Instant XML — Stylesheet Automation

Geoff Nolan

Senior Systems Engineer
Turn-Key Systems

While much effort has been spent in enabling XML documents to be rendered in sophisticated styles, there are also times where the rapid provision of a more basic styling is required. For example, we may wish to produce a rough draft of a new document for which we do not yet have a suitable stylesheet. Alternatively, we may wish to produce a basic stylesheet for a new document type to which additional refinements can be added later.

The latest version of Turn-Key's TopLeaf rendering system is equipped with just such an automatic stylesheet generator. This process requires only a representative XML source document; no DTD/Schema or rendering hints (such as a CSS on-screen stylesheet) are required. This document outlines the methods used to derive styling from XML source, and exactly what styling features should (and should not) be supported.

Initial Aims

Our company's flagship product is TopLeaf, an XML rendering system which is designed specifically to allow non-specialists to set up and maintain stylesheets with a minimum of effort. However, even the simplest styles required some initial work to set up, mainly involving defining basic style mappings for the key elements in the source DTD.

While TopLeaf can import styles from CSS, or from other TopLeaf publications, we lacked a method for setting a reasonable initial styling for new document classes where no suitable base stylesheet existed. We therefore decided to add a facility whereby a complete (but basic) stylesheet could be generated using only an XML source document for input.

We set out to produce a stylesheet generator which:

- handles any well-formed XML source document, irrespective of whether a DTD/Schema is available;
- produces (at least) one mapping per element type [for this purpose a mapping can be defined as an association between an element and a set of rendering actions, similar to a CSS rule];
- is responsive to how an element is actually used in a document, as opposed to where it fits into an abstract model;
- applies sufficient styling to make the output readable, without adding complexities which may lead to problems later on.

Our first decision was whether to use DTD/Schema information if available, or rely purely on the source document for styling information. We decided that DTDs should be (largely) ignored since

their limitations, particularly the lack of ability to define elements in context, often lead to general definitions which fail to capture how an element is actually used.

While Schemas do not generally suffer from the same limitations as DTDs, the programming effort required to accurately extract styling information from the large, complex and continually evolving collection of Schema types does not justify the relatively modest benefits which would result.

The one piece of really useful information that cannot always be obtained from analysing the source document is the proper resolution of entities. It is helpful to know that entities such as **`&mdash;`** and **`&company-name;`** resolve to text strings. It is even more useful to be able to replace entities such as **`&section3;`** or **`&disclaimer;`** with the contents of an XML subdocument.

So for generating styles the TopLeaf parser is set up to use a DTD for entity resolution if possible, any unresolved entities are simply ignored. Likewise the parser discards comments, processing instructions, attribute values, and the content (but not the presence) of text fragments. What we are left with is a document skeleton which contains only tags, plus markers indicating the presence of (non-whitespace) text at that location.

The following section details how this skeleton is processed to assign styling rules for every element.

Document Analysis and Stylesheet Generation

In order to generate a stylesheet, we have to assign to each tag a role to which a style can be attached. Moreover these roles have to be determined without any prior knowledge of the semantic significance of each tag, without analysing textual or attribute content, and without even a DTD or Schema to assign content models.

The following sections explain how this is achieved.

Extracting element properties

An important step in developing the stylesheet generator was deciding which properties of elements in the source document would yield the most cues to styling. Available information includes:

- ancestor and descendant elements;
- textual content;
- attributes and their values;
- depth within the document hierarchy;
- position within parent;
- presence of text in/around the element.

Perhaps surprisingly, the first three of the above points contain very little useful information and are largely ignored. The remaining points are noted, and after the whole document (including any available subdocuments) has been parsed, the information is distilled into the following basic properties for each element:

- **maximum-position** — the maximum position of any occurrence of the element within its parent (relative to other sibling elements).

- **absolute-depth** — the minimum *possible* depth the element could occupy in the hierarchy. Note that this can be less than the minimum *actual* depth of the element in the source document.

  For example if a **`footnote`** occurs only within a **`para`** inside a table cell (say the para is 9 levels deep) then the footnote will have a *minimum* depth of 10. However, if other **`para`** elements occur much higher up, say at level 3, then the *absolute* depth of **`footnote`** will be 4. This is because the stylesheet generator assumes that the footnote *could* appear within one of these higher level paras in other documents.

