Typesetting product catalogs and other database-driven documents with the speedata Publisher

Patrick Gundlach

Abstract

The speedata Publisher is a database publishing system based on LuaTEX. Although it is a fully commercial-driven development, it is available under an open source license (AGPL). The main goal of the speedata Publisher is to provide the high quality of TEX's typesetting output while fulfilling the needs of database publishing. The speedata Publisher implements its own language for defining layout rules. This language is inspired by HTML, XSL and CSS, and specializes in layout generation and excels in optimizing layout such as rearranging objects on the page, whitespace optimization, copyfitting and other means.

LuaTEX has the ability to manipulate and build the internal data structure that TEX uses to assemble the pages and to break paragraph into lines. It provides an extremely powerful environment for non-standard typesetting tasks by allowing all necessary steps to be done programmatically while still falling back to TEX's algorithms.

1 Introduction

LATEX is great if you want to write articles or books. But how about if you want to publish product catalogs or data sheets? Is TEX still the tool of choice?

Normal text vs. data from databases

Product catalogs and similar documents are at best created from data stored in a database. But what distinguishes normal text from data retrieved from databases?

Texts in a TEX document could look like this

```
The quick\footnote{yes, really quick} brown
fox jumps...
```

while data from the database has a rather schematic structure:

```
<productdata>
  <articlegroup
        name="interior lights"
        number="123">
    <article number="123-12345">
      <property1>...</property1>
      <property2>...</property2>
    </article>
    <article number="123-12346">
      <property1>...</property1>
      <property2>...</property2>
    </article>
  </articlegroup>
</productdata>
```

Data processed in such systems is almost exclusively formulated in XML, regardless of how it is stored on disk. Textual descriptions are generally stored either as plain text or as HTML formatted text, rarely as Markdown formatted text.

A fundamentally different approach is therefore necessary to process data from databases. The processing of documents is no longer linear, i.e. from 'top to bottom'. Instead, the data must be assembled according to a logic that differs from application to application. From the data above, it is not possible to see how it should be represented. Whether it is intended to be typeset as a table, a nicely designed page with several products, or as a data sheet — this cannot be seen from the data alone.

1.1 Strict separation of data and layout

Apart from a few exceptions, no information about the appearance is found in the data. While the strict separation is in theory a good concept, it makes the typesetting part much harder. Sometimes the separation is even impossible to maintain. For example: when having a full page background image and text to be placed in a *good* position, you need some information coming from a human decision.

So how do you arrange the data on a page? According to which rules must the elements be placed? To get to the solution, it helps to look over the shoulder of a graphic designer when creating a document (e.g., a product catalogue). Professional graphic designers work according to rules: How is a catalog structured? Which products should be displayed in which way? How many products fit on one page? Which colors and fonts are used? Where is a page break inserted?

The same rules are usually applied when filling the pages, even if pages often look very different. If you can manage to write these often formal rules in text form or in a programming language, you are often already very close to the goal.

1.2 Software used for product catalogs

Adobe InDesign is certainly the software most often used to create documents with non-linear layout. It is professional desktop publishing software for Windows and Mac. This very powerful program has excellent graphic qualities and can be automated by plugins. There is also QuarkXPress, which works similarly. However, these programs have limitations in automation. In practice, these programs usually work by using a database interface and page templates to fill pages. When finished, the pages need

to be finalized via a tedious manual process. This workflow is suitable for documents that change only occasionally, but quickly reaches its limits if the database changes very frequently or if documents are to be generated fully automatically.

1.3 TEX as an alternative?

For the readers of *TUGboat*, the question is of course how far TEX can be used here. The advantages of TEX are well-known to us:

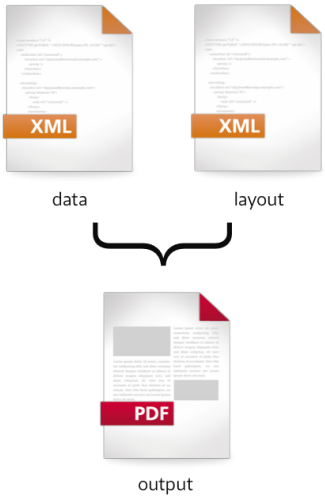
- Full automation. You can set up the process so that a document is generated at the push of a button or at a given time.
- Free software. No dependence on proprietary software. TEX can be used on any number of computers without any additional license fees.
- High output quality. We all know that TEX's line breaking algorithm is superb.
- High speed. TEX can output up to 300 pages per second on my old 2015 laptop with almost no startup time. (Less than that with more complex documents and lots of fonts.)

