XSLT throughout the document lifecycle

Wendell **Piez**

Abstract

Originally and primarily, of course, XSLT is designed to fit within the “production” phase of certain kinds of electronic media. A text has been prepared for publication; the outcome of this process, in the classic paradigm, is XML; XSLT is used to convert this “source document” to an output format such as HTML or PDF. From the time of the release of XSLT 1.0 in 1999, however, XSLT has been used for much more than the simple case of rendering documents that are fully-edited and “good to go”. For example, two years ago at XML2003, I showed how XSLT stylesheets can be applied to semi-automate some of the more laborious aspects of copy editing, proof-reading and fact checking.

More broadly, XSLT applications within the document lifecycle can be divided between pure XSLT applications, in which an XSLT engine is the main processor and its output will be used directly, and “ancillary” or “supporting” XSLT applications, in which XSLT is used to enhance the functionality of another tool. If the latter sort of XSLT is included, these various applications include (but are not limited to) the following: for micro-publication (internal pre-publication; peer review); for extra-schema validation; for data aggregation and filtering, for example, generating indexes to a set of documents or collecting a set of metadata; in support of schema technologies (schema documentation, profiling); in CM systems (both open-source and proprietary CM and data-federation systems increasingly support XSLT queries and aggregation); and for task organization and configuration (e.g. XSLT can be used with Ant for task management/automation).

In many cases, these applications are possible due to the “network effects” of bringing XML technologies in general into workflow management. For example, it is now considered routine to generate file listings in XML (there are free toolkits available, such as XML Starlet, that can provide just this sort of low-level function) -- not because XML is so great in itself, but because XSLT can then be used to program innumerable “file-wrangling” sorts of chores. A tool like Apache Ant, which uses XML to configure many jobs that might otherwise be done by shell scripts (or by hand), can be enhanced with XSLT to provide dynamic processes like copying all the graphics referenced by a given document from an archive into a working directory.

Table of Contents

One of the great reasons for XSLT's success is that by designing a lightweight language for generic transformations, the XSLT 1.0 Working Group gave us a technology whose utility goes far beyond its core use case, the display of XML data.[1] Although not quite a general-purpose language with the full capabilities of a Java or Python[2], XSLT's applications have proven to go far beyond the generation of PDF or HTML files. Assuming you have XML data, XSLT is going to be one of the more useful tools in your box: your pipefitter, if not your hammer and tongs.

This paper supposes you are lucky (or unlucky) enough to have XML entirely end-to-end in your system, but does not assume as much: everything proposed here may also find some use in applications of XML that do not run end-to-end, but begin with or include other formats along the way. Since what we are talking about here amounts to using XSLT as "glue", its suggestions apply across the board, even if you use XML only in parts of your work processes. Indeed, this is probably the more normal case. But thinking about what XSLT can do hypothetically in a system where XML has already become the norm (in part because of what one can then do with it in XSLT) enables us to explore a fuller range of applications of the kind of data transformations XSLT supports.[3]

