3

Elements of a web site

River and bridge and street and square
    Lay mine, as much at my beck and call,
Through the live translucent bath of air,
    As the sights in a magic crystal ball.

ROBERT BROWNING, *Old Pictures in Florence*

3

Elements of a web site

This chapter is a practical complement for Chapter 2, "The source definition." Having discussed the ins and outs of building a comprehensive and useful source definition, we'll now look at how these rules can be applied to real-world source XML documents of a typical web site.

I cannot claim to cover everything: Your web site may well contain unique elements that won't fit common schemes. Here, only the most general and frequently used constructs are covered, and the approaches described in this chapter may not be optimal for all situations. Many examples are given, but rather than copy them over, try to use the reasoning behind these examples to analyze your own constraints and requirements.

The first part of the chapter deals with markup constructs commonly used in page documents, including headings, paragraphs and paragraph-like elements, links, images and other non-XML objects, tables, and forms. Then we will analyze the master document (**3.9**) to find out what data it needs to store and what is the best XML

representation for this data. The last section (3.10) presents complete summary examples of a page document, a master document, and a Schematron schema to validate them.

3.1 Page documents: top-level structures

In this and subsequent sections, we look at the informational core of a web page, stored in its own source document (*page document*). Peripheral components such as navigation, parameters of the site environment, and metadata are stored in the *master document*, the subject of 3.9.

3.1.1 Page metadata

Every XML document has a root element, and since we're talking about page documents here, there's no reason not to call this element `page`. Its attributes and children are the natural place to store the page's *metadata*.

In addition to its primary content, each page document includes certain metadata. Some of it may end up as a visible part of the web page, some may be hidden in HTML metadata constructs (keywords and descriptions in `meta` elements), some may be used during transformation but not included in the resulting HTML code, and some may not be used at all except for reference or source annotation purposes. Common examples of metadata include page creation date, change log, author(s) and editor(s), copyright and licensing information, and the language of the page.

Note that only information specific to this particular page must be stored in it; if some metadata bits are shared by more than one page, their proper place is in the master document (3.9) and not in any of the page documents.

Page ID. The most important piece of metadata is the page's unique identifier used to resolve internal links (3.5.3). However, we cannot store this identifier in the page itself, or we'll have a catch-22 situation: We can get from the `id` to the page location, but to obtain the `id` we

must access the page — that is, we must already know its location. Because of this, the proper place for the `id`s of all pages is in the site's master document.

Page coordinates. The same applies to the information on the position occupied by this page in the site's hierarchy. As we'll see later (**3.9.1.1**), the branch of the site's tree that this page is a leaf of is most naturally deduced from the site directory in the master document. Duplicating this information in the page document itself is unnecessary and prone to errors.

Everything else. Any other page metadata is normally stored in the page document. Simple values can be stored in attributes of the page's root element. More complex constructs that require their own elements can be placed either directly under the root or inside an umbrella parent element (e.g., `metadata`) that is a child of the root element.

Existing vocabularies. RDF (**1.1.5**), besides being the cornerstone of the Semantic Web, can be used as a powerful tool for representing metadata in the traditional Web. It allows you to use standardized descriptors for common values such as author and date, but you can just as well define your own semantics for your unique metadata.

3.1.2 Sections and blocks

You'll likely need some intermediate structural layers between the root element, `page`, and text markup constructs such as paragraphs and headings.

Sections or blocks? The traditional document hierarchy — sections, subsections, subsubsections, and so on — is not often seen on the Web. Instead, information is more commonly broken into relatively small *blocks* with few or no hierarchical relations between them. Different sites may call these blocks "stories," "blurbs," "columns," "modules," "writeups," and myriad other names.

Among these names, one which appears to be the most intuitive is the one you should use for your block construct's element type name. Contentwise, a block is a unit with mostly fixed structure that may

include both obligatory (e.g., heading and body) and optional (e.g., icon, heading links, author byline) components in both parallel and sequential (**2.3.7**) arrangements. Here's an example:

```
<block id="unique" icon="block_icon" type="story">
  <head link="address">Block heading</head>
  <subhead>Optional block subheading</subhead>
  <p>And here goes a paragraph of text.</p>
  <p>Possibly one more paragraph.</p>
  <author>An optional author byline</author>
</block>
```

Block types. It is likely that you will have more than one type of block construct — for example, front page news blocks, subpage body blocks, and ad blocks. In the simplest case, everything on the page can be treated as one big block, so the page's root element can be considered the root element of a block.

Different types of blocks will likely have many common structural features — in part because they all belong to one site with its common information architecture and visual design. Only if different types of blocks have clearly distinct structures can you use different element types for them; otherwise it is best to use the same generic element type (e.g., `block`) with different values of the `type` attribute. This provides two major benefits:

- Your validation code will be simpler to write and maintain.
- Management of orthogonal content (**2.1.2.2**) will be much easier to implement — for example, you may be able to reuse `blocks` from regular pages as orthogonals on other pages, or turn documents storing orthogonal content into regular pages.

In general, analogous but different structures should only differ by a minimum number of obvious features; avoid random, meaningless differences.

3.2 Headings

Brief highlighted text fragments that preface or summarize longer pieces of text are very common on web pages. A heading may apply to the entire page, a section within the page, or even a single sentence or link — but it must apply to something, for a heading only exists as a member of a "head and body" pair.

3.2.1 Element type names

Look up the number. HTML has long used the `h1` to `h6` element types for six levels of headings. You can borrow these names, or you can make them less cryptic by using `head1`, `head2`, and so on. In any case, this approach only works if you really need several levels of headings and if these levels are free of any additional semantics — that is, if you can more or less freely move a branch of your headings tree upward or downward in the hierarchy.

If this is not true — for example, if your third-level headings are reserved specifically for sidebars that cannot be promoted to second-level sections — then the number-based naming scheme is not a good idea at all. Imagine that one day you need to add sections *inside* a sidebar — this will look ugly if your sidebar headings are marked up, say, as `h4`, while sections are `h2`.

Ask my parent who I am. It is vastly more convenient to use some descriptive names, such as `chapter-head`, `section-head`, or `sidebar-head`. An even better approach is to take advantage of the "head and body" duality mentioned above. If you've defined different element types for the complete structural units (`section`, `sidebar`, etc.), then the single `head` element type can be used for headings at any level:

```
<section>
  <head>This is a section heading</head>
  ...
  <subsection>
    <head>And this is a subsection heading</head>
    ...
  </subsection>
</section>
```

This scheme is intuitive, easy to remember, and therefore easy to use. Even though there is only one heading element type, XSLT or Schematron will have no problem determining the role of each particular heading by checking its parent element. At the same time, implementing processing that is common to *all* headings is very straightforward with this approach.

XHTML 2.0[1] implements a similar scheme except that its element type for a heading, `h`, is always a child of a `section` (although `section`s can nest). This is understandable — XHTML cannot realistically cover all possible kinds of structural units that might require headings. On the other hand, this brings us back to an "anonymous" naming scheme that is only slightly better than the old `h1...h6`: Now you can easily move sections around with their headings, but still no useful semantics is attached to each heading. You can, however, use the CSS `class` attribute to designate exactly what kind of a heading or section this is.

3.2.2 Attributes

The next question is, what is the auxiliary information to be stored with your headings?[2] In most cases, the plain text of the heading itself is sufficient, but there are exceptions. For example, a heading usually has a unique (either within the page or, more usefully, within the entire site) `id` attribute used in cross-references or hyperlinks to this section from elsewhere.

1. `www.w3.org/TR/2003/WD-xhtml2-20030131/`
2. Formatting attributes such as font, color, or numbering style are out of the question — the whole point of semantic XML is that these must be abstracted away.

In fact, a typical reference is supposed to refer to the section (or other structural unit) to which the heading belongs, not to the heading itself. Still, most authors prefer to use headings for linking, partially due to the HTML inertia (there are no `section`s in today's HTML) and partially because this allows them to more easily reuse the text of the heading in the textual part of the link.

For example, if your heading is marked up as

```
<head id="attrib">Attributes</head>
```

and you have a reference to it from somewhere, written as

```
...see <link to="attrib"/> for more on this.
```

this can be easily transformed into

```
...see 2.1, "Attributes" for more on this.
```

in plain text, or to

```
...see <a href="#attrib">2.1, Attributes</a> for more on this.
```

in HTML (here, "2.1" comes from an automatic count of preceding and ancestor sections). On the other hand, given that XSLT can easily traverse from a heading element to its parent, there's no real reason to use headings for linking in XSLT-based projects. The same link rendering could just as well be obtained from

```
<section id="attrib">
  <head>Attributes</head>
  ...
</section>
```

which looks less tautological and better reflects the fact that both the `head` and the `id` are properties of the `section`.

If necessary for your site's design, you may need to store a reference to a graphic file for each heading (see **3.6** for a discussion of image references), but only if the correspondence between headings and images is not automatic. The image may be used, for example, as a background or an icon-like visual alongside the heading.

3.2.3 Children

The question of what children to allow within headings boils down to the question of how far beyond plain text you are willing to go. Would you need textual emphasis within headings? What about links? The laziest approach is to allow everything that is allowed within a paragraph of text — and it will work fine in most cases. Only if you think you may encounter problems with complex markup in headings and want to guard against them, might a different content model for headings be necessary.

Depending on your requirements, other children may be necessary for heading elements. For example, you may want to store the same heading in two or more languages, with the stylesheet selecting one of the languages for presentation depending on a global language parameter (see also **2.3.5**).

You may also want to keep both full and abridged versions of a heading. For example, newspapers often use a specific abbreviated English syntax for their headlines (as in *U.S. Patriot Act attacked as threat to freedom*), but for the purposes of automatic indexing and natural language processing, the fully grammatical version of the same heading might be required (*The U.S. Patriot Act was attacked as a threat to freedom*).

3.2.4 Web page title

A special kind of a heading specific to HTML documents is the `title` of a page, normally displayed in the title bar of a web browser window as well as in bookmarks or search results listing this page. Even though, as a general rule, your target vocabulary must not influence your semantic source vocabulary, you should plan ahead as to what source element(s) will be transformed into the web page title.

If each of your pages has a visible on-page heading that applies to the entire page, it is natural to duplicate it as a web page title. Otherwise, it is always a good idea to provide a heading for *any* sufficiently large information unit. Even if in your target rendition this heading will

only be used for a peripheral element such as page title or not used at all, the very act of christening a piece of data disciplines your thinking and serves as an additional checkpoint to ensure the consistency of your source's information architecture. Besides, the title may turn out to be more important for other renditions of the same source document.

