◊ ◊ ◊

From database to presentation via XML, XSLT and ConTEXt

Berend de Boer

Introduction

Much data exists only in databases. A familiar example is an address list. Every once in a while this data must be presented to humans. To continue with the address list example, annually an address list must be printed and mailed.

In this article I attempt to given an exhaustive overview of going from structured data through ConTEXt to output, see figure 1.

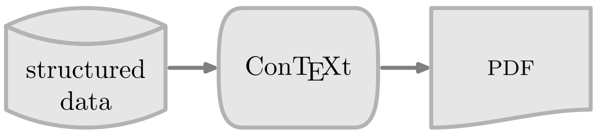

Figure 1 Going from data through ConTEXt to output

As any data format can be represented by XML, this document focuses on typesetting data in XML in ConTEXt, see figure 2. When the data is in XML, in can be directly handled by ConTEXt. ConTEXt has a built-in XML typesetting engine that can handle XML tags just fine. You don't have to convert the XML to ConTEXt macro's first. This is the subject of the following section.

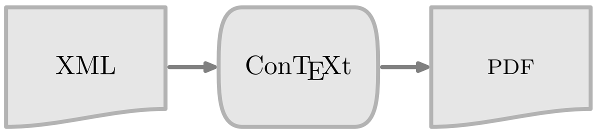

Figure 2 Going from xml through ConTEXt to output

When the data is not yet in XML format, is has to be converted to XML. 'Converting comma . . .' covers converting comma separated data to XML. 'Converting relational

. . .' covers converting data residing in relational databases such as DB/2 and InterBase to XML. 'Typesetting sql . . .' covers going from such data straight to ConTEXt without converting to XML first.

The XML data you have might not be easy to typeset. An advantage of XML is that it is easy to transform into XML with a different format. There is a specific language, XSLT, to transform XML into XML, see figure 3. This is the subject of 'Transforming XML ...'.

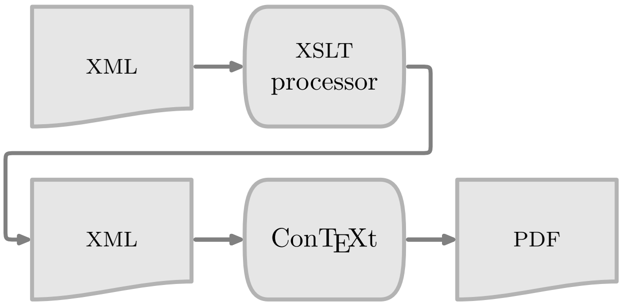

FIGURE 3 GOING FROM XML THROUGH CONTEXT TO OUTPUT

TYPESETTING XML IN CONTEXT

This section assumes that the data to be typeset is already available in XML. The next sections cover converting data to XML.

For this article a special XML format was chosen:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE rows SYSTEM "example.dtd">
<rows>
  <row>
    <field>Re-introduction of Type 3 fonts into the TeX world</field>
    <field>Wlodzimierz Bzyl</field>
  </row>
  <row>
    <field>The Euromath System - a structure XML editor and browser</field>
    <field>J. Chlebkov, J. Gurican, M. Nagy, I. Odrobina</field>
  </row>
  <row>
    <field>Instant Preview and the TeX daemon</field>
    <field>Jonathan Fine</field>
  </row>
</rows>
```

This example files shows the first three entries in the abstract list of euroTEX 2001 at the time of this writing. The DTD of this XML file is:

```
<!-- DTD used for examples in article "From database to presentation
     via XML, XSLT and ConTeXt". -->

<!ELEMENT rows (row*)>

<!ELEMENT row (field*)>

<!ELEMENT field (#PCDATA)>
```

I still prefer DTDs above XML Schema's. They're far more readable and you can't express all well–formed XML files with XML Schema's anyway, so what's the advantage?

Our examples have the root tag `<rows>`. Our examples can have 0 or more `<row>` tags. Each `<row>` tag can have zero or more `<field>` tags.

With ConTEXt we can typeset this with the `\processXMLfilegrouped` macro:

```
\processXMLfilegrouped {example.xml}
```

The result of this is:

Re-introduction of Type 3 fonts into the TeX world Wlodzimierz Bzyl The Euromath System - a structure XML editor and browser J. Chlebkov, J. Gurican, M. Nagy, I. Odrobina Instant Preview and the TeX daemon Jonathan Fine

As you can see, this gives us just the plain text, no formatting is done. We can typeset our XML in a table with adding the following definitions and processing it again:

```
\defineXMLenvironment [rows]    \bTABLE \eTABLE
\defineXMLpickup       [row]     \bTR \eTR
\defineXMLpickup       [field]  \bTD \eTD

\processXMLfilegrouped {example.xml}
```

These definitions bind the start and end of a tag to a certain ConTEXt macro. Our result is now:

Re-introduction of Type 3 fonts into the TeX world	Wlodzimierz Bzyl
The Euromath System - a structure XML editor and browser	J. Chlebkov, J. Gurican, M. Nagy, I. Odrobina
Instant Preview and the TeX daemon	Jonathan Fine

The above example uses the new table environment of ConTEXt. As this specific environment cannot yet split across pages, the tabulate environment is a better choice for typesetting data. For this environment we need the following definitions:

```
\defineXMLpickup [rows]   {\starttabulate[|p(6cm)|p|]} \stoptabulate
\defineXMLpickup [row]    \NC \NR
\defineXMLpickup [field]  \relax \NC

\processXMLfilegrouped {example.xml}
```

Our result is now:

Re-introduction of Type 3 fonts into the TeX world	Wlodzimierz Bzyl
The Euromath System - a structure XML editor and browser	J. Chlebkov, J. Gurican, M. Nagy, I. Odrobina
Instant Preview and the TeX daemon	Jonathan Fine

I hope I've made clear the basic ideas of typesetting XML:

1. Make sure the XML data is in a proper tabular format.
2. Define mappings to the ConTEXt table, tabular or TABLE environment.
3. Use `\processXMLfilegrouped` to process your XML file.

Converting comma separated files to XML

Not always is data in the proper format. This section and the next cover converting non XML data into XML data.

Many programs can give CSV (Comma Separated Variable) data as output. An example of this format is:

```
"Fred","Flintstone",40
"Wilma","Flintstone",36
"Barney","Rubble",38
"Betty","Rubble",34
"Homer","Simpson",45
"Marge","Simpson",39
"Bart","Simpson",11
"Lisa","Simpson",9
```

In this format, fields are separated by comma's. String fields can be surrounded by double quotes. In XML this data should look like:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE rows SYSTEM "example.dtd">
<rows>
<row>
 <field>Fred</field>
 <field>Flintstone</field>
 <field>40</field>
</row>
<row>
 <field>Wilma</field>
 <field>Flintstone</field>
 <field>36</field>
</row>
<row>
 <field>Barney</field>
 <field>Rubble</field>
 <field>38</field>
</row>
<row>
 <field>Betty</field>
 <field>Rubble</field>
 <field>34</field>
</row>
<row>
 <field>Homer</field>
 <field>Simpson</field>
 <field>45</field>
</row>
<row>
 <field>Marge</field>
 <field>Simpson</field>
 <field>39</field>
</row>
<row>
 <field>Bart</field>
 <field>Simpson</field>
 <field>11</field>
</row>
<row>
 <field>Lisa</field>
 <field>Simpson</field>
 <field>9</field>
</row>
</rows>
```

Converting CSV to our ‘standard’ XML format can be done by a simple Perl script:

```
#!/usr/bin/perl -w use strict;

# test arguments
if (@ARGV == 0)
{
    die "Supply a filename as argument";
}

use Text::ParseWords;

open INPUT, "$ARGV[0]" or die "Can’t open input file $ARGV[0]: $!";

print "<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>\n";
print "<!DOCTYPE rows SYSTEM \"example.dtd\">\n";
print "<rows>\n";
while (<INPUT>) {
    chop;
    my @fields = quotewords(",", 0, $_);
    print "<row>\n";
    my $i = 0;
    foreach $field (@fields) {
        print "\t<field>$field</field>\n";
        $i++;
    }
    print "</row>\n";
}
print "</rows>\n";
```

Use this script as follows:

```
perl -w csv2xml.pl flintstones.csv > flintstones.xml
```

If you don't know what Perl is, you can read more about it at `http://www.perl.org`. Most UNIX users have Perl installed by default. Windows or Macintosh users can download Perl at `http://www.cpan.org/ports/index.html`. I'm not a particular fan of Perl, I can't remember the syntax if I've not used it for a few days. However, you can count on it being available for almost all operating systems.

Converting relational (sql) data to xml

Much of this worlds data resides in relational databases. It is not difficult to retrieve data from a relational database and turn it into XML.