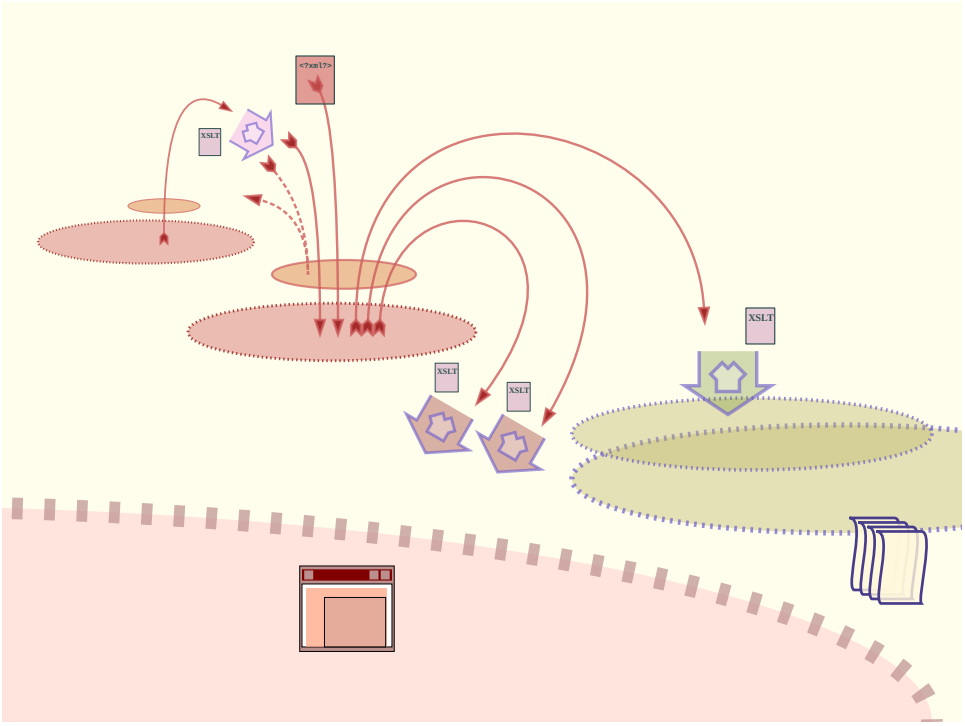
Even given that working assumption, the question arises what we mean by "document lifecycle". We can set that interesting issue aside by admitting that what we are inescapably talking about workflow: some kind of sequencing of operations that takes some kind of input and produces it in some way ready for distribution to some kind of consumer or audience. Whether you produce user manuals for electronics equipment, a weekly financial newsletter, government reports, or any of innumerable different kinds of document production, this general situation is the same.

A high-level diagram simplifying a typical document production workflow (such as might be used to produce a set of conference papers) appears in Figure 1, "A model of a workflow". Considering how even this simple an arrangement of roles and tasks is architected, it is possible to conceive of an abstract "ideal" workflow (see Figure 2, "An idealization"); having done so, however, we find it is that much easier to discriminate how complex real-world workflows actually tend to be (see Figure 3, "Isn't yours more like this?").

[1] Famously, the XSLT Recommendation does not prejudge how useful XSLT will be for tasks outside the formatting application domain, allowing its possibility without stressing it: "XSLT is also designed to be used independently of XSL. However, XSLT is not intended as a completely general-purpose XML transformation language. Rather it is designed primarily for the kinds of transformations that are needed when XSLT is used as part of XSL." [XSLT-1.0]

[2] Actually XSLT is generally limited not by its capabilities as a language, but by the architecture in which it is usually deployed, its lack of system-level calls and so forth. Dimitre Novatchev has demonstrated how general-purpose XSLT can be computationally: see [FXSL]

