illumino: An XML document production system with a TEX core

Matteo Centonza and Vito Piserchia

Abstract

XML is the state of the art in publishing technology. Publishers, through the "one source, multiple output" paradigm, are able to publish the same content to multiple media without much effort. In this paper we'll investigate current scenarios for publishers adopting a LATEX workflow and introduce illumino, our fulltext XML production system built around TEX.

1 Introduction

XML publishing in scholarly publications is nothing new. Publishers, through content/format separation, can leverage the many benefits of XML:

- Publish the same content to multiple media
- Store production data in a neutral format, the "*lingua franca*" of Internet applications
- Use XML as a neutral format for long-term archival of content
- Disseminate content through syndication
- Have content ready for data harvesting/mining (discussed in sect. 4.3)

With the term "XML publishing", we are referring to procedures and methods generating final output media from XML sources. XML sources are authored to produce final output, ready to be published. On the other hand, XML publishing is a complex task since content should be structured to be valid XML, *i.e.*:

- Encoded with correct metadata granularity
- Follow an XML grammar

XML publishing tools are often complex content management systems (CMS). Users need to perform content authoring according to tool specifications. Import tools may be provided, but imported content needs to be reviewed. This is a time-consuming task.

Publishers interested in XML publishing and adopting a LATEX based workflow, are either supposed to develop complex in-house solutions or outsource most of the publishing chain. There are many outsource facilities more or less ready to do the job but the price to pay is losing control of the work.

In this paper we'd like to present illumino, our fulltext XML production system that is trying to change this scenario. We'll present the ideas behind this technology, system capabilities and discuss future development.

1.1 illumino

illumino is a fulltext XML production system, built around (LA)TEX, which integrates international standards such as:

- DocBook 5.0
- MathML 2.0
- SVG Tiny 1.2
- Unicode 5.0

illumino converts LATEX sources to its internal XML format (DocBook) and the publishing chain, starts from XML sources.

The process is similar to the one described in the seminal article by E. Gurari and S. Rahtz [3] but uses different XML technologies. For a graphical representation of the full process, please see figure 1.

illumino is a multiplatform application built around TEX (TEX Live and the embedded TEX4ht), XSLT 2.0, Java, git (as SCM) and once configured, has native support for publisher LATEX classes and generates publishers' native production files as output. It is able to run unmodified in the old LATEX workflow.

illumino aims to integrate as smoothly as possible with any LATEX workflow, minimizing production changes to obtain fulltext XML publishing.

To achieve this goal, illumino performs automatic metadata enrichment through heuristic methods to match content granularity needed by a given XML grammar. In order to guarantee content safety while heuristically enriching unstructured information, illumino has been designed to produce output that perfectly matches that of the LATEX production source file the system is processing: we test for equal checksums of source and production output (currently PostScript output) to ensure this. When this perfect match ("*equivalence*") applies we are sure that the system has not introduced any modification to document content, so there's no need to review the article content.

illumino has embedded content checking (via SHA checksums) and the user is warned when the system outcome is not the perfect equivalence; in those cases, illumino is able to visually highlight differences found, so that visual validation can take place.

illumino is an incremental (à la Apache ant), client/server application and is able to run through the network with speed similar to that of a conventional LATEX workflow. By integrating SCM technologies, illumino can be used concurrently in a safe way. The complete list of features is given on the main illumino web page.

