	90	5 min	44973	1.14
6.8.2002			20203	0.30
7.8.2002			12988	0.30
8.8.2002	80	4 min	38271	1.06

TILISIIRTO GIRERING

John Smith
Garden Avenue 40
43234 Bigtown

696224

31.1.2002

EUR 14.13

PANKKI
BANKEN

Fig. 9. An example of invoice data, printed from a PDF document.

contents on fields of given widths required some ingenuity, but we were able to implement their layout as individual XSL tables. More details are given in [5].

The final result of the formatting is shown in Fig. 9.

4. EVALUATION

We performed a small experiment and measured the time taken by the transformation and formatting of sample invoices. The control file had fixed size and it was processed only once for each class of input documents – all the

invoices in this case. Its processing time was therefore excluded from these measurements as not significant.

The experiment was carried out on a Sun Fire 280R server equipped with a 750 MHz processor and SunOS 5.8 operating system running Java J2SE v1.4.1. The XSLT processors used were Xalan [8] and Saxon [9], and for PDF-formatting we employed Apache FOP [10]. Used processor time was measured with the Unix `time` command. The plain-text source document consisted of one or more copies of a typical 456-byte long invoice. Two parts of the conversion, from plain-text invoices into XML, with XW, and then into PDF, were timed separately. Furthermore, two different ways of performing XML-to-PDF conversion were compared: 1) the conversion is done in one step with FOP and its built-in XSLT processor Xalan, 2) in two steps using Saxon for the XSLT transformation and FOP for PDF-formatting. The results are shown in Fig. 10.

According to the results, XW conversion to XML takes only a fraction of the time taken by formatting, most of which is spent on FO-to-PDF formatting. XSLT conversion from XML to FO takes a relatively small part of the total time. Differences between the two XSLT processors were not significant.

The results are encouraging. With 80 invoices, one invoice takes only about half a second to process and the per-invoice time appears to shrink with bigger volume. With these figures, already this architecture would allow processing of large amounts of invoices per day, around 173 000. Furthermore, the computer system used here is, by contemporary standards, of rather limited capacity. This promises even greater capacity for modern equipment and realizes the potential of this technology for large-scale delivery of digital invoices. Also, the programs FOP and Saxon, used for our test formatting, are freely available and it is conceivable that high-end commercial tools could be more efficient.

The end goal of automatic transformations and formatting is the ability to process different types of input documents with the same transformation scripts, independently from the contents of those documents and without the need to

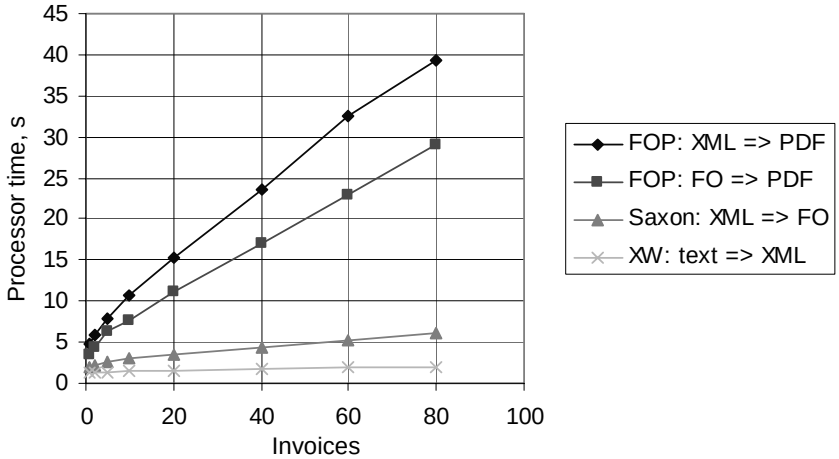


Fig. 10. Transformation and formatting times of the invoice data.

tailor the scripts to any particular type of input. The transformation scripts presented earlier achieved, with some restrictions described below, this content-independence. Such independence from the content, however, is not without its quirks. All the text that is to find its way into the result must be included in the elements. The resulting elements, such as `<invoicenumber>Invoice number: 14045</invoicenumber>`, are a bit clumsy. However, this content-independence enables the reuse of transformation scripts for countless different types of input documents.

The implementation of the architecture was also a practical test for the applicability of XW. The needed transformations are based on the declarative simplicity of XW that supports metaprogramming; automatic generation of scripts is possible with such a high-level description language.