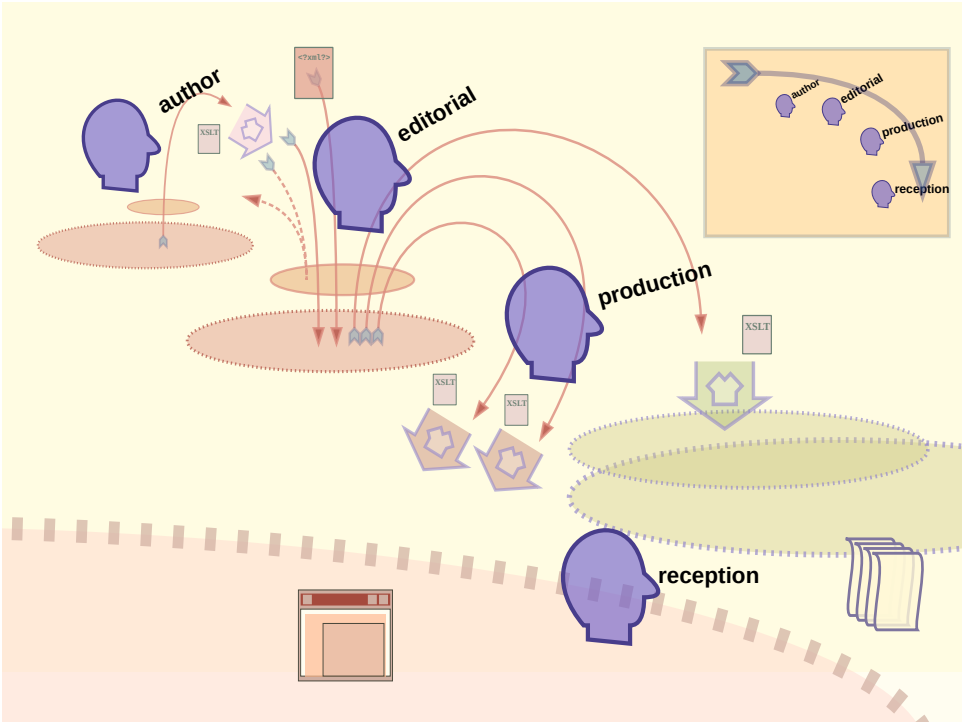
[3] That XSLT is designed for "down-conversions" while we also have "up-conversions" to worry about complicates this picture, but does not change it. Indeed, it gives us impetus to do the right thing, namely do all the upconversion as near to the front of the process as possible.



A simple instance. Highly theoretical. Real-world workflows will be elaborations of basic patterns seen here.

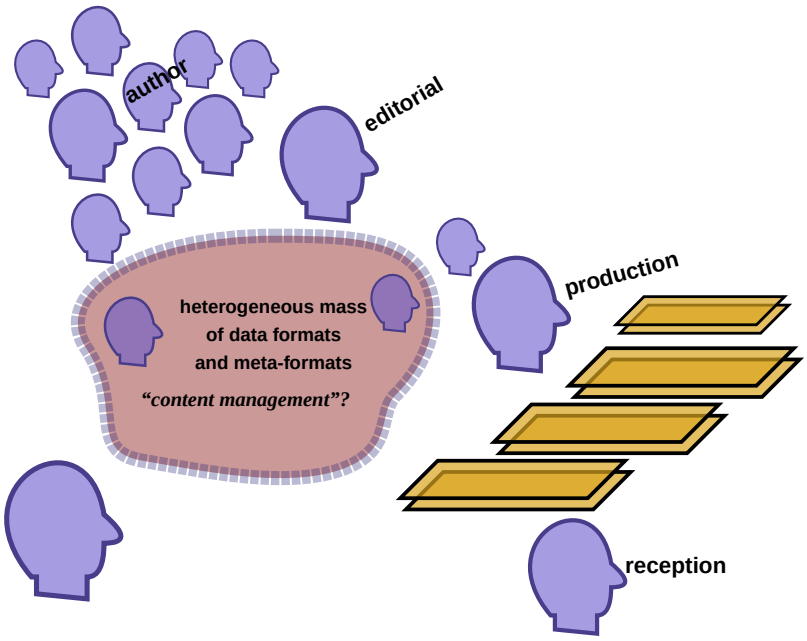
If you look carefully, you can see the classic “hub-and-spoke” architecture here. Incoming formats are unified, then split again as outgoing (presentation) formats.

Figure 1. A model of a workflow



If only it were this simple: workflow reduced to its minimum, a single expression through a succession of agents. Below this point, roles fuse altogether and document production becomes art. A two-person operation is still a partnership. Only when author, editor (or publisher or producer) and production specialist are discriminated we can begin to think of an organization that transcends individuals — that is likely to benefit, that is, from the formalization and automation of processes implied by XML. This is not to say that any of the applications described in this paper will not work as well on even a smaller scale: on the contrary.