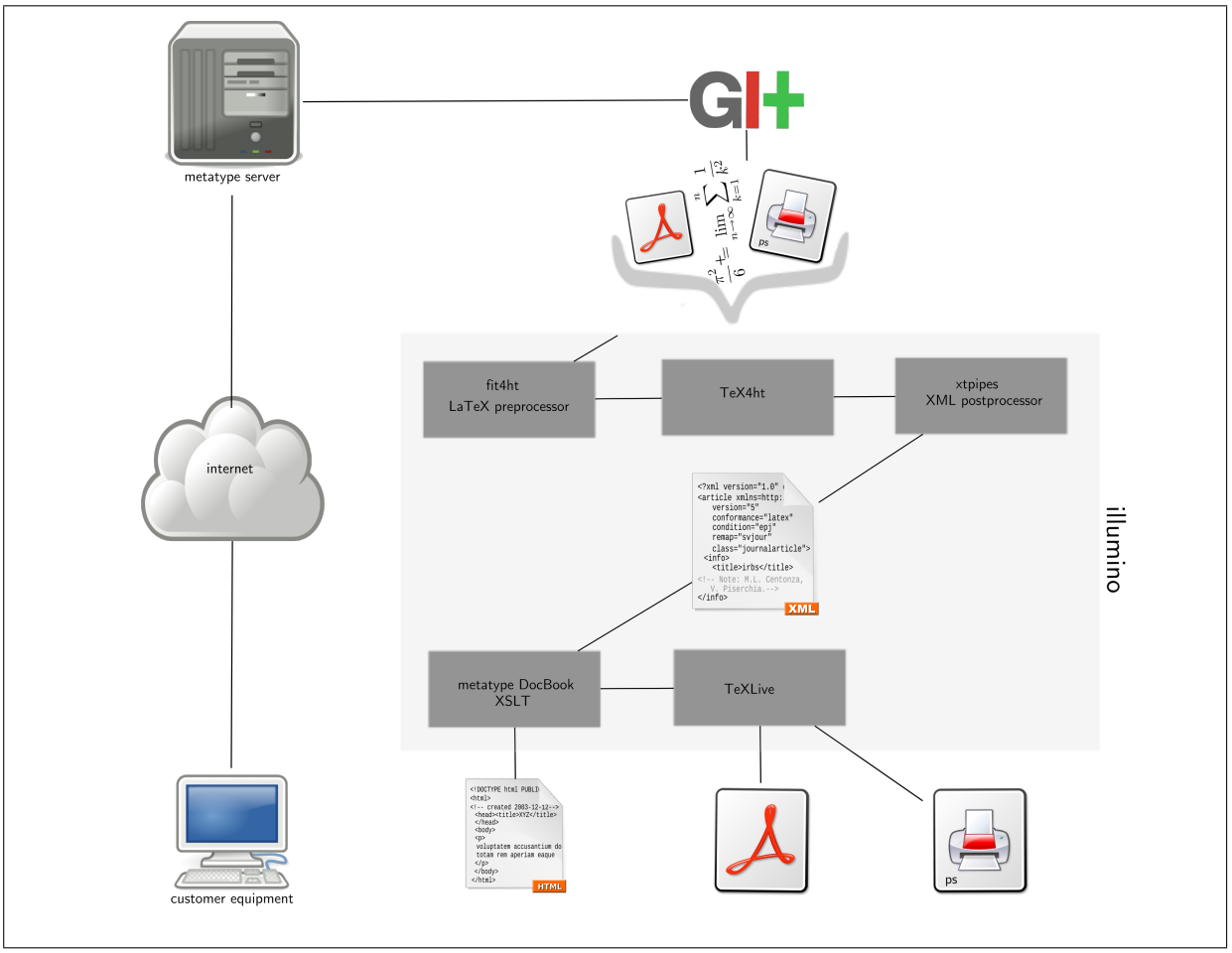

Figure 1: The `illumino` architecture

2 `illumino` architecture

Figure 1 shows current `illumino` client/server interaction. `illumino` uses standard components and implements standard and open protocols.

`illumino` has its foundations on just two main components: TEX Live and Java.

From a technical point of view, `illumino` is based on Apache `ant` and is implemented as several custom `ant` tasks, through our `illuminant` library (antlib). By using `ant`, `illumino` is an incremental (through dependencies and timestamps calculations) and multithreaded application (Java).

The system is completely standalone[1] and `ant`, used also to build the whole stack, is able to update and rebuild all upstream dependencies.

What follows is a description of high-level processes of which `illumino` is made.

2.1 fit4ht

This part of `illumino`, as its name may suggest, is responsible for making the initial LATEX source file "fit" to be run under TEX4ht. This workflow segment parses LATEX document and by using heuristic algorithms performs:

- Automatic document cleanup (*e.g.* standardize misused TEX primitives and sloppy constructs to LATEX)
- Enrich document metadata structure (split and tag content according to information patterns)
- Make some constructs ready to be correctly interpreted by TEX4ht

From a low-level point of view, fit4ht is implemented as an `ant` filterreader.

2.2 TEX4ht

TEX4ht is the heart of `illumino` and is the component taking care of LATEX to XML transformations.

[1] With the exception of the Apache Tomcat servlet container (used to implement the caching XSLT engine) and `git` SCM program.

We'll not delve into TEX4ht internals since this is out of scope for this article. For a more in-depth explanation of how TEX4ht works, the reader may refer to [2, 1].

TEX4ht's most notable difference with other similar technologies is the use of the real thing, the TEX parser, when converting a TEX file to another format.

For simplicity, we'll condense the TEX4ht workflow to three main steps:

1. Seed configurable (at the control sequence level) hooks in DVI output
2. Harvest the seeded hooks to generate a given markup representation
3. Post-process the outcome to undergo validation

We have heavily customized TEX4ht[2] mainly to:

- Implement a native backend for DocBook 5.0 output.
- Add support in the TEX4ht core for editorial fine tuning control sequences (*e.g.* supporting all tuning `toks`, vertical, horizontal, and math spaces, ...) as XML processing instructions.
- Enrich control sequence mapping in order to go from LATEX→XML and back without degradation in information quality.