Multistage titles. A web page title is often used for orientation within the site. A sequence of parent sections' headings, culminating in the name of the entire site, may be appended or prepended to the current page's title (e.g., "Foobar Corporation — Products — Foobar Plus"). Such a hierarchical title may be informative and useful, especially with deep site trees (even if the same information is duplicated on the page itself). Of course, it is the stylesheet that builds such a compound title, while the XML source of each page only provides that page's unique part of the title.

3.3 Paragraphs

A paragraph is a sequence of sentences that traditionally represents a complete, single thought. Today, however, paragraphs are often used for structuring the text flow visually, rather than for organizing the flow of ideas within it. Online, paragraphs tend to be smaller than in print, and other means of text organization (such as blocks, **3.1.2**) may make traditional paragraphs less common.

Still, whenever you have a container for more than just a small bit of text, your schema should permit inserting one or more intermediate paragraph elements between this container and its text. In most cases, this intermediate level may be optional; for example, your block elements could be allowed to contain either direct text content (for short fragments less than a paragraph) or a sequence of paragraph elements (for longer pieces of text). This approach adds a degree of laxity to your schema but is very convenient in daily markup practice.

As for the element type name, there is no reason not to use HTML's `p`, although `para` would be more appropriate for users who might find `p` too cryptic.

3.3.1 Lists

Lists are a special construct that is closely related to paragraphs. Two common types of lists offered by HTML are unordered (bulleted) and ordered (numbered), differing in how the items in the list are adorned. For our XML markup, we could borrow HTML's model, with a parent element (e.g., `ordered-list`) enveloping the entire list and children elements (e.g., `item`) marking up individual items.

The only possible ambiguity with regard to list markup is how to correlate list items with paragraphs. Often, each list item is a paragraph, so you may be tempted to consider paragraph elements redundant and disallow them from list markup completely. However, as soon as you run into an item of two or more paragraphs, you may regret this decision. I recommend using the convention discussed in the last section: Allow *both* paragraphs and direct text content within list item elements.

In fact, this is what is implemented by HTML 4; its `li` element can contain both inline and block content (i.e., both text children and paragraph elements, among others).

3.3.2 Paragraphs as link targets

Most links refer to entire web pages, but sometimes you need to pinpoint a particular location within a page. In HTML, you can make a link target from as small a piece of text as you like, down to a single sentence or word (by enclosing it in an `a` element with the `name` attribute).[3] In most graphic browsers, however, the only visible result of jumping to an in-page link is the page being scrolled down so that the linked point is at the top edge of the window.

This means that — unless your linked sentence happens to start at the beginning of a screen line — the visible portion of the newly loaded page starts in mid-sentence. This result is quite confusing and

3. A linked element is often called an *anchor*, and HTML uses this term for both the source of the link (*source anchor*) and its destination (*destination anchor*); hence the use of the `a` element for both ends of a link.

makes it nearly impossible to guess what *exactly* the link referred to. For this reason alone, it is advisable to only allow anchoring links to block-level elements, including paragraphs.

With XML, it is easy to enforce that rule because you most likely won't have any target enveloping element (like `a` in HTML). What you need instead is an attribute, only applicable to block-level elements, that turns its element into a link target. It often makes sense to reuse the almost-standard `id` attribute for this purpose (in addition to its numerous other uses); it won't do any harm if some of the elements with `id`s will create HTML link targets but will never be linked to.

3.3.3 Displayed material

Sometimes, you'll need to present an object that breaks the paragraph flow, but doesn't necessarily start a new paragraph. Often, this is a mathematical formula or a programming code fragment that must start on a new line.

Such a piece of *displayed material* is a block-level element from an HTML perspective; semantically, however, it is often an inseparable part of an adjacent paragraph containing the introductory or explanatory text for this displayed item. Therefore, it makes sense to allow the displayed material elements to be used only as children of paragraph elements.

3.4 Text markup

The "HT" in HTML stands for *HyperText*, and the early historical Web was very much textual. Despite all the graphic and multimedia advances of recent years, this textual foundation has not eroded. The advance of XML has, if anything, only strengthened it.

Any text markup language must provide a sufficient inventory of markup constructs for in-flow text fragments that for some reason must be differentiated from their context. Examples of such fragments include emphasized words or phrases, names or identifiers, quotes, and foreign language citations.

Block and inline elements. HTML (as well as other presentation-oriented vocabularies, for instance XSL-FO) differentiates between *block-level* and *inline-level* objects. This distinction has to do mostly with visual formatting, as block-level elements are supposed to be stacked vertically, while inline elements are part of the horizontal flow of text.[4] Therefore, it is not really relevant for your semantic XML markup, which must reflect content structure, not formatting. Still, since HTML is your primary target format, the block/inline distinction may sometimes have repercussions for your source definition.

Thus, it may be difficult to handle situations where a source element that normally transforms into an inline-level target element has to apply to a larger fragment of a document (**3.4.3**). From the XSLT viewpoint (**4.5.1**), block-level elements are more often generated by pull-style trunk templates, while inline-level elements are the exclusive domain of push-style branch templates.

Existing vocabularies. DocBook[5] is an established standard dating from 1991 that is used mostly for technical books and documentation. It may well be the most widely used XML vocabulary after XHTML; when somebody tells you, "My documents are in XML," chances are it's actually DocBook. Software support for this vocabulary is also quite good.

DocBook is vast but not too deep, so it is simple to learn despite its large number of element types (epigraphs, bibliographies, programming code, glossaries, and so on). If you don't understand what a particular element type is supposed to do, probably you don't need it (yet). For those constructs you do need, however, DocBook may be a rich source of text markup and structuring wisdom.

TEI[6] (Text Encoding Initiative) is an older and bigger beast, developed for markup of all kinds of scientific and humanities texts. Compared to DocBook, it is focused more on low-level text markup than on high-level book structures. The TEI DTD offers many modules that cover everything from verse to graph theory, so it is highly

4. In Western writing systems, of course.
5. `www.oasis-open.org/specs/docbook.shtml`
6. `www.tei-c.org`

recommended if you need to mark up specialized texts. The *TEI Guidelines*[7] is a very comprehensive and detailed guidebook explaining the use of the TEI DTD as well as many finer points of marking up complex text constructs.

3.4.1 Mark up the meaning

Your source XML must be semantic; that is, it must reflect the meaning of text-level constructs, not their presentation. The `em` and `i` element types, both present in HTML, provide a canonic illustration of this principle. While an `i` element dictates using an italic face in visual media, an `em` only designates an *emphasis*, which is a semantic concept rendered differently in different media. For example, a fragment of text inside `em` can be set in italic in a graphic browser, but it can also be highlighted in a text-mode browser or read aloud emphatically by a speech browser.

Modern HTML deprecates `i` and other presentation-oriented element types; instead, you are supposed to use appropriate semantic element types such as `em`, possibly in combination with CSS. In your XML source, however, deprecating anything is not an option — you have to make sure that with your schema, no presentation-oriented markup is possible at all. Formatting hints (**3.6.2**) can only be used in your XML when absolutely unavoidable.

3.4.2 Rich markup

The same visible formatting may result from different source markup. For example, you may use the same italic font face for both emphasis and citations, but they must be marked up differently in your source. What only a human reader can distinguish in the formatted result should, ideally, be automatically distinguishable in the source.

In general, semantic markup in the source should be richer and more detailed than the resulting HTML markup after transformation. For example, it is often a good idea to use special element types to mark

7. `www.tei-c.org/P4X/`

up all dates, person names, or company names in your source, even though in the resulting web pages they are not formatted in any special way.

Why mark up what you don't need right here and now? Because your XML source is more than just an undeveloped (as in "undeveloped film") version of the web site. Rather, it is the start of a project that will keep growing and changing, sprouting new connections and renditions over time. For example, you may want to reuse your web site material in PDF brochures, interactive CDs, archival and search applications, and more.

This means that your XML source must be able to serve as the semantic foundation not only for your current site but also for everything it can potentially become. You may not need any extra markup right now, but it may come in very handy when you extend your site or reuse the source documents for anything beyond the web site pages.

Imagine that one day you need to convert all dates on your site from one format to another (e.g., from `MM/DD/YY` to `DD/MM/YY`). Dealing with dates scattered in the text is so much easier if all of them are marked up consistently — for example,

```
... which happened on
<date><month>09</month><day>04</day><year>2003</year></date>.
```

instead of simply

```
... which happened on 09/04/2003.
```

With rich markup, you can change dates' rendition (e.g., reorder date components or use a different separator character) without touching the source at all, simply by modifying the stylesheet.

On another occasion, you may decide to paint all company names (or only your own company's name) green on your web pages. Or, you may find it a good idea to automatically compile an index of all persons' names mentioned on your site. All of these tasks are only possible if your source XML has these elements consistently and unambiguously marked up.

The need for rich text markup obviously depends on the quality, value, and planned longevity of your material. You don't need rich markup for short-lived stuff, but if you want your material to remain useful in the long term, you should always try to think in terms of "what markup is perfect for this content" rather than "what markup is sufficient for the task at hand." Examples of long-lived or otherwise valuable content include standards, specifications, historical texts, etc.

Existing vocabularies. As an example (and a good source of ideas), consider NITF[8] (News Industry Text Format), which is a standard vocabulary for rich markup of news stories. Only a necessary minimum of NITF markup may be used in a story that goes directly to press; however, for exchange, syndication, or archival use, a complete enriched NITF markup is required. A properly prepared NITF news story uses rich markup to answer questions such as *who* the story is about, *when* and *where* the described event occurred, and even *why* it is considered newsworthy by the story author.

3.4.3 Transcending levels

The text elements we've discussed in this section would be termed *inline* in HTML, meaning they are only allowed within block elements such as paragraphs. However, this limitation does not always make sense. For example, a rich markup element such as emphasis may need to be applied to more than one complete paragraph.

Usually, this is an indication that these paragraphs constitute some logical entity, such as a quotation, which (rather than the emphasis itself) you need to mark up. However, there may be situations where no such element exists, but inline text markup still has to spread across one or more block elements. What are we to do in such cases?

Inserting a separate inline markup element within each paragraph is the least elegant solution:

8. www.nitf.org

```
<p><em>This is the first paragraph using emphasis throughout.</em></p>
<p><em>And this is the second emphasized paragraph.</em></p>
```

This leads to unnecessary duplication of markup, poor maintainability, and just plain ugliness. This is the only option, however, if your emphasis spans one paragraph *and a half*.