This prototype architecture was implemented without considering complicated formatting issues like pagination or tables. Using positioning attributes brought us, however, the possibility to format tables, which in our example meant the opportunity of defining layouts like in Fig. 9. A relatively simple case study showed that it is possible to carry formatting out automatically under realistic assumptions. The main requirement was that the data and the control data have to be given in a line-based textual form. In this case the architecture works automatically guided by two short conversion scripts, written for the data in question, and a formatting script.

The scripts could be used as such for different data, too, when the data and the control data are given in a similar form. As an example, we could imagine data of various invoices, not only phone invoices but also electricity invoices or invoices for magazines, for example. When the scripts are defined for one invoice data, the whole architecture can be used for related but different data. The architecture should be relatively easy to adapt to rather dissimilar data, too, with slight rewriting of the conversion scripts; defining XML conversions with a declarative language such as XW does not require much work.

In simplest cases, the formatting script can be made totally content-independent in the sense that all data fields are treated identically, with their specific formatting properties retrieved from the control file. In our case study we had to tailor the formatting script slightly for the invoice material at hand. This tailoring included the treatment of the rows of the specification data (shown in the middle of Fig. 9) as XSL FO tables, and the specification of the boilerplate of the invoice form. Such tailoring is often needed in practice, but adapting the formatting script to different situations should not be a major task either.

The architecture of automatic transformation and formatting of legacy data covers the need of tailoring several tedious processing scripts for legacy data, for which there exists a legacy stylesheet. When both the legacy data and legacy stylesheet have all the information needed to process the data, the conversion and formatting can be done with rather mechanical modifications. Controlled by three relatively short scripts, the whole transformation and formatting pipeline works automatically.

ACKNOWLEDGEMENTS

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

REFERENCES

1. Waldt, D. Getting data into XML: Data collection and conversion techniques. Abstract. In *Proc. of XML Europe*. Barcelona, 2002.
2. Adler, S. et al. (eds.). Extensible Stylesheet Language (XSL) Version 1.0. W3C Rec., 2001, http//:www.w3.org/TR/xsl/
3. Ek, M., Hakkarainen, H., Kilpeläinen, P., Kuikka, E. and Penttinen, T. Describing XML wrappers for information integration. In *Proc. of XML Finland 2001* (Löppönen, J.-M., ed.). Tampere, 2001, 38–51.
4. Ek, M., Hakkarainen, H., Kilpeläinen, P. and Penttinen, T. Declarative XML wrapping of data. Report A/2002/2. University of Kuopio, Dept. of Comp. Sci. and Appl. Math., Kuopio, 2002.
5. Tervo, H., Kilpeläinen, P. and Penttinen, T. Automating the XML conversion and XSL formatting of textual legacy data. In *Proc. of Symposium on Programming Languages and Software Tools 2005* (Vene, V. and Meriste, M., eds.). University of Tartu, Dept. of Comp. Sci., Tartu, 2005, 191–205.
6. Thompson, N., Beech, D., Maloney, M. and Mendelsohn, N. (eds.). XML Schema Part 1: Structures. W3C Rec., 2001. http://www.w3.org/TR/xmlschema-1/
7. Clark, J. (ed.). XSL Transformations (XSLT), Version 1.0. W3C Rec., 1999. http://www.w3.org/TR/xslt
8. Apache Xalan Version 2.4.1, 2001. http://xml.apache.org/xalan-j/index.html
9. Saxon Version 6.5.3, 2001. http://saxon.sourceforge.net/saxon6.5.3/index.html
10. Apache FOP Version 0.20.5, 2002. http://xml.apache.org/fop/index.html

Tekstiliste pärandandmete XML-teisenduse ja XSL-vormingu automatiseerimine

Heli Tervo, Pekka Kilpeläinen ja Tommi Penttinen

XML-tehnoloogia toetab andmete mitmekanalilist avaldamist. Üleminek pärandsüsteemi töövoost XML-tehnoloogia rakendamisele ei eelda ainult andmete teisendamist XML-kohasteks, vaid ka vorminguprotsessi uut realisatsiooni. Artiklis on uuritud pärandandmete ja vormingukirjelduse ühist teisendamist XML-i ja XML-laaditabelisse. Probleemi on analüüsitud nii reaalsest tootmiskeskkonnast saadud kui ka lihtsustatud reastruktuuriga andmete näitel. Pärandandmete vormingukirjelduse alusel andmete teisendamise ja vormingu automatiseerimiseks on loodud prototüüparhitektuur. Viimane on realiseeritud XML-vabavara ja XML-teisenduskeele XW abil. Keele XW lihtsus võimaldab juhtida kohati keerulisi protsesse kolme suhteliselt lihtsa skripti abil.