  More formally, the absolute depth of the outermost (DOCTYPE) element is 1, while the absolute depth of any other element is 1 greater than the minimum absolute depth of any of its parents.

- **has-kids** — true if the element has any element children.
- **has-text** — true if the element contains text directly (ie. outside any child element).
- **within-text** — true if the element and text both occur directly within the same parent.

Having established the above properties for each element in the source document, we can apply specific styles as explained in the following sections.

Classifying elements by property

We can assign each element to a specific role based on properties by applying the following rules in order:

- **principal** — if absolute-depth is 1;
- **titleN** — if maximum-position is 1, has-text is true and within-text is false;
- **empty** — if has-kids and has-text are both false;
- **structural** — if has-text is false;
- **inline** — if within-text is true;
- **container** — otherwise.

Note that **title** elements can be further subdivided according their absolute-depth:

- **title1** — if absolute-depth is 2;
- **title2** — if absolute-depth is 3;
- **title3** — otherwise.

Now that we have a functional classification for each element, we can assign a particular style to each class.

Assigning styles

Now that we have assigned a role to each tag in the source document, we need to generate a corresponding stylesheet. Fortunately, we only have to assign a style to each role once. We can then use these standard styles for any source document.

We can determine appropriate styling based on the significance of each role as follows:

- **principal** — used to set various defaults (eg. font and size), otherwise just a *block*.
- **structural** — these are the skeleton of our document, but since they do not contain text directly they are simply neutral *block*s.

- **container** — represent the transition from element to mixed content; in other words these are the basic paragraph type and are styled as *block*s with appropriate *inter-paragraph space*.
- **inline** — used to delimit particular portions of paragraph text, so use *italic* font for emphasis.
- **empty** — could represent anything from a graphic to a link destination; since there is no suitable default format these are represented by inline insertion of the element name, say emptyTag, inside a small frame.
- **titleN** — a special paragraph with *bold* font for emphasis, plus changes to font *size* and *space above* depending on the level **N**, and a *bind to following paragraph* so the title isn't separated from its text.

Note that there are two additional roles which can only be assigned via heuristics (see following section):

- **table** — used to set CALS/HTML table elements.
- **list-item** — used to set list items.

Heuristics — filling in the gaps

While the general styling rules will usually give a pretty good rendition of an arbitrary document, there are some elements which require special handling. To this end, the stylesheet generator allows the specification of a number of heuristic rules. These are simply "educated guesses" based on the normal appearance of certain nominated tags.

The first set of heuristics concerns table tags. If the generator sees a hierarchy which resembles a CALS (**`tgroup/row/entry`**) or HTML (**`table/row/td`**) table, then it will treat all relevant elements accordingly. This behaviour can be disabled where not appropriate.

The next heuristic set identifies list items. For example, any tag named **`item`**, **`litem`** or **`list-item`** with a role of **structural** or **container** will be assigned a new role of **list-item**.

Finally there are set of miscellaneous rules to cover particular situations. For example the tags **`title`** and **`heading`** will normally be assigned the **title** role irrespective of their position.

What comes out?

The final design decision concerned the output of the stylesheet generator. We decided that CSS was the best choice because:

- It is a standard format which is also human readable. This allows the output to be examined in detail during development and testing.
- It can be imported directly by TopLeaf in a process which is transparent to the end user.
- It has the potential to be leveraged to display source material directly via the Web.

So a TopLeaf user setting up a new class of documents is given the following choice:

- import the style from an existing TopLeaf publication
- import an existing CSS stylesheet
- auto-generate the style from the source document

If the last option is chosen the source document is processed and the generated CSS stylesheet automatically imported. The whole process typically takes about 5 seconds, after which the document is ready to print.

Could we and Should we?