The simplest approach is to just do away with the inline/block distinction and allow any text markup to be applied at any level of the hierarchy, both below and above the paragraph level. This will allow you to enclose all affected paragraphs into a common parent element specifying emphasis:

```
<em>
<p>This is the first paragraph using emphasis throughout.</p>
<p>And this is the second emphasized paragraph. Note that we can use
<em>nested emphasis</em>.</p>
</em>
```

This might make sense, especially in contexts where you want to allow both paragraphs and short non-paragraph text fragments (**3.3**). The problem with this approach is that it blurs your hierarchy of element types, thereby making your documents harder to maintain and more prone to errors.

It might be argued, on the other hand, that the emphasis spanning one or more paragraphs is semantically different from the emphasis that spans one or more words. Therefore, they could use different element types:

```
<emphasis>
<p>This is the first paragraph using emphasis throughout.</p>
<p>And this is the second emphasized paragraph. Note that we can use
<em>nested emphasis</em>, but this time it is a different element
type for the inline level.</p>
</emphasis>
```

If the paragraph-level emphasis is semantically connected with the paragraph element, you can instead add an attribute to those paragraphs that fall within its scope:

```
<p type="emphasis">This is the first paragraph using emphasis
throughout.</p>
<p type="emphasis">And this is the second emphasized paragraph.
Again, <em>nested emphasis</em> is possible.</p>
```

Among these options, there is perhaps no single winner suitable for all situations. Your choice will depend on the semantics of the element in question, the frequency of its use at inline and block levels, and the possible connections between its semantics and that of the standard block-level element (paragraph).

3.4.4 Nested markup

Another issue with text markup is whether nesting elements of one type is to be allowed. Presentation-oriented markup never uses, for instance, `i` within `i` — but for semantic markup, a similar structure may be meaningful. Thus, emphasis within emphasis or a quote within a quote are both perfectly valid *semantically*, even though in an HTML rendition, nesting of the corresponding formatting elements may have no visible effect.

Therefore, to properly transform nested semantic markup, you must use different formatting depending on the nesting level of the semantic element. For example, if you use italic face for emphasis, nested emphasis can be rendered either as regular face ("toggle" approach, where you switch between regular and italic faces for each new nesting level) or as bold italic face ("additive" approach, in which the italic rendition of the parent is augmented by the bold formatting of the child).

3.5 Links

A hyperlink is a very rich concept, even though its implementation in HTML is rather primitive. Basically, an HTML link consists of two parts: the address that tells the browser where to go and the link element itself that (with its attributes and children) defines the link's presentation and behavior. However, in HTML, all possible address types are limited to a single syntax (URI), and all possible link types

are served by one element type (a) with a limited set of attributes. Let's see how we can improve this scheme.

Note that this section only covers *inline links* that are part of the body of a page and thus need to be specified in the page's XML source. *Navigational links*, created by the stylesheet based on the master document data, are discussed in **3.9.1.1**.

3.5.1 Elements or attributes?

When deciding how to cast your linking semantics into XML constructs, it is natural to reuse the HTML approach with a link consisting of an element (signaling the link) and its attributes (providing the address and other link properties). For example, you might write

```
This was <link address="address">reviewed</link> elsewhere.
```

However, this only looks good when you're linking text fragments within a text flow. As soon as you have a separate element representing some object that may have a link property (among others), it is much more convenient to designate the link by an extra attribute of that element rather than a wrapper element. For example, it is easier to create a linked image like this:

```
<image src="button" link="address"/>
```

compared to the HTML-inspired approach:

```
<link address="address"><image src="button"/></link>
```

Not only the address but other properties of a link as well (such as its title, behavior, or classification) might similarly attach as attributes to an element that represents a nontextual link.

So, we see that it is natural to express linking semantics via a set of attributes that may apply to many different elements (or even to any element at all) instead of an element type with its own fixed attributes. This is because a link is most often an attribute of some object rather than an object in itself. This approach was implemented in W3C's

XLink standard,[9] and you may consider incorporating XLink into your source definition for link markup (however, please read the rest of **3.5** for other possible link properties, not all of which are supported by XLink).

For in-flow textual links, you still need a generic linking element type (such as `link` in the example above) that only serves as a markup container for the same set of linking attributes. Most schema languages have no problem defining a separate set of attributes that can be used in different element types.

3.5.2 Link types

Along with the `href` attribute with the link's URI, an `a` element in HTML may provide a `target` attribute for specifying the target window or frame for the linked resource. However, just as you can add attributes with JavaScript code to program various aspects of the link's behavior (e.g., actions performed when the link is activated), your source XML may also need to provide link properties other than the address.

This does not mean, of course, that you'll have to embed JavaScript into your XML source. As with any other data, what you need to do first is develop a *classification* of all possible types of link attributes or behaviors, without detailing their implementation. As soon as you have such a classification, it's easy to coin an appropriate attribute and define the vocabulary of allowed values for it.

Categorizing links. For example, analysis may reveal that your links fall into one of the following categories:

- *internal links* (links to other pages within the site);
- *external links* (links to other sites);
- *dictionary links* (links to a script on an online dictionary site providing definitions for linked words); and

9. `www.w3.org/TR/xlink/`

- *thumbnail links* (thumbnails linked to pop-up windows with larger versions of images or pages).

Both thumbnail links and dictionary links may be either internal or external. However, they need to be classified separately because of their special role on the pages, resulting in different formatting and behavior. On the other hand, you may not be planning any formatting or behavior differences between internal and external links, but separating them into different types is still a good idea because it is a natural classification and because this lets you make your address abbreviations (discussed in **3.5.3**) more logical.

Classifier attributes. To differentiate these link types, we could add a classifier attribute, e.g. `linktype`, specifying the type of the link:

```
...available on the <link linktype="external"
link="www.kirsanov.com/te/">original site</link> and
<link linktype="internal" link="mirror/te">mirrored here</link>.
```

This approach works both with standalone `link` elements and with any other elements that may need to use these linking attributes (e.g., `image`). Note that we used `linktype` rather than `type` and `link` rather than `address` for the attribute names so that the common prefix, `link`, will help you keep track of these attributes as a group without the risk of confusing them with their parent elements' native attributes. You can also separate all linking attributes into a namespace of their own, but this is not really necessary unless you plan to use them with different document vocabularies.

It's also advisable to make all linking attributes but the address (i.e., `link`) optional and provide sensible default values. For example, you can mandate that the missing `linktype` attribute in a linked element implies that the link is internal.

Classifier element types. For in-flow links, instead of (or, better, in addition to) the bulky classifier attribute, a separate element type for each link type is more convenient. As these element types will be used quite often, each should have a short but clear name:

...available on the <ext link="www.kirsanov.com/te/">original site</ext> and <int link="mirror/te">mirrored here</int>.

Separate element types have the additional advantage of being easier to validate with grammar-based schema languages like DTD or XML Schema.

Advanced link types. Other link types may have their own sets of required and optional attributes and may perform other functions, besides creating a link. For instance, dictionary links from the above classification are likely to be used only within text flow, so we can introduce a special element type for them and declare that whenever the address attribute is missing, the element's content is taken as the (abbreviated, **3.5.3**) address:

...was going to <def>disembogue</def> profusely.

...at which point it <def word="disembogue">disembogued</def> itself...

Here, two occurrences of the obscure word *disembogue* are linked to a dictionary site, so that a pop-up window or floating tooltip with the word's definition could be displayed when the link is activated in some way (e.g., clicked or hovered over). You don't need to specify the dictionary site to use, or the complete URL for accessing the dictionary script, or the JavaScript code to create the pop-up; all this is taken care of by the stylesheet. The only thing you may need in the source is the `word` attribute that optionally provides the base form of the linked word or phrase; if it is absent, the contents of the `def` element are used.

For generality, this special kind of link can also be given by a `link` element with `linktype="dictionary"` and the `link` attribute playing the role of `word`.

Similarly, a thumbnail link could be created by a `thumb` element with a single attribute (e.g., `image`). This attribute would provide the identifier of the corresponding image, with the stylesheet doing all the rest: inserting and formatting the thumbnail, creating a display page with the full-size version of the image, and linking it to the thumbnail. The stylesheet can even automatically create the thumbnail from a full-size image (**5.5.2.6**).

3.5.3 Abbreviating addresses

When creating a link, we usually want to specify a certain piece of *content* that the link will point to. What a URL allows us to specify, however, most often is a *file* that can be moved, renamed, or deleted even if the content we are interested in is still out there somewhere. Moreover, a URL includes a lot of technical information (protocol, file extension) that is not relevant for our purpose of establishing a content-level link.

All this invites the idea of using *abbreviated addresses* that would hide the underlying technical complexity of URLs and provide an abstraction layer protecting our semantic XML from URL changes. For each address, we will create an identifier to be used in the XML source; at transformation time, the stylesheet will resolve this identifier into the actual URL to be put into the corresponding HTML link element.

Example: RFC links. Suppose you often need to link to enumerated documents such as RFCs.[10] Such links could use a special value of the link classifier attribute and/or an element type of their own. However, to make them even more convenient, it is natural to use only the RFC number as an abbreviation for the complete URI:

```
...as per <rfc num="1489"/>.
```

Or, the same could be spelled out in a generic fashion:

```
...as per <link linktype="rfc" link="1489"/>.
```

This latter variant uses generic linking attributes that can be applied to different elements to make links out of the corresponding objects, whereas the `num` attribute is only recognized in an `rfc` element.

The XSLT stylesheet will have to recognize this type of link, possibly apply some special formatting to it, and most importantly resolve (*unabbreviate*) the abbreviated address. In this example, unabbreviation would supply the complete URL of the referenced document for the HTML link:

10. An RFC (*Request for Comments*) is one of the series of standards created by the Internet Engineering Task Force (IETF) and governing most of the underlying technical structure of the Internet.

```
...as per
<a href="ftp://ftp.rfc-editor.org/in-notes/rfc1489.txt">RFC 1489</a>.
```

You could also allow an `rfc` element to enclose character content:

```
...which was <rfc num="1489">defined</rfc> in 1993.
```

which would give the following in HTML:

```
...which was
<a href="ftp://ftp.rfc-editor.org/in-notes/rfc1489.txt">defined</a>
in 1993.
```

Mnemonic addressing. Abbreviated addresses in your source XML must be unique only within your site, as opposed to URLs that are globally unique. This means you can make them easier to remember and more meaningful (to you) than are URLs. The abbreviated addresses are also completely devoid of irrelevant technical details and can be arbitrarily long (i.e., detailed and readable) or arbitrarily short (i.e., quick to type and quick to read).