A few difficulties remain, however:

- XML encoded in UTF-8 as input format. TEX is neither made for XML input nor for UTF-8 processing. LuaTEX allows more than 256 glyphs in a font, so that helps significantly. Since April 2018 LATEX defaults to UTF-8 input, so this part is getting better.
- Assembling the data. As seen, the input is not linear. You have to go back and forth within the data and fetch data from different XML elements in the input.
- Output of HTML sources. Text in database publishing is usually stored as plain text or as HTML fragments. To render HTML, a CSS+HTML parser is needed.
- Optimization of pages. Many applications demand some kind of whitespace optimization such as adding images to the page until the text completely occupies the space. Other applications require, for example, reducing the text size until the text fits on one page (copyfitting).

2 speedata Publisher

I have developed the speedata Publisher precisely for the purpose of non-linear documents with high demands for layout flexibility. It is open source software based on LuaTEX. You can download it from the homepage [1] and use it immediately without any further installation of dependencies (not even TEX is required; it is included in the ZIP file). A comprehensive manual [2] describes in detail how to use the software.



In addition to the data, which must be available in XML format, the layout instructions are also formulated in XML. This has several advantages.

- You stay in the XML environment, which is required for handling the data anyway.
- With a schema, editing XML in a text editor is also fun.
- You see syntax errors immediately.
- XML can be easily created and transformed from programs.

2.1 The speedata layout language

Since the data can be structured in any way, the layout language must be very flexible. It must also be possible in the layout files to formulate and evaluate the above-mentioned design rules. Supporting queries to the data is necessary, such as: how wide has the object become? Is there still enough space on the page?

Existing layout or formatting languages do not allow such flexibility. (X)HTML has no programming support, XSL has no knowledge of layout, CSS is fine for output, but has only limited programming abilities. XSL-FO is rigid in its output and has no way to respond to dynamic queries. In this respect, the layout language is a mixture between the languages mentioned here.

2.2 Hello, world

In the following sections I would like to give a small insight into the layout language. The classic "Hello, world" example serves as an introduction to speedata.

In database publishing the input usually consists of two files: the data file and the layout file. I ignore images and font files for now.

The data file in the "Hello, world" example consists of one line:

```
<greeting>Hello, world!</greeting>
```

This file must be saved in an otherwise empty directory under the name `data.xml`.

The layout file (`layout.xml`) is rather larger, and looks scarier than it really is (the arrow indicates an editorial line break):

```
<Layout
  xmlns="urn:speedata.de:2009/publisher/en"
  xmlns:sd="urn:speedata:2009/publisher/  ↙
    functions/en">
  <Record element="greeting">
    <PlaceObject>
      <Textblock>
        <Paragraph>
          <Value select="."/>
        </Paragraph>
      </Textblock>
    </PlaceObject>
  </Record>
</Layout>
```

If you have the speedata Publisher installed, you can run the command `sp` to create a PDF file.

The processing starts at the root node in the data file. In this case it is the element `<greeting>`. It sets the "focus" to this element so you can access it from the layout file. The software now looks for an entry point in the layout file (`<Record>`) and executes every command that is found within this `<Record>` element. In this example, a text block is output for the current element. The `<Value>` uses the dot (`.`) to select the contents of the current focussed element in the data file, in this case just the text "Hello, world". The dot is a so-called XPath expression with which you can select data. A more detailed description of the "Hello, world" example can be found in the manual [3].