Figure 2. An idealization



This cartoon only hints at the complexities lying under the surface here: rather than trying to depict the complications of the real world, it simpy warns us not to forget them: they are endless and only too familiar. The proliferation of authors or sources (each author having inevitably some distinctiveness), the profusion of data formats and meta-formats in the best of cases, the multiplicity and complexities of the platforms on which systems must be developed and maintained, all of these make for an enormously complex and inconstant working environment. Given this complexity, we might generally be skeptical of a top-down approach to managing workflow. It may be more practical to work bottom-up instead.

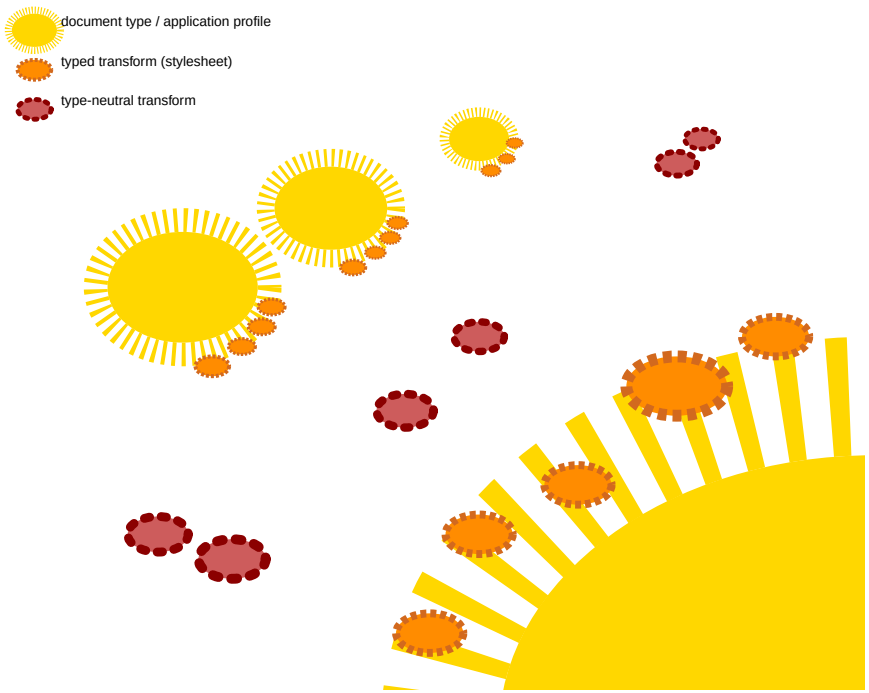
Figure 3. Isn't yours more like this?

It is possible, in theory, to apply XSLT directly to this problem of flow and flow management. Many CMS (content management) systems offer just this capability, or soon will, while specific applications of XSLT to workflow in particular domains may help enormously for particular kinds of workflow.[4] Here as always, however, the devil is in the details, and not every process is mature enough to be susceptible to managing entirely from the top, even with the assistance of the most versatile processing language. Nevertheless, if it makes sense to have XML data, it should make sense to have it even if one doesn't automate *everything*, but in a system where there is the same mix of technologies in use and under development as there is in any workplace. Accordingly, we concentrate here not on top-down approaches to modeling workflow in XML and managing it with XSLT (as both a presentation and a transformation technology), but on bottom-up approaches: how can you get useful chores done with XSLT.

These applications are too various to count and characterize. A paper I delivered at XML 2003 (an IDEAlliance conference in Philadelphia, PA) [Piez-2003] suggested some uses of XSLT that went beyond simple document formatting, giving some indication of where things can go. But that paper was given to describing applications of XSLT that, while not devoted necessarily to presentational tasks, were still developed to work over a particular class or type of input document. For example, a false-color proofing stylesheet will be tuned to a similar range of inputs (typically a set of

[4]Systems of direct workflow management, whether application-specific or generic and whether externally specified ("standardized") or proprietary or semi-proprietary for particular tools, are outside the scope of this paper. A fairly mature implementation (with a fair degree of vendor support) of a language to describe specifically print production is JDF (Job Definition Format). See http://www.adobe.com/products/extreme/jdf.html. The reader might also consider http://www.wfmc.org/standards/docs.htm (the Standards page of the Workflow Management Coalition).