3.5.3.1 Multiple abbreviation schemes

You can use as many independent abbreviation schemes as necessary. Each more or less complete and logical group of addresses can be served by its own abbreviation algorithm (and the corresponding resolver in the stylesheet). For example, links to an online dictionary or search engine might be abbreviated to just the word you want to look up; links to W3C standards can be represented by their unique identifiers as used by the W3C site (e.g., `xslt20` for XSLT 2.0, which unabbreviates into `http://www.w3.org/TR/xslt20/`). Any address domain whose URLs can be "losslessly compressed" into a shorter or easier-to-remember form is ripe for abbreviation.

With multiple abbreviation schemes, the stylesheet must be able to know which one to use for each link. This is where link types (**3.5.2**) are useful, distinguished by a classifier attribute value (`<link linktype="rfc" ...>`) or the element type (`<rfc ...>`) used for each link. It is natural to define abbreviation schemes on a per-link-type basis, or even to define link types based on the abbreviation schemes they are using.

Along with resolving the address, your stylesheet can perform other processing tasks, such as retrieving the title of the referenced RFC to be displayed in the link's floating tooltip. A Schematron schema for your source definition, in addition to performing link syntax validation, can also check for broken links (**5.1.3.3**). Another important advantage is that you can easily change all your RFC links from one RFC repository to another simply by editing the stylesheet.

3.5.3.2 Unabbreviation algorithms

To expand the abbreviated addresses, your stylesheet may use any sources of information, such as local or remote database queries or even web search. It's easiest, however, to create simple algorithmic abbreviations that map to the corresponding URLs through some calculations or string manipulations.

Thus, for external links, the most obvious and perhaps the only sensible abbreviation is dropping the protocol specification (usually `http://`) from the URLs. Even this simple provision can make address input somewhat easier by allowing you to type `www.kirsanov.com` instead of `http://www.kirsanov.com`.

Note, however, that in this case the stylesheet must be able to recognize the protocol part of an address and only add `http://` if it is missing. Addresses that already contain a protocol specification (be it `http://`, `https://`, or `ftp://`) must not be modified in any way.

3.5.3.3 Multicomponent abbreviations

An address abbreviation may contain more than one component. This is often necessary to link to scripts (as opposed to static pages) that require a number of parameters in the request URI. Some of these parameters (e.g., the partner's ID or formatting options) are static and can therefore be filled in by the stylesheet, but the key information pointers (e.g., the date and the number of the article within that date) must be present in the source of the linking page. Here's an example of a link with a multicomponent abbreviated address:

```
As <foonews
      date="02-12-2003"
      num="6490">reported</foonews> by FooBarNews...
```

which could be expanded into the HTML link:

```
As <a
href="http://foonews.com/news?date=02-12-2003&num=6490">reported</a>
by FooBarNews...
```

3.5.3.4 Internal links

One highly recommended abbreviation scheme that makes sense for almost any site is using page identifiers, defined in the master document, instead of pathnames[11] for internal links. This will make your site's structure much more flexible because you will be able to rename a page or move it around without changing all the other pages that link to it.[12]

Linking a foobar. For example, suppose you have a page on your site describing a product called Foobar Plus. You don't want to spell out the complete pathname each time you link to that page, as it may be quite long (e.g., `/products/personal/foobar_plus`). Much more convenient would be using that page's unique (within your site) and easy-to-remember identifier. Since you don't, in all probability, have another Foobar Plus on your web site, it is natural to use an abbreviated name of the product as the identifier:

```
Check out our new <int link="fb+">Foobar Plus</int>!
```

The correspondence between web pages and their identifiers is to be set in the master document (**3.9.1.2**, page 129). Now it doesn't matter if your Foobar Plus page is moved, say, from `/products/personal/foobar_plus` to `/products/corporate/foobarplus`. All you need to do is change the reference in the master document and retransform all site pages.

11. Strictly speaking, HTML links to URIs, not pathnames, but links within a site almost always use relative or absolute pathnames (without a server part) that are also valid URIs.

12. Unfortunately, this only works for your own site. Visitors coming from another site linking to yours will still get a 404 for a moved page.

Aliases. To make life even easier for site maintainers, you can allow them to use any of a number of *aliases* referring to the same page. For example, the Foobar Plus page might just as well be linked to as `fb+`, `foobar+`, or `foobar-plus`. All you need to do is register all such aliases in the master document (see Example 3.2 on page 143).

Linking translations. In multilingual sites, a special kind of link that must be present on every page is the link(s) to the other language version(s) of the same page. The absolute minimum of information needed to construct such a link is, obviously, the identifier of the language we are linking to. Thus, if we write on the Foobar Plus page

```
<lang link="de">This page in German</lang>
```

then the stylesheet will use the current page's pathname to construct the proper HTML link — for example,

```
<a href="/products/personal/foobar_plus.de.html">This
page in German</a>
```

or

```
<a href="/products/personal/foobar_plus.html?lang=de">This
page in German</a>
```

or any other variant, depending on your web site setup. Once again, the correspondence between languages and link URIs is deduced from the master document's data.

3.6 Images and objects

The majority of static images, Java applets, and Flash animations on web pages are not independent objects. Most often, they are components of higher-level content constructs. An image may be a visual accompanying a section heading, a background of a table or the entire page, or a navigation button that is part of a larger navigation system.

In all these cases, your source XML will not contain any image references at all: It is the stylesheet's responsibility to know what images to use with what content structures, where to take these images, and how to format them. Much less frequently, usually within text

flow, you might need to display an image for its own sake — such as a photo, a technical illustration, or a map. It's only these standalone objects that you'll have to specify explicitly in the semantic XML source of a page.

This section covers both static images and various embedded objects such as Java applets, ActiveX controls, and Flash animations. All of these are similar from the viewpoint of XML source markup; below we talk mostly about images, but you should keep in mind that the same applies to most non-HTML external objects used on web pages.

Element type names. The name of the element type for including standalone images in your documents may be either generic (e.g., `image`) or specific (e.g., `map` or `portrait`). If you're only planning to use a few well-defined types of images in a few well-defined situations, you can use narrow and descriptive names for each type. Otherwise (or if you do not yet have any specific plans for the use of images at all), a generic `image` element would be just fine.

Images as attributes. An image object may be quite complex, with additional components, such as a photo caption or credit, stored in attributes or child elements. However, quite often all you need to specify is a source location or an identifier for an image that is an attribute of some other object rather than a standalone object in its own right. For the image types that can be used this way, you can use an attribute of the same name as the standalone image's element type. For example, if your `section`s may feature a photo next to the section's heading, it is more convenient to write

```
<section image="location">
<head>Section heading</head>
...
</section>
```

than to write

```
<section>
<image src="location"/>
<head>Section heading</head>
...
</section>
```

even though your stylesheet may be programmed to create identical formatting for these two inputs.

3.6.1 Abbreviating location

Just as a link's main attribute is the destination address, an image element must, before all, specify the location of the image resource. And, just as we used abbreviated addresses in links, it is natural to use mnemonic identifiers instead of complete image locations. For example, by writing

```
<image src="nymap"/>
```

instead of

```
<image src="img/maps/nymap.png"/>
```

you make your XML source more readable, easier to edit manually, and less prone to errors.

In the simplest case, an abbreviated image reference can be made from its filename by removing the path and extension (which is supposed to remain constant for all images). In more complex cases, an abbreviation might be composed of several parts expressed as attributes, such as a date or a classifier. Finally, your master document could simply store a list of all image locations associated with arbitrary identifiers and possibly aliases (compare **3.9.1**); in this case, all image references in your source will be completely independent of the corresponding locations or other image properties.

Abbreviating aggressively. Along with stripping directory and extension, filename-based abbreviations can be made even more convenient by programming the stylesheet to perform case folding (converting everything to lower- or uppercase) and to remove all whitespace and punctuation. With these provisions, to reference `img/maps/nymap.png` in the above example, we could use any of `nymap`, `ny map`, `N.Y. Map`, and so on.

The goal of using abbreviations is to have your image references named intuitively and consistently and to provide just enough information

in XML for the stylesheet to be able to reconstruct the complete pathname or URI.

3.6.2 Formatting hints

Standalone images may be particularly difficult to separate into independent aspects of content and formatting. The idea of specifying an image identifier and possibly its role in the XML source and then letting the stylesheet figure out all the formatting parameters is attractive, but the reality may be not so neat. Sometimes, you'll have no choice but to add ugly formatting clues to the XML source to get the correct rendition.

An example is a layout where several images are placed on a page, interspersed with text, and aligned alternately against the left or right margin. It is natural to have the stylesheet do the alternating alignment so that only the image identifiers need to be supplied in the source. However, sometimes you may want to force a particular image to a particular margin in the middle of a page. Adding `align="right"` to your XML source is hardly semantic but may be unavoidable if, for example, a left-aligned image visually conflicts with a nearby left-aligned heading.

Think ahead. It is much easier to prevent a disease than to cure it. Thus, it is preferable to design your page layout in such a way that it can be created strictly automatically based on nothing but the semantic XML source. Avoid situations where only manual interaction can produce acceptable formatting.

For example, if you plan to use alternating alignment of images, you could either use centered headings (which will not conflict with either image alignment) or mandate that any image be at least one paragraph away from the nearest heading (this restriction is easy to enforce automatically using Schematron).

Separate namespaces. However, there are situations where adding manual formatting hints to your XML source cannot be avoided. This may happen not only with images, although they are a frequent source of problems. It is advisable to use a separate namespace for all hints that pertain to the same output format (e.g., HTML):

```
<page xmlns:forhtml="http://www.kirsanov.com/formatting-hints-html">
  <p forhtml:column-break="true">
    ...
    <image src="solid wood table" forhtml:align="right"/>
    ...
  </p>
</page>
```

Here, a hint is added to the `p` element specifying that this paragraph must start a new column in a multicolumn layout (assuming the stylesheet cannot figure this out automatically). Another hint floats an image within that paragraph to the right margin.

Now, if you want to render the same XML source into a different format, such as PDF, the new stylesheet will have no problems ignoring anything from the "for HTML" namespace. It is also very easy to strip all HTML formatting hints to produce a purely semantic version of the source. You can store several sets of formatting hints in the same source documents, each in its own namespace, and have the stylesheet select the set corresponding to the current output format (such as "HTML with columns," "HTML without columns," "printable HTML," "PDF," etc.).