Consider the following SQL table:

```
create table "family member" (
  "id_family member" smallint not null primary key,
  "surname" character varying(30) not null,
  "family name" character varying(40) not null,
  "age" smallint not null);
```

And the following insert statements:

```
insert into "flintstone" ("id_flintstone", "surname", "family name", "age")
  values (1, 'Fred', 'Flintstone', 40);

insert into "flintstone" ("id_flintstone", "surname", "family name", "age")
  values (2, 'Wilma', 'Flintstone', 36);

insert into "flintstone" ("id_flintstone", "surname", "family name", "age")
  values (3, 'Barney', 'Rubble', 38);

insert into "flintstone" ("id_flintstone", "surname", "family name", "age")
  values (4, 'Betty', 'Rubble', 34);
```

```
insert into "flintstone" ("id_flintstone", "surname", "family name", "age")
  values (5, 'Homer', 'Simpson', 45);

insert into "flintstone" ("id_flintstone", "surname", "family name", "age")
  values (6, 'Marge', 'Simpson', 39);

insert into "flintstone" ("id_flintstone", "surname", "family name", "age")
  values (7, 'Bart', 'Simpson', 11);

insert into "flintstone" ("id_flintstone", "surname", "family name", "age")
  values (8, 'Lisa', 'Simpson', 9);
```

A simple ANSI SQL query to extract the data and sort it in surname is:

```
select surname, age
  from flintstone
  order by surname
```

SQL output is usually not returned in XML format, and certainly not in the format we've described in the previous section. Here is the output that is generated by InterBase:

```
Database:  flintstones.gdb

surname                             age
============================= =======

Barney                               38
Bart                                 11
Betty                                34
Fred                                 40
Homer                                45
Lisa                                  9
Marge                                39
Wilma                                36
```

Before embarking on our tour to make this SQL more ConTEXt friendly, let's first explore how to get such output. Most relational databases offer a command line tool which can execute a given query. Frequently this tool is called `isql`. To present the above example I called `isql` as follows:

```
opt/interbase/bin/isql flintstones.gdb -i select1.sql -o select1.out
```

The actual InterBasequery, instead of the ANSI query presented above, looked like:

```
select "surname", "age"
  from "flintstone"
  order by "surname";
```

At the end of this section I present the command line tools of PostgreSQL and DB2.

There are two methods to typeset SQL output in ConTEXt:

1. Embed XML tags in the `select` statement.
2. Embed ConTEXt macro's in the select statement.

The first approach will be discussed in this section, the latter approach in the next section.

Embedding XML in a select statement to generate the format discussed before can be done with this InterBase `select` statement:

```
select
    '<row><field>',
    "surname",
    '</field><field>',
    "age",
    '</field></row>'
  from "flintstone"
  order by
    "surname";
```

The first two rows of the output look like this (slightly formatted for clarity):

```
Database:  flintstones.gdb

               surname                   age
====== ======= ======== ======== ======= ===== ======== ======

<row>  <field> Barney   </field> <field>    38 </field> </row>
<row>  <field> Bart     </field> <field>    11 </field> </row>
```

There are five problems with the output of InterBase `isql`, four of which are present in the above output:

1. There is no container tag, i..e the `<rows>` tag is missing.
2. The first line contains the database used: `flintstones.gdb`.
3. Column headers are present.
4. InterBase inserts columns headers after every 20 lines. Because there are just a few flintstones, this does not show up in my example, but I've typesetted thousands of entries, and there you have to deal with it. Fortunately, this can be easily solved by using the `-page` parameter and calling `isql` as follows:

   ```
   isql flintstones.gdb -i select1.sql -o select1.out -page 32000
   ```

   This will insert a column headers only every 32000 rows.
5. There is a lot of superfluous white space. White space is usually not a problem with TeX, and it also isn't with ConTeXt's XML typesetting macro's. I consider this a feature. If white space is a problem, you can attempt to write a somewhat different SQL statement like:

   ```
   select
     '<row><field>' + surname + '</field><field>' + age + '</field></row>'
     from flintstones
   ```

   This example uses string concatenation instead of putting the XML tags in different columns.