You might ask: *Surely we can do even more with more complex algorithms?* Well, we certainly can. A few of the things which could be added quite easily are:

- use different fonts and maybe underlining for text and titles
- attempt to resolve graphic references
- analyse tags in context, so for example **`chapter/title`** could map quite differently to **`subsec/title`**

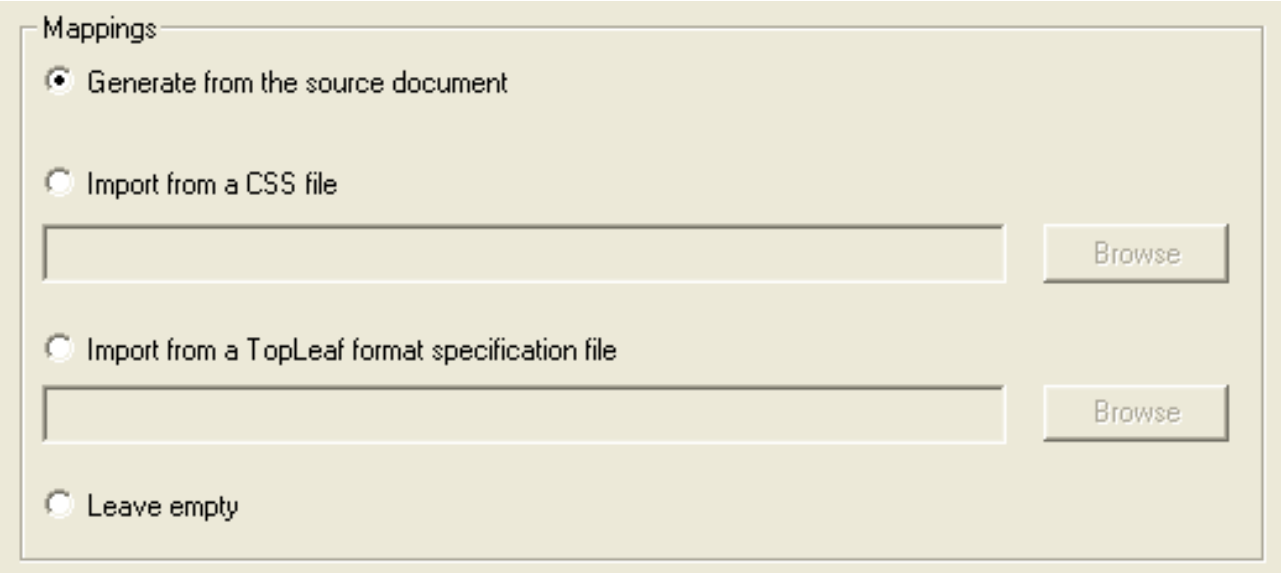
However the real question here is not whether we *could* do this, but whether we *should*. Consider the implications of the "enhancements" above:

- any attempt to perform changes of typeface or effects makes a lot of extra work if the style does *not* require such changes (as is often the case);
- graphics references are often non-trivial and the chances of getting such factors as search path, scaling, resolution, rotation etc. all correct is fairly small — better to let the user set up the graphics mappings properly in the first place;
- mapping parent/child pairs would produce a large number of additional mappings, many of which would be unnecessary and many more simply wrong — a basic mapping for each tag is less confusing, and often a lot less work, for the user.

In practice, adding new features rapidly runs into a region of diminishing returns, where the costs of implementing, maintaining, and in many cases undoing the more sophisticated typesetting effects far outweighs the benefits gained.

Case Study: Through the looking glass

In searching for s suitable case study, it occurred to me that the XML for this paper contained just about every type of element, except for an empty tag and a table. So I've included them here. The graphic is a screenshot of the TopLeaf publication creation dialog . . .



And the following table lists some of the tags in this document along with their significant properties, and the role they were assigned (heuristic assignments are marked thus*).

Element	abs-depth	max-pos	has-kids	has-text	within-text	Role
abstract	3	3	Y	N	N	structural
author	3	1	Y	N	N	structural
book	1	1	Y	N	N	**principal**
chapter	2	6	Y	N	N	structural
emphasis	4	16	N	Y	Y	*inline*
graphic	3	2	N	N	Y	EMPTY
itemizedlist	4	8	Y	N	Y	structural
listitem*	5	6	Y	N	N	• list-item
markup	4	14	N	Y	Y	*inline*
para	3	7	Y	Y	N	container
table*	3	1	Y	N	N	`table`
title	2	1	N	Y	N	**title1**

Stylesheet Enhancement

Once you have set up your new publication, you will use the normal TopLeaf facilities for fine-tuning the style, and adding features not handled by the stylesheet generator. These include page headers and footers, advanced font effects, paragraph indents, alignment and bindings, multi-column pages, footnotes, table of contents, indexes etc.