HTML documents often use the `height` and `width` attributes in `img` elements as spatial hints to speed up rendering of the page in a browser. You don't need to supply these values in XML; a stylesheet can find out the dimensions of all referenced images itself (**5.5.1**).

3.6.3 Image metadata

Besides the location (full or abbreviated) and possibly formatting hints, an image element may contain various other information.

Textual descriptions. The XHTML specification requires that each image be provided with a piece of text describing what the image is. Traditionally, the `alt` attribute of an `img` element has been used for short descriptions, but in HTML 4.01 and XHTML the `longdesc` ("long description") attribute was added to complement `alt`. Normally, an image description should contain:

- nothing (empty string) for purely decorative images (such as components of frames, backgrounds, and separators);
- the text visible on the image for images that display text (thus, the `alt` of a graphic button must contain exactly the button's label and nothing else);
- a short description of the image's role or content for meaningful images (e.g., `John's photo`).

It's only in the last of the above cases that the image description may need to be supplied in the XML source, preferably in the content of an image element (**2.3.3**). However, if your abbreviated image identifiers are sufficiently readable most of the time, you can save some typing and just reuse these unresolved identifiers (such as `NY map`) for `alt` values.

Captions. Often, a standalone image must be accompanied by a visible descriptive piece of text (as opposed to `alt` descriptions that are normally not shown by graphic browsers). This may be a caption, a photo credit, a copyright notice, or anything else that semantically belongs to this image.

Since this content may need further inline markup, it is better to store it in children of your image element rather than in attributes (**2.3.3**, page 79). The formatting of a caption or caption-like element is determined by the type of the parent image element, which in turn is evident either from its element type name or from the value of a classifier attribute. For example, a photo could be marked up as follows:

```
<photo src="sight">
  <caption>A rare sight.</caption>
  <credit>Dmitry Kirsanov</credit>
</photo>
```

Upon encountering a `photo` element, the stylesheet would expect to format its `caption` child element as a photo caption and the `credit` child element, if present, as credit (e.g., separately from the caption, in a smaller font size, and with "Photo by" prepended to the credit text).

3.6.4 Imagemaps and interactive objects

A simple linked image can be created by adding linking attributes (**3.5.1**) to the image element. Sometimes, however, you may need to create an imagemap where different regions of the image are linked to different destination addresses.

The quick-and-dirty approach. It is natural to reuse the generic `link` element type for specifying multiple links inside an imagemap, by placing `link` elements in the `image` and adding coordinate attributes to define the linked area:

```
<image src="chart 3">
  <link link="address1" shape="rect" x1="0" y1="0" x2="100" y2="20"/>
  <link link="address2" shape="circle" x="50" y="50" radius="5"/>
</image>
```

In HTML, all coordinates for an imagemap area are cramped into one comma-separated attribute value string. You don't need to reproduce that in your XML — instead, you can specify one value per attribute and use descriptive attribute names. It's a good idea to use your schema to check that the set of coordinate attributes in each `link` element corresponds to the value of `shape`.

The thoroughly semantic approach. The syntax shown above may work for an occasional imagemap, but it is still not semantic enough and needs to be improved if you routinely use imagemaps (or other interactive objects). Namely, do the pixel values in the `link` attributes really belong in the source? Probably not, as they are closely bound to the image's "presentation" and tell us nothing about its "content." A better approach is to use each `link` element to associate the *identifier* of an image area with a link address — for example,

```
<image src="chart 3">
  <link link="address1" area="block1"/>
  <link link="address2" area="central-blob"/>
</image>
```

The correspondence between the area identifiers (`block1` and `central-blob` in this example) and the actual pixel coordinates may be stored

in the site's master document. If, however, you want an imagemap to be truly orthogonal to everything else on the site and easily portable to other sites, consider creating a separate XML document for each imagemap storing its active areas and their identifiers.

Accessibility. Interactive objects such as Java applets and Flash movies may also incorporate multiple links (one example is an animated Flash menu). Even though you don't have to specify these links in the HTML code embedding the object, it still makes sense to list them in the XML source of a page so that the stylesheet can construct an alternative access mechanism for those users who cannot (or don't want to) peruse this interactive object.

3.7 Tables

Tables are perhaps the most abused feature of HTML, with the vast majority of tables on web pages being used for layout purposes, not for presenting inherently tabular data. If (like most web designers) you are going to use HTML tables for web page layout, you cannot reflect that in the semantic XML source of a page in any way. It's only the stylesheet that needs to be concerned with layout table construction.

Sometimes, however, you may have some genuinely tabular data that you want to format into some sort of a table on a web page. Still, this does not mean that you have to think in terms of rows and columns when creating a semantic source for such a table.

If you have something you can name, do it. For example, consider a sales data table listing sales figures for several products across several years. The XML way of marking up this data would be to forget that you're working on a table and simply list all available data in an appropriately constructed element tree:

```
<sales-table>
  <product>
    <name>Foobar</name>
    <sold><year>1999</year><number>123</number></sold>
    <sold><year>2000</year><number>140</number></sold>
    <sold><year>2001</year><number>142</number></sold>
  </product>
  <product>
    <name>Barfoo</name>
    <sold><year>1998</year><number>89</number></sold>
    <sold><year>1999</year><number>14</number></sold>
  </product>
</sales-table>
```

This approach frees you from worrying about column alignment, sort order, or empty cells — just dump all your data and you're done. All the rest will be performed automatically by the stylesheet: It can filter out a subset of the provided data, group values in rows and columns, sort them, and fill in "N/A" for missing values. Thus, the above example might come out as follows:

	1998	1999	2000	2001
Barfoo	89	14	N/A	N/A
Foobar	N/A	123	140	142

Tables from triplets. In some cases, such a data-centric approach may also make your source significantly more compact than the table rendition. Thus, a sparse table with mostly empty cells can be represented in the source by *triplets* consisting of a row name, a column name, and the corresponding value at their intersection. Since such a source does not contain separate lists of all columns and rows, the stylesheet will compile them from the triplets.

Is it worth it? Granted, for an occasional table or two, this may be too much work: You'll have to program your stylesheet to recognize various element types and perform various operations (such as normalizing dates) that may be necessary for your tabular data. For simple isolated tables, you may be better off more or less directly reproducing in XML the structure of the target HTML table. However, if you have a lot of simple tables (or a few complex ones) with similar data, or if your tables are updated often, the benefits of the semantic data-centric

approach may easily outweigh the simplicity of the straightforward HTML imitation.

Also, the tabular data on your web site is likely to be coming from some external source, such as a database or a spreadsheet. When you write the code to update your tables automatically, it is usually much easier to first transform the external data into a semantic XML tree and then let the stylesheet do table layout.

3.8 Forms

Interactive elements in HTML are grouped into *forms*. Simple forms such as site search or email newsletter subscription are often used on many pages of the site, and your XML does not need to detail the structure of these forms. Instead, in your source you can treat such a form as an indivisible entity — for example, as a special type of orthogonal block (**3.1.2**) that can be inserted wherever a normal block is allowed.

Sometimes, even this is not required. For example, if *all* pages on your site contain a search field in the page footer, you don't need to mention it in the XML source at all. Your stylesheet will simply add this form to every page it produces, just as it adds all other page components that remain the same from page to page.

What if you need to build something more complex, such as a shipping address input form or a survey form? In these cases you'll need to create an appropriate element type for each variety of the form's input controls (such as text fields, radio buttons, and drop-down lists) as well as for any higher level semantic constructs within the form. This work can be made much easier by reusing some of the existing form vocabularies.

Existing vocabularies. An obvious choice for the existing vocabulary from which you could borrow form-related markup is XHTML, especially if it is your target vocabulary. The Forms module,[13] available

13. www.w3.org/TR/xhtml-modularization/abstract_modules.html#s_forms

starting from XHTML 1.1, may be a good first approximation. It covers many widget types and allows for proper logical structuring of your form.

However, in many cases the XHTML form markup may be too presentation-oriented to be useful for your semantic XML — or simply too awkward. This is mostly due to the historic baggage of older HTML versions. Modern HTML and XHTML had to pile their logical markup provisions on top of the old — limited and inflexible — form components.

For instance, in XHTML you have to write

```
<label for="firstname">First name:</label>
<input type="text" id="firstname"/>
```

instead of the more natural

```
<textfield id="firstname">
  <label>First name:</label>
</textfield>
```

HTML 4.0 *had* to define a separate `label` element that is linked to its `input` by a `for` attribute simply because it had to stay compatible with older HTML versions that did not allow any children in an `input`.

In your source definition, you are free from these concerns and can therefore mark up your forms in a more logical and readable way. It is also important that your own markup may be better integrated with other parts of the system; for example, you could use an abbreviation (**3.5.3**) for the form submission address.

Another existing vocabulary worth looking at is XForms,[14] recently developed by the W3C (see **6.1.3.1** for an XForms example). This is a modern XML-based processing framework that defines not only forms markup but data submission and processing as well. Compared to XHTML forms, XForms markup is more logical and presentation-independent (for example, one form can be rendered both visually in

14. www.w3.org/MarkUp/Forms/

a graphical browser and aurally by a speech browser). Once again, your choice between borrowing XForms markup or developing your own should depend on the complexity of your forms and their relative importance in the project.

Formatting hints. Form presentation is a difficult task. Even with full manual control, it's not always easy to lay out a form so that it looks perfect and remains usable for any data that may be filled into it. Even more difficult is to automate form layout, enabling the stylesheet to consistently build good-looking form pages from the semantic description of the forms' structure. To add insult to injury, different browsers on different platforms often render form controls in wildly different ways.

The key is keeping the layout simple and flexible. Don't strive for precise placement or alignment of controls, as this is impossible to achieve given the vastly different font and screen sizes in browsers. (Also, do not tie the position of other parts of the page to the size or placement of a form — this often results in a broken page layout.) Take advantage of the form structure described in the source by separating groups of form controls into independent layout blocks.

All that said, adding formatting hints (**3.6.2**) to control form layout may turn out inevitable. The most common case is specifying the size of text input fields.[15] If you think you need something more elaborate than that, it is usually an indication that you should try instead to simplify your form's presentation (or your page design in general).

3.9 Master document

In the previous chapter (**2.1.2.1**, page 49), we found that any web site consisting of more than one page must have a master document providing shared content and a site directory. In this section, we'll look at some practical examples of constructs in a typical web site's master document.