documents conforming to a given schema) as a more conventional presentational stylesheet. Yet because XSLT can take any XML as input, and so the scoping of stylesheet to document type is not determined by the tools, but can be entirely *ad hoc*, "untyped" applications are also straightforward to implement. Finally, some stylesheets (such as an aggregator that extracts and merges data from a set of inputs) may inhabit a mid-zone between these extremes. This variety is complicated further by the fact that there is typically more than one document type (whether this term is construed loosely or formally) in a system, and so also more than one set of stylesheets to handle them.



Even if you have a single primary document type, and not several, you may have any number of ancillary or ad-hoc schemas or "meta-formats" (which will include the schemas for any schema documents, naturally). Dictionaries, process outlines and manuals, specifications, calendars and timesheets, control files, metadata aggregations and profiles, indexes, all of these may classify as types or derivative subtypes, which may have their own sets of transforms. Other transformations may be designed to be permissive of more and different kinds of input, to do generalized tasks, diagnostics and utilities over anything.

Figure 4. Systems of document types and transforms

For example, the production of a conference proceedings may include not only the collection of papers themselves (which may require a set of transforms for formatting, validation, normalization etc.), but also several ancillary (ad-hoc) document formats: lists of the papers (since given such a list, an XSLT processor can query the documents to extract more information), metadata extracts (for the generation of indexes), front matter copy, and so forth, to say nothing of target presentation formats such as HTML or XSL-FO/PDF. More complex productions can have correspondingly more different kinds of documents, both formalized and process-driven.

These distinctions suggest we can categorize XSLT applications along a scale, running from typed stylesheets (stylesheets that operate over fairly constrained sets of input) to untyped or generic stylesheets (that operate over any inputs), including a set in between, that expect input with special constraints or that operated generically over any kind of input or a very generalized kind of input. It is convenient to draw this somewhat arbitrary line based on how these constraints on the input are enforced, since not all inputs have (or need to have) conformance with particular schemas for their transformation to provide some useful functionality.

1. Typed stylesheets

These are transformations that are written to perform over documents that are fairly well controlled as members of a "document type", typically by the requirement that they be valid to a particular schema. Such stylesheets are sometimes, but not always, variants of ordinary presentational stylesheets; they include stylesheets with heuristic applications (generate a synopsis of all tables of contents for a set of documents, or display graphical renditions in SVG of a set of stylesheets) as well as more conventional presentational or interrogative applications (filter out all graphics needed by a given document or set of documents, perhaps to present this list to a shell script for copying or resizing).

Given a suitable framework (and even in a low-end solution, the XSLT `document()` function can be pulled into service for this) this kind of application can include document merger or update routines, where inputs from two or more places are assembled into a complete document.

Note that these types are not limited to the primary types of a document production workflow. XML formats can also be introduced to provide a project with calendars and time-tracking, specifications of jobs to be done, control files, and so forth. XSLT provides a way for any of these semantics to be expressed in the system through appropriately automated routines.

Note also that not every input to a stylesheet in this family needs to conform to the primary document type. For example, an "authority control" stylesheet might look up every marked-up occurrence of a name appearing in a document or set of documents, checking it against a list of names kept externally. The list of names is also held in XML, and provides a second input stream to the stylesheet. This list, however, does not conform to the schema that describes the primary input, but only to an ad-hoc schema recognized by the stylesheet.

2. Roughly typed stylesheets

Some transformations are meant to run on well-formed XML whose type is known or expected, but whose validation constraints are not necessary to the task at hand (or even expected not to apply). A simple example of this is a stylesheet used for validation. The type of document going into a validation transform is (strictly speaking) unknown, although the hope is that the transform will reveal the document to be of a particular type (by virtue of conforming to a set of constraints expressed by and checked in the stylesheet).