By pre-processing input files and slightly modifying some TEX4ht internals, we have made the LATEX→XML conversion a completely automated process.

We have developed custom "(LA)TEX4ht compile" `ant` tasks to have automated compilation of sources. Compile reruns are handled automatically (*e.g.* TEX4ht, for complex tables have to run several times, and LATEX needs to be rerun when labels are modified).

Through TEX4ht's power and flexibility we've been able to have fine-grained content resolution and exactly remap a LATEX file into its corresponding DocBook 5.0 counterpart, producing the same output (we call it "*equivalence*" and their outputs have identical checksums).

`illumino` testcases are made of "*equivalences*" with research papers in physics from different scholarly publications. This approximately 400 pages and 30 articles test suite is `illumino`'s internal certification system and is used to avoid regressions and to spot inconsistencies in the whole `illumino` application stack (including upstream dependencies). For every build, `illumino` must pass these test cases that are constantly updated as soon as we implement new features or fix bugs.

At present, `illumino` has been tested on approximately 4k pages of content from hard sciences.

[2] Thanks to the invaluable help of Eitan Gurari.

2.3 XML transformation phase

`illumino` uses XSLT to transform the raw XML document generated by the previous phase (TEX4ht).

In more detail, `illumino`'s XML transformation phase is currently using XSLT 2.0 and takes advantages of its features, *e.g.* by using `xsl:function`, `xsl:character-map`, regular expressions and pattern matching features extensively.

The XSLT 2.0 phase must be seen as a multistage stack of stylesheets, where every filter accomplishes a different task.

XSLT stylesheets are organized in two main sets:

- `xtpipes`, an XSLT pre-phase, which takes care of space rearrangement and element positioning, and produces an enriched and valid DocBook document;
- Metatype DocBook XSLT, transforming the resulting DocBook document to all supported formats (including LATEX with publisher's class).

2.3.1 `xtpipes` stylesheets

In this stage, the filter performs:

- Space rearrangements
- Element reordering and structure enrichment
- Validation fixes

Space rearrangements are strictly related to the design decision of aiming for full equivalence with source output. LATEX and XML spaces obey completely different sets of rules in determining the output. In LATEX spaces can appear almost anywhere in the source document but may be relevant to output in only some cases; conversely, an XML grammar strictly controls the allowed spaces in the document tree.

In order to achieve "*equivalence*" between source and production output, we have handled all corner situations in which the meaning of spaces from LATEX and XML differ.

Regarding element reordering and enriching, we have to face the different nature of semi-structured and structured data. For example, in LATEX documents, many commands can change the properties of the entire group or environment when specified inside that group. Almost all the alignment commands have this behaviour (*e.g.* `\centering` inside a floating environment). On the other hand, on the XML side we have to specify this behaviour with the tag that represents the LATEX environment, with permitted attributes, if any (*i.e.* `align="center"` inside the CALS table element).

Keeping in mind that seeding of TEX4ht hooks is sequential and happens when TEX sees the commands, we have two possibilities:

- using elements and attributes suggested by the XML schema, when meaningful and close to LaTeX counterparts (*e.g.* alignment in table environments)
- using a powerful transclusion and linking technique

`xtpipes` stylesheets follows the first approach where possible and in the remaining cases reverts to using a built-in `xlink`/`xpointer` processor, implemented with XSLT function extensions.

For example, the `xpointer` scheme can be used to link other elements in the document and the `xinclude` syntax can be used to transclude from other documents.

We have been able, with our XSLT 2.0 `xpointer` implementation, to point to any other element in the document and *e.g.* change attribute values. In short, we have XSLT transformations driven by the XML content, so in the final analysis governed by TeX4ht.

When the latter method is not applicable, we resort to bare XML processing instructions to render the construct.

Validation techniques are discussed in sect. 4.1.

2.3.2 metatype DocBook stylesheets

This phase produces the supported output formats, starting from valid DocBook 5.0 sources. Leveraging XML's strengths, we can generate several output documents (*e.g.* simple text, HTML, LaTeX or documents in other XML markup languages) from the same XML source.

2.4 DocBook version 5

DocBook, developed by the OASIS consortium, is a semantic markup language for technical documentation. As a semantic language, DocBook is focused on content and meaning (DocBook has not been designed to visually format content).