15. It might be argued that the size of an input field is one of its essential semantic aspects and not a superficial formatting property.

You may find the sample master document described here (see Example 3.2, page 143, for a complete listing) somewhat eclectic. This eclecticism, however, stems from the real-world practice of XML web sites. In fact, the master document is more of a database than a document (**1.2**). The layout of components in this database is rarely important, as they are not processed sequentially but accessed in arbitrary order. For lots of ideas on how to access and use the master document content from the stylesheet, see Chapter 5.

A master document represents a new document type, with its root element type different from that of a page document, and most other element types usable only in a master document. However, if you don't use DTDs (**2.2.4**) or XSDL, this distinction has little practical value, and you can use one schema to validate all of your XML (both page documents and the master document). Such a schema written in Schematron is shown in Example 3.3, page 149 (see also **5.1.3** for advanced Schematron checks).

3.9.1 Site structure

The role of the master document is that of a hub that all other documents refer to when they need to figure out a wider context of the web site or establish mutual links. Whenever the stylesheet needs some information that is not supplied by the currently processed document, it will consult the master document to find either that information or a link to it.

Therefore, the most important part of a master document is the *site directory* — a collection of information about all pages of the site and their organization. This directory is used for building the site's navigation as well as for resolving abbreviated internal links (**3.5.3**).

Besides pages, other components of the site may also be mentioned in the master document, such as all Flash animations you have or all images of a specific kind used on the site. Units of orthogonal content must be listed in the master document as well (**3.9.1.3**) so that pages can reference and incorporate them. Finally, sources of dynamic content must be registered for the stylesheet to know what to insert into static page templates (**3.9.1.4**).

3.9.1.1 Menu structure

A flat list of all pages is not sufficient for building a usable site. We also need to represent the structure of the site's menu and the correspondence between menu items and pages.

A simple site's menu may be little more than a linear list of links to each of its pages. However, most sites require more complex menu structures. Common are hierarchical menus where some of the top-level items encompass multiple subpages and/or nested submenus. Such a structure is straightforward to express in XML.

Some sites may have more than one menu. For example, there may be a menu of *topics* (content sections) and another independent menu of *tools* (pages that help navigate the site, such as search and site map). Such orthogonal menu hierarchies can be stored in independent XML subtrees within the master document.

3.9.1.2 Menu items and pages

What do we need to store in the master document for each menu item? To build a clickable menu element, we must know at least its label (the visible text displayed in the menu) and the page that it is linked to. A label may contain inline markup and should therefore be stored in a child element. As for the link, it is natural to use the general linking attributes with abbreviated addresses that we've developed for in-flow links on site pages (**3.5.1**).

Items vs. pages. A menu item is not the same as a page of the site. Some pages may not be available through the menu, while others may be linked from more than one menu item. Therefore, the page itself must be represented by a separate element that the menu item element will link to.

However, that does not mean that these page elements must be stored in a different part of the master document. You can still categorize all your pages under the branches of the menu tree: Even if a page is not *linked* from the menu, usually you can find a branch where it logically *belongs* (unless it is orthogonal content, **3.9.1.3**). The stylesheet will

thus be able to read the menu structure both hierarchically (when looking for menu items) and sequentially (when looking for pages).

Here's a possible representation of a menu item:

```
<item link="products">
  <label>Products</label>
  <page id="products" title="Our products"
        src="products/"/>
  <page id="software" title="Our software"
        src="products/software/"/>
  <page id="hardware" title="Our hardware"
        src="products/hardware"/>
</item>
```

In addition to a `label` and one or more `page`s, an `item` may also contain other `item` children. A complete menu description would thus consist of a hierarchy of `item`s under one parent, e.g. `menu`. Note that in each `page` element, the `id` attribute provides a unique identifier of not only that element, but of the page itself. It is these identifiers that are used as abbreviated addresses (**3.5.3**) in internal links.

How unabbreviation works. When resolving a link, the stylesheet translates the page identifier into the location of that page taken from the `src` attribute. However, that attribute's value is also somewhat "abbreviated" in that it omits irrelevant technical information such as the filename extension and the default filename (usually `index.html`) in a directory. These omitted parts are easy to restore by applying simple rules, so the three `page` elements in the above example would yield these page locations:

```
/products/index.html
/products/software/index.html
/products/hardware.html
```

Note that a location ending with a "`/`" is considered a directory and has "`index.html`" appended; other locations only receive the "`.html`" extension.

Accessing the source. There is one more reason to store page pathnames without extensions. When locations are resolved for the

purpose of accessing the source XML documents rather than creating an HTML link, the same `src` values are transformed into `*.xml` file locations (assuming the directory structure of the site source is similar to that of the transformed site, **3.9.3**). For stylesheet code examples to access this menu structure, see Chapter 5 (**5.1.1**, **5.7**).

Storing page metadata. Sometimes, a more complex layout for the `page` elements may be necessary. For example, if your bilingual site provides two language versions of each page, a `page` element could hold both metadata that is common to all language versions of the page (e.g., the page's identifier and source location) and language-specific metadata (e.g., title):

```
<page id="software" src="products/software/">
  <translation lang="en">Our software</translation>
  <translation lang="fr">Nos logiciels</translation>
</page>
```

Some of the metadata (**3.1.1**) may also be moved from page documents into the master document for convenient access. For example, if you want to control which pages of the site are to be seen by search engine spiders and which are hidden from them, you could add a corresponding value to each page's source document. However, since this information will be pulled from all pages of the site simultaneously, it is more convenient to add a spider control attribute to the `page` element in the master document. This way, the stylesheet will be able to produce a site-wide `robots.txt` file for external spiders and/or a configuration update for a local search engine spider without accessing all page documents.

3.9.1.3 Orthogonal content

Along with all pages, a master document should also list all the units of orthogonal content that your site will use (**2.1.2.2**, page 51). However, unlike pages, orthogonal content references cannot be categorized under the menu hierarchy (that is why this content is *orthogonal*, after all). You'll need to create a separate construct to associate orthogonal content identifiers with corresponding (abbreviated) source locations — for example,

```
<blocks>
  <block id="news" src="news/latest"/>
  <block id="subscribe" src="scripts/subscribe"/>
  <block id="donate" src="scripts/donate"/>
</blocks>
```

Now if the stylesheet processing a page document encounters a `block` that has no content of its own but references some orthogonal content unit — for example, by specifying `idref="news"` — the document at `news/latest.xml` will be retrieved and inserted into the current document, formatted as appropriate for an orthogonal content block.

It is important that the `id` and `src` attributes of a master document's `block` element have the same names and semantics as the attributes of `page` elements (**3.9.1.2**). We will use this when writing stylesheet code to unabbreviate links or search through all pages of the site (Chapter 5), since every page must be registered as either a `page` in the menu or a source of an orthogonal `block` (or both).

Extracting orthogonal content. In the last example, each orthogonal block was stored in its own file — but this is not always the best approach. You may want to reuse parts of regular pages as orthogonal content.

For instance, the news page of a site is often a list of news items in reverse chronological order. You may want to automatically extract the most recent news item and display it in an orthogonal content block on other pages of the site. Another example is a "featured product" blurb extracted from that product's own page and reused on the front page of the site.

For these situations, what we need is a way to specify what part of the original page document is to be reused as orthogonal content on other pages. Since this part will most likely also be a `block`, we only need to indicate the `id` of the block we are interested in. Thus, if the most recent news `block` on the news page always has `id="last"`, we could write in the master document:

```
<block id="last-news" src="news/" select="last"/>
```

Now any page can place a copy of the latest news item by referencing the corresponding orthogonal block by its identifier, `last-news`. For example, your page document might contain

```
<block idref="last-news"/>
```

Likewise, the featured product blurb could be extracted from the `block` with `id="blurb"` on that product's page:

```
<block id="feature" src="products/foobar" select="blurb"/>
```

Here, the featured product is identified by the path to the corresponding document (`products/foobar.xml`). When you want to feature a different product, all you need to do is change this value so it points to another product's page (assuming each product page has exactly one `block` with `id="blurb"`; see also **5.1.3.7**). After that, all pages that use

```
<block idref="feature"/>
```

will (after you rerun the transformation) display the blurb for the new product.

Logically, without the `select` attribute, a master document's `block` will reference the entire content of the document pointed to by the `src` attribute. Your Schematron schema could also check that the referenced elements actually exist in the referenced documents (see **5.3.3.1**, page 224 for how to code this).

No perfection in this world. It would be even more natural to use XPath expressions for extracting orthogonal blocks. Then we could use not only the `id` attribute value but any XPath test for identifying the block we need. For instance, for the first `block` on the page, we would write

```
<block id="news" src="news/" xpath="//block[1]"/>
```

Selecting the last `block` that has a `section` inside would be as simple as

```
<block id="lastsection" src="dir/page"
      xpath="//block[section][last()]"/>
```

There's only one problem with this kind of selector: In XSLT, you can't take a string and treat it as an XPath expression — and what the master document (or any other document) stores in its attributes is always just strings from the XSLT processor viewpoint.

Saxon offers the `saxon:evaluate()` extension function (4.4.2.1) that might save the idea, but its implementation is quite limited, not to mention nonportable to other XSLT processors. Much better is the `dyn:evaluate()` function[16] from EXSLT (4.4.1) which is currently supported by several processors but not by Saxon.

3.9.1.4 Registering dynamic content

Recall our discussion of dynamic sites in 1.5. We found that a dynamic web page is produced from two main parts — static templates and dynamic values — and that both can (and should) use XML markup. It's now time to see how these concepts fit into the source definition we are building.

One way of many. There exist different ways to aggregate dynamic content and static templates. Some of them come before XSLT transformation, which is usually the last stage in a dynamic XML web site workflow; in these cases, you don't need any special source markup because your stylesheet will get complete seamless page source with both static and dynamic content. However, in some situations (notably offline XSLT processing, 1.4.1) implementing dynamic content aggregation in XSLT is convenient. This section shows one approach to organizing such transformation-time incorporation of dynamic content.

Reusing blocks. An orthogonal content block that the stylesheet extracts from another document may be considered a special case of a composite dynamic value. Therefore, it makes sense to extend our blocks' markup constructs so that they cover the "truly dynamic" content as well — content that is calculated or compiled by some external process and not just stored in a static document.

We can define a number of block conventions that will allow us to use blocks not only for enveloping independent bits of content but also as links to external sources of information. Once again, our guiding principle is: Let the page author use short mnemonic identifiers and hide all the gory details of accessing data in the master document and/or stylesheet.