2.3 Dynamic layouts

To enable dynamic layouts, the speedata layout language has programming options such as loops and variables and the ability to query the appearance of the page and objects that are put in virtual areas. Together, these have enough expressiveness to implement complex layout requirements. As with TEX, you can put any objects into a virtual area which is not output in the PDF. In TEX this is called a box; here it is called a group. For example, to compare the width of a group to the size of the page and act on the outcome of the comparison:

```
<Switch>
  <Case test="sd:group-width('mybox') >   ↙
    sd:number-of-columns()">
    <!-- too wide, recalculate -->
  </Case>
  <Otherwise>
    <!-- great, fits on the page -->
    <PlaceObject groupname="mybox" />
  </Otherwise>
</Switch>
```

The requirements in practice are of course much more complex. Interestingly, however, a few building blocks are sufficient to create complex layouts. Many functions in speedata Publisher fall into the category *syntactic sugar*, i.e. not strictly necessary but helpful constructions. For example, an image file can be specified as a fallback for image output:

```
<Image file="myfile.pdf"
       fallback="placeholder.pdf" />
```

This could also be written in a different way:

```
<SetVariable
      variable="filename"
      select="'myfile.pdf'" />
<Switch>
  <Case test="sd:file-exists($filename)">
    <Image file="{ $filename }"/>
  </Case>
  <Otherwise>
    <Image file="placeholder.pdf"/>
  </Otherwise>
</Switch>
```

2.4 Grid typesetting

Objects can be placed anywhere on a page or in a grid. Grids can be any size and define a coordinate system that helps in placing objects automatically and ensuring that no object overlaps any other.

```
<Layout xmlns="urn:speedata.de:2009/  ↙
    publisher/en"
  xmlns:sd="urn:speedata:2009/publisher/  ↙
    functions/en">

  <SetGrid height="12pt" nx="10"/>
  <Trace grid="yes"/>
  <Pageformat width="8cm" height="4cm"/>

  <Record element="data">
    <PlaceObject column="3" row="2">
      <Textblock>
        <Paragraph>
          <Value>Hello world!</Value>
        </Paragraph>
      </Textblock>
    </PlaceObject>
  </Record>
</Layout>
```

Using a grid has several advantages:

- Every object allocates an area on the page. It is easy to check how big this area is.
- Objects that are placed on a grid cannot overlap, unless forced to. The system moves the object to the next free space.
- It is easy to achieve typesetting on a grid just by letting the output start at a new grid row.

Of course not everything can be placed within a grid. Logos or background images for example need to be placed at absolute positions:

```
<!-- grid -->
<PlaceObject row="4" column="5">
    <Image file="_samplea.pdf" width="5"/>
</PlaceObject>

<!-- absolute -->
<PlaceObject row="12mm" column="5cm">
    <Image file="_samplea.pdf" width="5"/>
</PlaceObject>
```

2.5 Other features

The speedata Publisher has many, many features. Here, I'd like to highlight just a few of them.

Accessibility It is possible to attach logical structure to the texts placed in the PDF so it can be PDF/UA (Universal Accessibility) compliant.

HTML input The speedata Publisher comes with a CSS+HTML parser that lets you typeset documents written in HTML as they would look in a browser.[1]

Master pages Page templates, including logos and other static and dynamic information, can be defined together with arbitrarily complex conditions for when the page will be chosen by the software.

Page areas You can define areas on the page to let text flow from one area to the next area. This is used in magazine typesetting.

HTTP assets Images and all other resources can be loaded on the fly from the web. This makes it easy to use digital asset management software.

[1] This feature is under development, so not all aspects are implemented yet.

Image wrapping Images can be (automatically) enriched with information about where text can wrap around the image. The paragraph shape is calculated automatically.

Advanced tables The speedata Publisher does not use any of TEX's table code. It ships with its own table model, which is inspired by HTML and supports static and dynamic headers and footers, controllable page breaks, running totals, complex table cell backgrounds and much more.

Server mode Included in the Publisher is a REST API that listens for incoming HTTP requests to start publishing runs. This makes it easy to build a server infrastructure for typesetting jobs.

Strong quality assurance There are more than a hundred documents that are automatically compared before making changes to the software, so we can be assured old documents will be typeset without changes in future versions.