The first three problems cannot be solved by some parameter. We have to use Perl again. Here my script to remove the column headers of an InterBase SQL output file and at the appropriate container tag:

```
#!/usr/bin/perl -w use strict;

# test arguments
if (@ARGV == 0)
{
    die "Supply a filename as argument";
}

open INPUT, "$ARGV[0]" or die "Can't open input file $ARGV[0]: $!";

# read up to the line with ====
while (<INPUT>) {
    if (/^=.*/) {
        last;
    }
};

# skip one more line
<INPUT>;

# now just dump all input to output
print "<rows>\n";
while (<INPUT>) {
    print;
}
print "</rows>\n";
```

The output is now a lot more like XML:

```
<rows>
<row><field> Barney                 </field><field>      38 </field></row>
<row><field> Bart                   </field><field>      11 </field></row>
<row><field> Betty                  </field><field>      34 </field></row>
<row><field> Fred                   </field><field>      40 </field></row>
<row><field> Homer                  </field><field>      45 </field></row>
<row><field> Lisa                   </field><field>       9 </field></row>
<row><field> Marge                  </field><field>      39 </field></row>
<row><field> Wilma                  </field><field>      36 </field></row>

</rows>
```

We can typeset this with:

```
\defineXMLpickup [rows]
  {\starttabulate[|p(7cm)|p|] \HL\NC surname \NC age \NC\NR\HL}
  {\stoptabulate}
\defineXMLpickup [row]
  \NC \NR
\defineXMLpickup [field]
  \relax \NC
```

```
\processXMLfilegrouped {select2.xml}
```

And the result looks great!

surname	age
Barney	38
Bart	11
Betty	34
Fred	40
Homer	45
Lisa	9
Marge	39
Wilma	36

As promised here the commands to extract data from DB2 and PostgreSQL. For DB2 use the `db2` command, like this:

```
db2 -td\; -f myfile.sql -r myfile.out
```

The `-td` option defines the command separator character. I use the ';' character for this. After the `-f` option follows an SQL file with one or more `select` statements. With the `-r` option you can redirect the output to a file.

PostgreSQL has the `psql` to extract SQL data. Use it like this:

```
psql -d flintstones -f myfile.sql -o myfile.out
```

The `-d` option specified the database name. The `-f` option specifies the file with the `select` statements. The `-o` option redirects the output to a file.

Typesetting SQL without generating XML

In the previous section SQL output was enhanced with XML tags. The XML tags were then mapped to ConTEXt macro's. It is possible to skip the XML tag generation by directly putting the ConTEXt commands in the SQL `select` statement:

```
select
    '\NC',
    "surname",
    '\NC',
    "age",
    '\NC\NR'
  from "flintstone"
  order by
    "surname";
```

From the output we again have to remove the lines we don't need. This can be done with more or less a Perl script like the one shown before. It can be even simpler as it doesn't have to add something before or after the data. After cleaning up the output should look like:

```
\NC   Barney                              \NC          38 \NC\NR
\NC   Bart                                \NC          11 \NC\NR
\NC   Betty                               \NC          34 \NC\NR
\NC   Fred                                \NC          40 \NC\NR
\NC   Homer                               \NC          45 \NC\NR
\NC   Lisa                                \NC           9 \NC\NR
\NC   Marge                               \NC          39 \NC\NR
\NC   Wilma                               \NC          36 \NC\NR
```

The ConTEXt code to typeset the data in this case is:

```
\starttabulate[|p(7cm)|p|]
\HL
\NC surname \NC age \NC\NR
\HL
\input select3.tex
\stoptabulate
```

Transforming XML with XSLT

In the preceding section we've seen how XML can be generated from non XML sources. This section is concerned with generating XML that can be typeset in ConTEXt from existing XML sources. Usually XML sources are not in a format that can be typeset easily. Such XML has to be transformed to the XML format presented earlier. Fortunately there is an entire language devoted to transforming XML to XML. It is called XSLT, a quite complete and not too difficult language. More information about XSLT can be found at `http://www.w3.org/Style/XSL/`.

The first example is making a list of euroTEX 2001 authors and their presentations. The program listing in XML at time of this writing looked like this:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<program>
  <day weekday="Monday" date="24 September 2001">
    <item time="9.00h"><opening/></item>
    <item time="9.15h">
      <presentation>
        <author>Hans Hagen</author>
        <title>Overview of presentations</title>
      </presentation>
    </item>
    <item time="9.45h">
      <presentation>
        <author>Wlodzimierz Bzyl</author>
        <title>Re-introduction of Type 3 fonts into the TeX world</title>
      </presentation>
    </item>
    <break time="10.30h" type="coffee"/>
    <item time="11.00u">
      <presentation>
```

```
        <author>Michael Guravage</author>
        <title>Literate Programming: Not Just Another Pretty Face</title>
      </presentation>
    </item>
    </day>
</program>
```

With the following XSL stylesheet we can transform this to our standard XML format:

```
<?xml version="1.0"?>

<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/program">
  <rows><xsl:text>&#xa;</xsl:text>
  <xsl:apply-templates select="day/item/presentation"/>
  </rows><xsl:text>&#xa;</xsl:text>
</xsl:template>

<xsl:template match="presentation">
  <row>
    <field><xsl:value-of select="author"/></field>
    <field><xsl:value-of select="title"/></field>
  </row><xsl:text>&#xa;</xsl:text>
</xsl:template>

</xsl:stylesheet>
```

This transformation gives us something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<rows>
<row><field>Hans Hagen</field><field>Overview of presentations</field></row>
<row><field>Karel Skoupy</field><field>NTS implementation</field></row>
</rows>
```

How we can typeset this, should be clear enough by now! It is probably more helpful to explain the XSL stylesheet a bit. An XSL stylesheet usually consists of many `<xsl:template>` tags. The XSL processor takes the first one that matches the root node (the '/' separator) as the main template. It starts the transformation there (The real rules are somewhat more difficult, but not important here). In our case we match the `</program>` node. We output the `<rows>` tag and next we output all the presentations. This is done with a `<xsl:apply-templates>` tag that searches for a template that matches the selected nodes. In the template that matches the presentation node, we output the `<row>` tag and the individual fields.

An XSL processor can do many advanced things with XML, see figure 4. It cannot only generate XML, but also straight ConTEXt code for example, or just plain text.

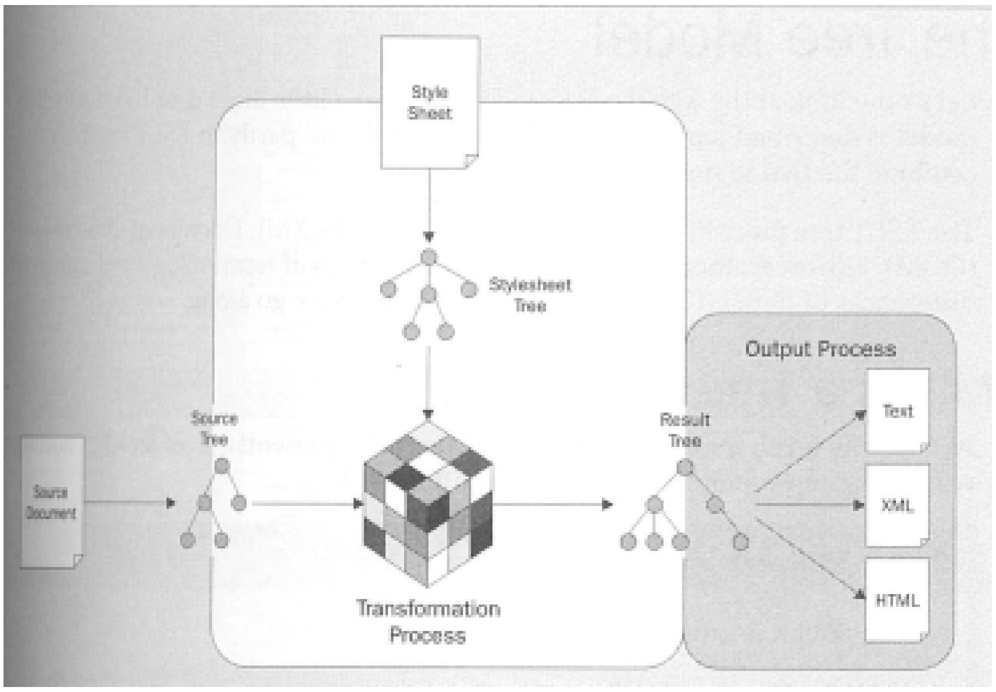

Figure 4 From xml to xml, text or what else

Besides just selecting the presentation, we can also sort them. We can do that with embedding a sort instruction in an `<xsl:apply-templates>` instruction:

```
<xsl:apply-templates select="day/item/presentation">
  <xsl:sort select="author"/>
</xsl:apply-templates>
```

If you want to learn more about XSLT, I can recommend "XSLT Programmer's Reference" by Michael Kay, also the author of the well–known XSLT processor Saxon. For this document I used Xalan, another well–known processor, see `http://xml.apache.org/xalan-c/index.html`.

Conclusion

My goal has been to give you a quite exhaustive overview of typesetting structured data, but not already expressed as TEX macro's, with ConTEXt. I did this by showing how you can typesetting XML in ConTEXt. And I covered converting from comma separated files, relational database data and XML to an XML format that can be handled easily by ConTEXt.