16. `www.exslt.org/dyn/functions/evaluate`

Calling a process. Suppose we want to build a site map page that automatically compiles a hierarchical list of all pages of the site. The first thing we need is the static part of that page — a document that stores all the static bits unique to the page, such as an introductory paragraph and heading(s). This is a normal page that is listed in the `menu` hierarchy in the master, just like any other page.

Wherever we want to insert our dynamic content into that static frame, we place a block reference, e.g.:

```
<block idref="sitemap"/>
```

In the master, however, we cannot associate the `sitemap` identifier with any source file, since no such file exists — the list of pages is generated dynamically.

Instead, we must associate our dynamic block identifier (`sitemap`) with an identifier of some abstract *process* that generates its data. You can think of a process as a kind of a script or application; it may accept some parameters that affect its output. Thus, if we write in the master document (within the same `blocks` envelope used for orthogonal blocks)

```
<block-process id="sitemap" process="sitemap" mode="text" depth="2"/>
```

then the stylesheet will know that a `sitemap` block needs to be filled in with data generated by the `sitemap` process with parameters `mode="text"` and `depth="2"`. This process can be, for example, a callable template within the stylesheet (**4.5.1**) or an external program. With this approach, document authors don't need to know anything about processes or parameters; they use identifiers to refer to data sources, and the master document associates each source with a process and its set of parameters.

Watching a directory. A stylesheet can access external files even if the list of these files is changing dynamically. For example, an external process (which may or may not be another stylesheet) might be dropping its output XML documents into a directory. Your stylesheet would then read the list of files in that directory (**5.3.2**) and do what it pleases with their content — such as dump all available content

from all files into one page or perform some elaborate selection, filtering, or rotation.

If, for example, your stylesheet implements a `list-titles` process that takes a directory as a parameter and returns the list of `title` elements from all XML documents in the directory, then you could define a block to perform this operation on all (dynamically updated) documents in the `news` directory by writing in the master document

```
<block-process id="news-list" process="list-titles" dir="news/"/>
```

In a page document that wants to use this list, you would then write simply

```
<block idref="news-list"/>
```

XML, not HTML. Note that processes similar to `sitemap` or `list-titles` should only aggregate content, not format it. This means that the corresponding templates or functions in your stylesheet must produce valid XML data (nodesets), not HTML renditions. You would then feed these nodesets to the regular formatting templates in the same stylesheet (see **5.3.3.1** for ideas on how to chain templates together). If a process is implemented as an external program, it should return serialized XML data or plain text that the stylesheet will be able to convert to nodesets.

3.9.2 Common content and site metadata

On a typical web site, all pages contain bits of information that either remain the same or change predictably from page to page. Some of this repeating data, such as the company logo or tag line, actually belongs to the domain of presentation rather than content and therefore needs to be filled in by the stylesheet rather than stored in the source. Other components, such as webmaster email links, "designed by" signatures, copyright or legal notices, etc., are natural to store in the master document.

It is recommended that you envelop all such bits of content in one or more umbrella elements, each containing data with similar roles or positions on the pages. Here's a master document fragment defining the footer to be placed at the bottom of each page:

```
<page-footer>
  <designed-by>Site design: <ext link="www.kirsanov.com">Dmitry
  Kirsanov Studio</ext></designed-by>
  <legal linktype="internal" link="legal">Legal notices</legal>
  <contact linktype="internal" link="contact">Contact us</contact>
</page-footer>
```

Note that the elements inside `page-footer` may have mixed content with any of the text markup, linking, or other elements that were developed for page documents. In particular, we see internal and external links used in this example, each with its own address abbreviation scheme (**3.5.3**).

The `page-footer` parent element makes the stylesheet simpler and more bulletproof: Instead of providing templates for each of the individual footer elements, you can program the stylesheet to process all items within a `page-footer` in turn, and only provide separate templates for those that differ from others in formatting. With this approach, you'll be able to add a new element type for a new footer object even without changing the stylesheet.

Similarly, we can create an envelope for storing metadata that applies to the entire site. Examples of such metadata include site-wide keyword lists (which could be merged with page-specific keywords supplied by the page documents, **3.1.1**) and extended credits (which could be put in comments in the HTML code of the site's front page).

3.9.3 Processing parameters

Your stylesheet will need to know some parameters of the environment in which it is run as well as the environment where its HTML output will be placed. The most frequently required processing parameter is the base URI that the stylesheet will prepend to all the image and link pathnames. By changing this parameter, you can turn all internal link URIs from relative to absolute with an arbitrary base, which is useful for testing the site in different environments. Other parameters may provide the path to the source tree and the operating system under which the stylesheet is run (which, in turn, may affect the syntax of pathnames).

Grouping parameters into environments. It is important that the same set of source files may be processed on different computers — for example, on a developer's personal system, then in a temporary (staging) location on the server, and finally in the publicly accessible area on the target server. Each of these environments will require its own set of processing parameters. It is therefore convenient to define several groups of parameter values, one for each environment, and select only one of the groups by its identifier when running the transformation.

Where to store the environment groups? Obviously, the need to group parameters and assign a unique identifier to each group makes using XML very convenient — as opposed to, say, storing the values within scripts used to run the site build process (**6.5.1**). Note also that scripts are the most OS-dependent part of the site setup, so it is best to keep them as simple and therefore as portable as possible. And of all the XML documents of a web site, the two most likely choices are the XSLT stylesheet and the master document.

Your stylesheet is more likely to be shared (in whole or in part) among different projects, so it is not wise to use it for storing information that is too project-specific. Also, even though you can use XSLT variables for storing processing parameters, it is more convenient to use custom element hierarchies for structuring and accessing this data. For these reasons, the master document emerges as the most natural storage for processing parameters.

This does not mean that your master document will differ among environments. Instead, all identical copies of it will have information on all environments, and each environment will extract the relevant set of data by passing a parameter to the stylesheet.

Here's an example group of parameters that define the processing environment called `staging` (see **3.10.2** for the meanings of the elements):

```
<environment id="staging">
  <os>Linux</os>
  <src-path>/var/website/src/</src-path>
  <out-path>/var/website/out/</out-path>
  <target-path>/test/</target-path>
  <img-path>img</img-path>
</environment>
```

3.9.4 Site-wide content and formatting

Normally, formatting of web pages is created by the stylesheet. Sometimes, however, formatting is dependent on certain parameters that, being more content than style, belong in the site's source and not in the stylesheet. Also, sometimes the stylesheet may need to create objects that are used on many pages but do not belong to any one page in particular. In both these situations, the master document is a convenient place to store data.

Site-wide buttons. An example of such an object is a pair of graphic buttons — "next" and "prev" — used on sequential pages (such as chapters in an online book). If your stylesheet generates other graphic buttons on the site (**5.5.2**), design consistency and maintainability will be much better if *all* buttons are done in the same way.

These buttons are not specific to any particular page; moreover, pages that use them don't even need to mention the buttons in the source because the stylesheet can automatically create the page sequence, including appropriate navigation. All we need is to store the button labels somewhere so the stylesheet can generate the buttons. It makes sense to use the master document for this.

You can store the button labels in a separate element in the master and program the stylesheet to regenerate the buttons when run with the corresponding parameter. For example,

```
<buttons>
  <button>prev</button>
  <button>next</button>
</buttons>
```

3.10 Summary examples

This section presents examples of complete documents that bring together everything we've discussed in the last two chapters (and more). The content is fictitious, but the structure and markup are from real web site projects (somewhat abridged for readability).

3.10.1 Page document

Compared to the master document example (**3.10.2**), the page document in Example 3.1 is short and simple. This is, in fact, what you should strive for in your project. Page documents are the primary work area for those who will update and maintain the site, so the layout of a page document must be as simple and self-evident as possible. (For instance: Do we need indentation in page documents? Probably not, unless it is taken care of automatically.)

The main rule of thumb is: If you can move a bit of information away from a page document to the master or to the stylesheet, do that.

3.10.2 Master document

Example 3.2 shows a master document that compiles most of the data we discussed in **3.9** but adds a few new twists.

Languages. Our example site is bilingual (English and German), so all titles and labels are provided in two languages, and the languages themselves are listed in a `languages` element. We add an internal DTD subset with mnemonic entity references (**2.2.4.3**) for German characters.

Environments. For every installation where the site can be built, an `environment` element with a unique `id` supplies the following information:

- `src-path` is the base directory of the XML source documents tree.

Example 3.1 en/team/index.xml: A page document.

```
<?xml version="1.0" encoding="us-ascii"?>
<page keywords="team, people, staff, competences, skills">

<title>Our team</title>

<!-- Main content block: -->
<block type="body">
<p>With backgrounds in technology and communications, FooBar's
experienced management team has - you guessed it -
<em>the right combination of skills for success</em>.</p>

<section image="mike">
<head>Mike M. Anager</head>
<subhead>CEO</subhead>
<p>CEO and Co-Founder, Mike leads FooBar towards bringing the vision
of "personal foobar" to reality. He previously served as Chief
Architect at <ext link="www.barfoo.com">BarFoo Corporation</ext>.</p>
</section>

<section image="ed">
<head>Ed N. Gineer</head>
<subhead>VP, Engineering</subhead>
<p>Ed has over 30 years of foobar design experience under his
belt. He has personally contributed to the most acclaimed of
our <int link="solutions">products</int>, including the famous
<int link="fbplus">Foobar Plus</int>.</p>
</section>

<section image="jack">
<head>Jack J. Anitor</head>
<subhead>Senior Janitor</subhead>
<p>Jack's expert janitorial skills and experience have been
critical in the success of FooBar.</p>
</section>
</block>

<!-- Orthogonal content blocks: -->
<block idref="subscribe"/>
<block idref="feature"/>

</page>
```

- `out-path` is the directory where the output files will be placed (used in batch mode, **5.6**). It is also assumed that the images subdirectory (`img-path`) is under `out-path`.
- `img-path` is where all the images (both static and generated) are stored. This path is relative to `out-path`.
- `target-path` is the common part of all URIs used in the resulting HTML files to refer to images or other pages of the site. Thus, if you transform and view your pages locally at `out-path`, then `target-path` may be the same as `out-path`. If, however, you are going to upload the transformed site to a directory on a web server and access it at, say, `http://www.example.org/test/`, then `target-path` may be either `/test/` (for absolute pathnames starting with `/`) or an empty string (for relative pathnames).