3 speedata and LuaTEX

As mentioned above, LuaTEX is used as the backend for speedata. Almost all parts of the speedata Publisher are written in Lua. No code from the plain, ConTEXt or LATEX formats is used. There is a tiny TEX wrapper that jumps directly into the Lua mode, which does all the processing.

```
\catcode`\{=1 \catcode`\}=2
\directlua{require("publisher.spinit")}
\end
```

All other functions are at the Lua level. These are, for example

- Parse the XML files (data and layout)
- Read in all images and font files
- Execute the program statements in the layout
- Assemble the data structures for TEX
- ... and much more

Some of the routines are written in the programming language Go and included as a library at runtime. This library handles the loading of resources via HTTP (including caching) and parsing of HTML and CSS files. It was easier to use existing libraries for these tasks than to rewrite them in Lua from scratch.

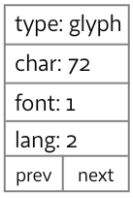
3.1 TEX without \TEX

If no input comes in the form of TEX code, how is LuaTEX able to typeset text and place other objects into the PDF?

TEX normally reads the files with the macro instructions (e.g., `\section`) and stores the contents as so-called nodes after some processing. These are the smallest data units, which store e.g. a character or a

glue. With these data units everything that is visible in the output (along with some other technical information) is represented. These data structures can then be used to create DVI or PDF output. Thanks to LuaTEX, these nodes can also be created and manipulated in Lua. Thus, the main part of the Lua program code consists of generating such nodes from the input data and the instructions of the layout file.

Node lists are linked lists of single nodes, which can also contain lists themselves. For example, the content of a horizontal box `\hbox{...}` is a list and the box itself can be part of another list. Each node consists of different fields, depending on what is stored. For example, the character "H" could be represented as a node as follows.

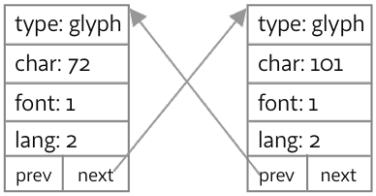


Such a character could easily be created with the following Lua commands (the double dash `--` is a Lua comment):

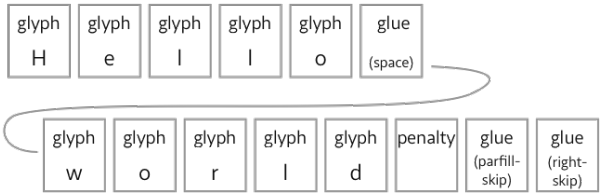
```
h = node.new("glyph")
-- 72 is the ascii code for H
h.char = 72
h.font = 1
h.lang = 2
```

You can chain the nodes by, for instance, setting the `prev` and `next` pointers:

```
-- as above
e = node.new("glyph")
e.char = 101; e.font = 1; e.lang = 2
h.next = e
e.prev = h
```



In this way, entire nodelists can be generated.



We are close to a nodelist that can be used for output. Three things are missing from a "perfect" paragraph:

1. hyphenation
2. kerns
3. ligatures

Hyphenation: there is a Lua function for TEX's hyphenation routine: `long.hyphenate(nodelist)`. When called, LuaTEX changes the nodelist and inserts the so-called discretionaries that signal a hyphenation point.

Kerns and ligatures: there are two very helpful routines that add ligatures and kerns to the nodelists: `node.ligaturing(nodelist)` for ligatures, and for kerns `node.kern(nodelist)`. The former replace some glyph nodes with ligatures, so that they can be dissolved again when a word is hyphenated. The latter inserts small (often negative) spaces between glyphs. Regarding ligatures, one can argue whether it is still appropriate to do this via the TEX mechanism. OpenType fonts often contain other ligatures that would have to be translated for TEX's ligature mechanism. Furthermore, libraries like HarfBuzz offer much more powerful functions for ligatures.

If hyphenation, kerns and ligatures are inserted, you can use

```
tex.linebereak(nodelist, parameter)
```

to create a paragraph broken by TEX. The parameters specify the values for paragraph settings like emergencystretch or linepenalty but also the paragraph style (parshape). The result is a vertical box with single lines in horizontal boxes.