DocBook offers several advantages over competing markup languages:

- Long history and schema stability
- Wide adoption and great availability of tools that support authoring of DocBook documents
- Capacity to generate output files in a wide variety of formats (HTML, XSL-FO and LaTeX for later conversion into PDF or other document markup languages), lately `epub`
- Semantic similarities with LaTeX commands
- Modular structure including widely adopted XML grammars (*e.g.* MathML and SVG)

For a more in-depth explanation of DocBook concepts, the reader may refer to [5].

2.5 `illumino-remote`

`illumino` is a client/server application built upon open protocols. `illumino` leverages SCM technologies, and the backend system exposes `git` (`http://git-scm.com/`) interfaces.

`illumino-remote`, the system client, interacts with the remote `illumino` server through the `git` protocol.

Whenever the `git` daemon receives new changesets (deltas) for a given article from a client, a new local (server) workflow run will be launched on the updated sources and results (*e.g.* XML, PDF deltas) will be sent back to the client.

Normally `git` roundtrips are very fast[3] in comparison to other SCM technologies and we are able, in combination with `ant` behind the scenes, to have `illumino` processing time be on the same order as a LaTeX workflow run.

`illumino-remote` is a Java application with JMS message passing between client and server. We are waiting for the pure Java `git` (`jgit`) implementation to mature, in order to have a pure Java client.

`illumino-remote` can control all remote backend behaviour such as:

- Repository operations (add, delete article resources)
- Enable/disable output formats
- Choose the PDF output engine (pdfTeX, Adobe Distiller, `ghostscript`)
- Show output differences[4]
- Enforce output equivalence[5]
- Choose a secondary XML output format

3 Usage caveats

`illumino` has been designed to integrate as smoothly as possible into any existing LaTeX workflow.

XML publishing, starting from unmodified LaTeX production sources, while a cost-effective way for publisher to enable a full text XML workflow, is also a complex software task. Aspects of this complexity are:

- Automatic enrichment of semi-structured content to a more structured form
- Proper separation of content from presentational elements.

What follows is a list of production caveats.

[3] Deltas (differences) for storing changesets and fast merging/indexing algorithms let `git` compete with some native filesystem operations.

[4] Visual differences are presented when the transformation does not end with output equivalence.

[5] `illumino` will fail the transformation if the result is not equivalence.

3.1 Automated content tagging

Often LaTeX sources are not sufficiently structured to permit a 1:1 mapping with the majority of XML schemata. To be able to fill all the data structures provided by an XML schema, we have to properly resolve pieces of information adhering to specific patterns. These patterns are able to take care of most of the production scenarios we have seen during the heavy test phase our product has undergone.

Out of the box, `illumino` is able to resolve correctly and to split various sparse information that in other semi-automatic systems users tag manually.

This process is by no means perfect since it is completely heuristic. In some corner cases, this approach may not be completely satisfactory and manual tagging is needed. If a new content pattern is found or highlighted, it will be added to existing filters.

In other cases, heuristic treatment is simply ineffective (such as affiliation splitting) and users must manually tag content to get the needed granularity (*e.g.* split into organization name, division, etc.).

Our long-term aim is to integrate `illumino` with the `UIMA` framework and leverage Bayesian annotators to automatically split what currently is done manually (see sect. 4.2).

3.2 Content/presentation separation

LaTeX has a plethora of commands, environments and class infrastructures which allow for a very high fraction of content separated from presentation.

Authors strictly adhering to LaTeX and class instructions will provide a very good source base to transform to XML. Unfortunately this is not always true, and non-standard environments, low level TeX code instead of standard LaTeX, TeX font primitives, etc., are easily found.

We have done our best to automatically transform non-standard code to a more conformant form, preserving its original meaning. This again will probably not cover all possible cases. In a few cases, users should manually convert the non-standard code.

3.3 XML validation

A document, to be valid according to an XML grammar, should be checked not only at the structural level but also at the element content level (*i.e.* not only how elements nest but also what elements contain).

This streamlines further processing to other formats and *e.g.* long term archiving of content (one of the most interesting parts of an XML workflow).

This (not surprisingly) comes at a cost: content sometimes should be rearranged in order to adhere to a given XML schema. The upside is that document overall quality will be increased.

In most situations, TeX4ht is able to produce valid XML documents, but some problematic cases exist. In our experiments, we have found at least two classes of problems in which validation should be refined at a later XML post-processing stage.

As already mentioned, this is due to the strict rules imposed on an XML document when compared with the weak structure imposed by the LaTeX grammar: LaTeX to XML transformation can produce XML chunks that do not fit in the XML structure (*e.g.* elements outside allowed parent).