3. Generic stylesheets

These are stylesheets that can work irrespective of the particular tagging of a document, as generic utilities. Simple examples include stylesheets for the normalization of whitespace or document encoding or the resolution of parsed entities. (Both of these tasks can be achieved with variants of the "identity transform", a stylesheet whose output mirrors its input.)

Some generic stylesheets can be fairly far afield of actual workflow requirements, and still be useful. For example, it is not difficult to write a stylesheet whose function is to poll an input document and generate a "rough cut" stylesheet, with templates for each of the element types appearing in the input, to save the work of writing a fresh stylesheet from scratch. Because XSLT instances are themselves XML — and can thus be created or processed by XSLT (just like any other XML), myriads of such "meta-applications" of XSLT are possible.

4. Free tools review

Since XSLT is standard and openly-specified, it is quite widely supported by readily-available tools, both free and commercial. Here are a few the intrepid independent XSLT developer should know about:

- Michael Kays' *Saxon* (see [Saxon]) is probably the premier XSLT 1.0 processor, either open source or proprietary. Saxon is highly configurable, supports standard Java APIs, and provides a range of extension functions that make it quite suitable for the kinds of tasks described in this paper.

- *Apache Ant* (see [Ant]) is a "Java-based build tool", originally focussed on the compilation and maintenance of entire software applications, but quite capable of "down-grading" to handle the typically lighter-weight tasks associated with document production, such as transformation in pipelines, moving files in the file system, etc. In particular, because Ant build files are in XML, XSLT can be used to make Ant dynamic. For example, Ant can initiate a transform to poll a set of documents and write the results of the poll to an XML document, which can then be accessed by Ant again (in a dependent process) for action (such as copying files etc.)

- *XML Starlet* [XML-Starlet] is a command-line toolkit, designed to emulate UNIX tools such as `grep`, `sed`, `awk` and so forth. It includes the capability to compose transformations on the command line (though one would use it only for simple operations). In particular, it provides for writing contents of directory listings in XML: a very handy feature.

- *Apache Lenya* [Cocoon-Lenya], built on top of the popular XML server platform Cocoon, is an open-source content management system (CMS) especially useful for XML documents. It includes XSLT in its infrastructure, so XSLT developers will find it to be highly customizable and configurable. It also has direct support for simple workflows, again leveraging an XML format.

- Developers who choose to use Python should know about *4Suite* [fourSuite], an open-source toolkit for XML data in Python, which includes XSLT among many other approaches to handling XML.

Bibliography

[XSLT-1.0] Clark, James, ed. *XSL Transformations (XSLT) Version 1.0*. W3C Recommendation 16 November 1999. See http://www.w3.org/TR/xslt .

[FXSL] Novatchev, Dimitre. *FXSL -- the Functional Programming Library for XSLT*. [No date.] See http://fxsl.sourceforge.net/ .

[Piez-2003] Piez, Wendell. *XSLT for Quality Checking in a Publication Workflow*. Presented at Idealliance XML 2003. Philadelphia, PA, December 2003. http://www.idealliance.org/papers/dx_xml03/papers/04-04-02/04-04-02.html Piez document QA paper

[Saxon] Kay, Michael. *Saxon*. See http://saxon.sf.net .

[Ant] Apache Software Foundation. *Ant*. See http://ant.apache.org/ .

[XML-Starlet] XML Starlet. See http://xmlstar.sourceforge.net/ .

[Cocoon-Lenya] Apache Software Foundation. Apache *Lenya*. See http://lenya.apache.org/ .

[fourSuite] Fourthought.com. 4Suite. See http://www.fourthought.com/4Suite.xml .

Biography

Wendell **Piez**
Consultant
Mulberry Technologies, Inc. [http://www.mulberrytech.com]
Rockville
Maryland
United States of America

Wendell Piez presents frequently on XML technologies, and has published and appeared in a number of venues to academic and commercial audiences. His hands are on XSLT most days as he develops his own research projects and supports internal systems for his employer, Mulberry Technologies, Inc., a consultancy in XML technologies in Rockville MD.