3.2 Output of nodelists

After nodelists have been assembled, they can be output. The speedata Publisher collects all material for the pages and outputs it in one go. `tex.shipout(n)` outputs the TEX box with the number n:

```
nodelist = node.vpack(nodelist)
tex.box[1234] = nodelist
tex.shipout(1234)
```

Before output, structural elements may have to be written to the PDF for PDF/UA. To do this, the content of the page is analyzed and a PDF object structure is written to the PDF for accessibility purposes.

3.3 Fonts and languages

In the example above, we just used some dummy values for 'font' and 'lang'. Usually TEX loads the font files or language patterns with `\font` and `\patterns`. The speedata Publisher has its own routines for both to allow UTF-8 input, similar to `fontspec` and `luaotfload` for LATEX.

A new language can easily be loaded in LuaTEX:

```
local l = lang.new()
l:patterns(pattern)
local id = l:id()
```

Here `pattern` is the content of a pattern file.

Loading a new font is a bit more complicated. The FontForge library that is part of LuaTEX is used to get information about (OpenType) fonts. An alternative routine based on HarfBuzz is planned, which is part of LuaTEX (under the name `luahbtex`) since the last TEX Live release.

```
font, err = fontloader.open(filename_with_path)
...
fonttable = fontloader.to_table(font)
```

`fonttable` now has all font properties in an extensive table, which can be made available to TEX:

```
local f = { }
f.name          = fonttable.fontname
f.fullname      = fonttable.fontname
f.designsize    = size
f.size          = size
f.direction     = 0
f.filename      = fonttable.filename_with_path
f.type          = 'real'
...
f.characters = {
    -- code for all glyphs in a font
}

-- define the font:
fontid = font.define(f)
```

You can use the font id in the nodelists above.

3.4 PDF specials

The PDF contains a lot more information than that which is visible at a first glance. For example, bookmarks, hyperlinks, document structure for accessibility, attached documents for electronic invoices are elements that need to be written into the PDF. Thanks to pdfTEX and the API in LuaTEX, this is an easy task once the required syntax for these PDF objects are known. They can be written to the PDF as follows:

```
pdf.obj(...)
```

This function has several different parameters that allow compressed or uncompressed text or data to be written as a simple or a stream object.

There are also visible objects that cannot be created with TEX's graphics routines. Colors, transparency, shades and other objects need to be written with PDF drawing instructions. For example, the borders in the following picture need to drawn with instructions such as `0 0 m 1 5 l` which means "move to position (0,0) and draw a line to (1,5)". There are operators to draw lines and Bézier curves, fill paths, clip contents from inside and outside of given areas and many other drawing operators.

These operations can be inserted into the PDF by *whatsits*:

```
n = node.new("whatsit","pdf_literal")
n.data = "0 0 m 1 5 l"
```

and then insert this whatsit into the nodelist (output grayscaled for *TUGboat*):

4 Outlook and conclusion

Of course I can only scratch the surface in this article. LuaTEX and also the speedata Publisher are two very powerful pieces of software. As can be seen, the speedata Publisher would not be possible in this way without LuaTEX. There is no need to understand TEX's macro language to use TEX, even for the programmer.

The speedata Publisher is in active development since 2009. I have a lot of plans for the future development (such as HarfBuzz integration), but the (paying) customers are the ones who drive most development of new features.

I'd like to invite you to try out the software, ask questions, look at the showcase on the homepage or just browse the manual for inspiration.

To close with Donald E. Knuth's words: Go forth now and create masterpieces of the publishing art!

References

[1] speedata homepage. `https://www.speedata.de`.
[2] speedata manual. `https://doc.speedata.de`.
[3] "Hello, world" example in speedata manual. `https://doc.speedata.de/publisher/en/helloworld/`.

◇ Patrick Gundlach
speedata GmbH
Odilostraße 43
13467 Berlin
Germany
`gundlach (at) speedata dot de`
`https://www.speedata.de/`