We have solved these validation problems by using XSL context-aware `xpath` expressions, rearranging the offending chunk and folding it with the most appropriate parent element, whenever the XML schema allows this. With this approach we are able to solve most validation problems. In some remaining cases, users must resort to recoding LaTeX sources to solve validation problems; a high fraction of problems come from offending XML chunks generated from a sloppy or invalid use of LaTeX constructs.

4 "*What the future brings*"...

4.1 XML validation

Currently we validate XML documents through the Namespace-based Validation Dispatching Language (NVDL).

NVDL is able to route content coming from a given namespace in order to be validated by the correct namespace grammar. In this way, we are able (by using DocBook) to intermix content validated through DTD, RelaxNG, and XML Schema.

`oNVDL`, an open-source NVDL implementation based on `Jing`, is our choice.

In the future, we want to explore the opportunity to take advantages of other XML validation languages. In particular our attention and future efforts are focused on the Schematron validation language. By using Schematron rules we will be able to deal more easily with current validation constraints.

4.2 Improving unstructured content parsing through the `UIMA` framework

In section 2.1 we introduced `fit4ht` filters taking care of document metadata structure enrichment, information tagging and code cleanup.

`fit4ht` is a set of specialized modules taking care of enriching information structure by adding context metadata. The nature of `fit4ht` modules is heuristic: whenever document excerpts adhere to a given pattern, information can be split (safely, since "*equivalence*" or visual validation comes to help).

Since one of illumino's tasks is to treat unstructured/partially structured information to convert into a more structured form, in the long term we'll port fit4ht modules to Apache UIMA (http://uima.apache.org/).

Unstructured Information Management applications are software systems that analyze large volumes of unstructured information in order to discover knowledge relevant to an end user. An example UIM application might ingest plain text and identify entities, such as persons, places, organizations; or relations, such as works-for or located-at.

The UIMA frameworks support configuring and running pipelines of Annotator components. These components do the actual work of analyzing the unstructured information. Users can write their own annotators, or configure and use pre-existing annotators. Some annotators are available as part of the UIMA project; others are contained in various repositories on the Internet.

By integrating illumino with the framework we will be able to leverage the software ecosystem built around UIMA and *e.g.* split information based on Bayesian inference or address other editorial tasks such as normalization of inflected forms.

4.3 Knowledge mining

Another interesting field for which scientific XML content is particularly suited is *knowledge mining*.

Several advances in computer science have been brought together under the rubric of "data mining" [4]. Techniques range from simple pattern searching to advanced data visualisation and neural networks. Since our aim is to extract comprehensible and communicable scientific knowledge, our approach should be characterised as "knowledge mining".

Our idea is to create a network of links between research articles from various fields of science and accelerate research, scientific discovery and innovation.

The key point is that scientific papers, especially from the hard sciences, encode most of their content using mathematical expressions. Every mathematical expression has a unique meaning.

By indexing all occurrences of mathematical expressions present in research papers, it would be possible to build a network of links between research articles. Analyzing links between different fields of knowledge would make it possible to deduce symmetries, patterns, and even similarities that could be used as research targets.

4.4 illumino GUI

We plan to develop a graphical interface in order to have a smooth interaction with the system. This graphical interface should integrate a LaTeX editor and will handle remote interaction with the system.

In our plans, this will be done by developing an Eclipse plugin, in order to leverage the Eclipse ecosystem to have advanced functionalities such as:

- Real-time shared editing
- Context sensitive editing
- Seamless remote interaction
- Versioning and change management (à la git).

References

[1] G. Cevolani. Introduzione a TeX4ht, Proceedings of the 2004 Italian GuIT meeting (in Italian). http://www.guit.sssup.it/guitmeeting/2005/articoli/cevolani.pdf

[2] M. Goossens and S. Rahtz with E. Gurari, R. Moore, and R. Sutor. *The LaTeX Web Companion*, Addison-Wesley, 1999.

[3] E. Gurari and S. Rahtz. "From LaTeX to MathML and back with TeX4ht and PassiveTeX". http://www.cse.ohio-state.edu/~gurari/docs/mml-00/mml-00.html

[4] P. Langley and H.A. Simon. Applications of machine learning and rule induction. *Communications of the Association for Computing Machinery*, 38(11), 54–64, 1995.

[5] N. Walsh. *DocBook: The Definitive Guide*, O'Reilly & Associates. http://www.docbook.org/tdg/

◇ Matteo Centonza
metatype, Via Santacroce 13/5,
I-40122 Bologna, Italy
matteo (at) metatype.it

◇ Vito Piserchia
metatype, Via Santacroce 13/5,
I-40122 Bologna, Italy
vito (at) metatype.it