Overleaf you will find the first and last pages of this document, exactly (except for the watermark) as rendered by TopLeaf using only the auto-generated stylesheet. While this does highlight the fact that the styling algorithms don't always get it right, the output shows that we can have a useable PDF ready within a couple of minutes of receiving XML source for a new class of document.

Refining the style, from that of the sample pages to the finished document you are reading now, took about an hour. The document can then be output as hard copy, PDF, a web page hierarchy, or a fully formatted MS-Word file.

Instant XML — Stylesheet Automation

Geoff

Nolan

Senior Systems Engineer

Turn-Key Systems

While much effort has been spent in enabling XML documents to be rendered in sophisticated styles, there are also times where the rapid provision of a more basic styling is required. For example, we may wish to produce a rough draft of a new document for which we do not yet have a suitable stylesheet. Alternatively, we may wish to produce a basic stylesheet for a new document type to which additional refinements can be added later.

The latest version of Turn-Key's TopLeaf rendering system is equipped with just such an automatic stylesheet generator. This process requires only a representative XML source document; no DTD/Schema or rendering hints (such as a CSS on-screen stylesheet) are required. This document outlines the methods used to derive styling from XML source, and exactly what styling features should (and should not) be supported.

Initial Aims

Our company's flagship product is TopLeaf, an XML rendering system which is designed specifically to allow non-specialists to set up and maintain stylesheets with a minimum of effort. However, even the simplest styles required some initial work to set up, mainly involving defining basic style mappings for the key elements in the source DTD.

While TopLeaf can import styles from CSS, or from other TopLeaf publications, we lacked a method for setting a reasonable initial styling for new document classes where no suitable base stylesheet existed. We therefore decided to add a facility whereby a complete (but basic) stylesheet could be generated using only an XML source document for input.

We set out to produce a stylesheet generator which:

- handles any well-formed XML source document, irrespective of whether a DTD/Schema is available;
- produces (at least) one mapping per element type [for this purpose a mapping can be defined as an association between an element and a set of rendering actions, similar to a CSS rule];
- is responsive to how an element is actually *used* in a document, as opposed to where it fits into an abstract model;
- applies sufficient styling to make the output readable, without adding complexities which may lead to problems later on.

Our first decision was whether to use DTD/Schema information if available, or rely purely on the source document for styling information. We decided that DTDs should be (largely) ignored since their limitations, particularly the lack of ability to define elements in context, often lead to general definitions which fail to capture how an element is actually used.

Case Study: Through the looking glass

In searching for s suitable case study, it occurred to me that the XML for this paper contained just about every type of element, except for an empty tag and a table. So I've included them here. The graphic is a screenshot of the TopLeaf publication creation dialog . . .

GRAPHIC

And the following table lists some of the tags in this document along with their significant properties, and the role they were assigned (heuristic assignments are marked thus*).

Element	abs-depth	max-pos	has-kids	has-text	within-text	Role
abstract	3	3	Y	N	N	*structural*
author	3	1	Y	N	N	*structural*
book	1	1	Y	N	N	*principal*
chapter	2	6	Y	N	N	*structural*
emphasis	4	16	N	Y	Y	*inline*
graphic	3	2	N	N	Y	*empty*
itemizedlist	4	8	Y	N	Y	*structural*
listitem*	5	6	Y	N	N	*list-item*
markup	4	14	N	Y	Y	*inline*
para	3	7	Y	Y	N	*container*
table*	3	1	Y	N	N	*table*
title	2	1	N	Y	N	*title1*

Stylesheet Enhancement

Once you have set up your new publication, you will use the normal TopLeaf facilities for fine-tuning the style, and adding features not handled by the stylesheet generator. These include page headers and footers, advanced font effects, paragraph indents, alignment and bindings, multi-column pages, footnotes, table of contents, indexes etc.

Overleaf you will find the first and last pages of this document, exactly (except for the watermark) as rendered by TopLeaf using only the auto-generated stylesheet. While this does highlight the fact that the styling algorithms don't always get it right, the output shows that we can have a useable PDF ready within a couple of minutes of receiving XML source for a new class of document.

Refining the style, from that of the sample pages to the finished document you are reading now, took about an hour. The document can then be output as hard copy, PDF, a web page hierarchy, or a fully formatted MS-Word file.