On Windows, all absolute paths must be given in the `file:/` URL format.[17] This is the only standard and reliable way to represent an absolute pathname that includes a drive letter. In HTML, URLs with `file:/` work for both links and image references in all browsers we tested. Other platforms may use absolute pathnames without the `file:/`.

Menu. The `menu` lists all the `page`s of the site. For each `page`, the `src` attribute contains the page's pathname (add `.xml` for source files or `.html` for output files) relative to the site's root directory.

Each page has an `id` attribute used to link to it. To make life easier, you can also provide a space-separated list of aliases in the `alias` attribute. In internal links to this page, you can use either its `id` or any of the aliases.

Each menu `item` has a `label` child storing the item's visible label. In the menu on a web page, each item is supposed to be linked to its first `page` child, so there's no need to specify a `link` in an `item`.

17. The single slash character in this URL means that the file is available locally and not on a network host.

It is assumed that the English and German versions of the source files are named the same but stored in different directory trees under the root directory. The corresponding directories are named after the language designations defined in `languages`. So, for instance, the complete path to the German `fbplus` source page in the `staging` environment would be constructed as follows:

```
/home/d/web/de/solutions/foobar_plus.xml
```

Blocks. The `blocks` element holds a list of orthogonal content blocks with their identifiers (`id`), source document locations (`src`), and block selectors (`select`, **3.9.1.3**). Note that the `subscribe` page is listed only once as an orthogonal source, while the `solutions/foobar_plus` page is both in the menu and in the `blocks` list. For this reason, a `block` must specify a complete location for the orthogonal content source and not just its `id`, as all other links do, because not all orthogonal documents are registered in the menu and assigned an `id`.

Misc. Finally, the master document lists the common part to be prepended to page titles (**3.2.4**) on all pages (`html-title`), page footer content (`page-footer`), and two labels for buttons that need to be created by the stylesheet (`buttons`).

Note that the `mailto` links used in `page-footer` represent a special link type (**3.5.2**, page 109) with an abbreviated address (the corresponding resolved URI will have `mailto:` prepended to the email address).

Example 3.2 `_master.xml`: The master document.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE site [
  <!ENTITY auml "&#228;">
  <!ENTITY ouml "&#246;">
  <!ENTITY uuml "&#252;">
]>
```

```
<site>

  <!-- Environments: -->
  <environment id="local">
    <os>Windows</os>
    <src-path>file:/C:/Work/Website/XML/</src-path>
    <out-path>file:/C:/Work/Website/Out/</out-path>
    <target-path>file:/C:/Work/Website/Out/</target-path>
    <img-path>Images</img-path>
  </environment>
  <environment id="staging">
    <os>Linux</os>
    <src-path>/home/d/web/</src-path>
    <out-path>/home/d/web/out/</out-path>
    <target-path>/</target-path>
    <img-path>img</img-path>
  </environment>
  <environment id="final">
    <os>BSD</os>
    <src-path>/var/tomcat/webapps/cocoon/foobar/</src-path>
    <out-path>/var/tomcat/webapps/cocoon/foobar/</out-path>
    <target-path>/cocoon/foobar/</target-path>
    <img-path>img</img-path>
  </environment>

  <!-- Languages: -->
  <languages>
    <lang>en</lang>
    <lang>de</lang>
  </languages>

  <!-- Menu: -->
  <menu>
    <item>
      <label>
        <translation lang="en">Home</translation>
        <translation lang="de">Home</translation>
      </label>
      <page id="home" alias="index front fp frontpage" src="index"/>
    </item>
```

```
  <item>
    <label>
      <translation lang="en">Solutions</translation>
      <translation lang="de">L&ouml;sungen</translation>
    </label>
    <page id="solutions" src="solutions/intro_solutions"/>
    <item>
      <label>
        <translation lang="en">Life</translation>
        <translation lang="de">Das Leben</translation>
      </label>
      <page id="life" src="solutions/life"/>
      <page id="fbplus" alias="foobar_plus fb+ foobar+"
           src="solutions/foobar_plus"/>
      <page id="fbminus" src="solutions/foobar_minus"/>
    </item>
    <page id="universe" src="solutions/universe"/>
    <page id="everything" src="solutions/everything"/>
  </item>
  <item>
    <label>
      <translation lang="en">Our team</translation>
      <translation lang="de">Unser Team</translation>
    </label>
    <page id="team" src="team/index"/>
    <page id="history" src="team/history"/>
    <page id="hire" src="team/hire"/>
  </item>
  <item>
    <label>
      <translation lang="en">Contact</translation>
      <translation lang="de">Kontakt</translation>
    </label>
    <page id="contact" src="contact/contact"/>
    <page id="map" src="contact/map"/>
  </item>
</menu>
```

```
<!-- Orthogonal and dynamic blocks: -->
<blocks>
  <!-- Extract the 'summary' block from the product page: -->
  <block id="feature" src="solutions/foobar_plus"
     select="summary"/>
  <!-- Extract the 'last' block from the front page: -->
  <block id="news" src="index" select="last"/>
  <!-- Take the entire subscribe.xml: -->
  <block id="subscribe" src="subscribe"/>
  <!-- Run site map generation: -->
  <block-process id="sitemap" process="sitemap"
     mode="text" depth="2"/>
  <!-- Run list-titles on all files in news/: -->
  <block-process id="news-list" process="list-titles"
     dir="news/"/>
</blocks>

<!-- The common part of the page titles: -->
<html-title>
  <translation lang="en">Foobar Corporation AG</translation>
  <translation lang="de">Foobar Corporation AG</translation>
</html-title>

<!-- Page footer content: -->
<page-footer>
  <copyright>
    <translation lang="en">&#169; 2003 by Foobar Corporation AG.
      All rights reserved.</translation>
    <translation lang="de">&#169; 2003 by Foobar Corporation AG.
      All rights reserved.</translation>
  </copyright>

  <language-switch>
    <translation lang="en">
      <lang link="de">Diese Seite in deutsch</lang>
    </translation>
    <translation lang="de">
      <lang link="en">This page in English</lang>
    </translation>
  </language-switch>
```

```
    <contact-webmaster>
      <translation lang="en">
        Problems using this site? Contact the
        <mailto link="webmaster@foobar.com">Webmaster</mailto>.
      </translation>
      <translation lang="de">
        Probleme mit dieser Web-Site? Kontaktieren Sie bitte unseren
        <mailto link="webmaster@foobar.com">Webmaster</mailto>.
      </translation>
    </contact-webmaster>
  </page-footer>

  <!-- Sequence navigation buttons: -->
  <buttons>
    <button id="prev">
      <translation lang="en">prev</translation>
      <translation lang="de">zur&uuml;ck</translation>
    </button>
    <button id="next">
      <translation lang="en">next</translation>
      <translation lang="de">vorw&auml;rts</translation>
    </button>
  </buttons>

</site>
```

3.10.3 Schematron schema

The schema in Example 3.3 is used to validate both the master document and page documents of our Foobar site. This makes sense because these document types have a lot in common. Still, for readability the schema is broken into three `pattern`s: One tests the master document, another tests page documents, and the last one tests constructs that occur in both document types (this includes links, images, and text markup).

Languages. The `lang-check` abstract rule checks that the element being checked contains exactly as many `translation` children as there are languages defined in the `languages` element. This rule can then be reused for any element that provides information in two languages. A separate rule with `context="translation"` additionally checks that

the `lang` attributes correspond to the defined languages and that each language version is provided only once.

Element presence. In this schema, many element-presence checks are lumped together for simplicity (e.g., all children of an `environment` are checked in one `assert`). This does not have to be that way; if you want your schema to be really helpful, you can write a separate check with its own diagnostic message for each element type, explaining its role and the possible consequences of its being missing from the source.

Context-sensitive checks. Note that there are two different `page` element types: One is used in the master document, and the other is the root element type in a page document. The same applies to `block`s. The schema, however, has no problems differentiating between these element types based on the context.

Reporting unknowns. One function of a schema is to check for unknown element type names (most often resulting from typos). In Schematron, this can be implemented by providing a dummy `rule` with no tests, listing all defined element types as possible contexts. Following that, a rule with `context="*"` signals error whenever the rule is activated. This technique is possible because each context will only match one rule per pattern; if an element was not matched by the dummy rule, it is caught by the next rule and reported as unrecognized.

It's only a beginning. This example schema demonstrates only the basic, most critical checks. Your own schema may be significantly larger and more detailed than this, although it will likely use mostly the same techniques. Consider this schema a phrasebook with common expressions for typical situations. Several advanced tricks for validating complex constraints are discussed in Chapter 5 (**5.1.3**).

Example 3.3 `schema.sch`: A Schematron schema for validating page documents and the master document.

```
<schema xmlns="http://www.ascc.net/xml/schematron">

<!-- Checks for the master document: -->
<pattern name="master">

<rule context="site">
  <report test="count(//environment) = 1">
    Only one 'environment' found; you will need to create more if you
    want to build the site in a different environment.
  </report>
  <report test="count(//environment) = 0">
    No 'environment' elements found; the stylesheet will be unable to
    figure out pathnames.
  </report>
  <assert test="languages and menu and html-title and page-footer
                          and blocks">
    One of the required elements not found inside 'site'.
  </assert>
</rule>

<rule context="page-footer">
  <assert test="copyright and language-switch
                          and contact-webmaster">
    One of the required elements not found inside 'page-footer'.
  </assert>
</rule>

<rule context="environment">
  <assert test="src-path and out-path
                and target-path and img-path and os">
    One of the required elements not found inside 'environment'.
  </assert>
  <assert test="@id">
    An 'environment' must have an 'id' attribute.
  </assert>
  <assert test="count(//environment/@id[. = current()/@id]) = 1">
    The 'id' attribute value of an 'environment' must be unique.
  </assert>
</rule>
```

```
<rule context="src-path | img-path | out-path | target-path">
  <report test="*">
    The '<name/>' element cannot have children.
  </report>
  <report test="(normalize-space(.) = '')
                and not(name() = 'target-path')">
    The '<name/>' element cannot be empty.
  </report>
</rule>

<rule context="languages">
  <assert test="count(lang) = count (*)">
    The 'languages' element can only have 'lang' children.
  </assert>
  <assert test="count(lang) &gt; 0">
    The 'languages' element must have at least one 'lang' child.
  </assert>
</rule>

<rule context="languages/lang">
  <assert test="count(//languages/lang[. = current()]) = 1">
    Each language must be specified only once.
  </assert>
</rule>

<rule context="menu">
  <assert test="count(item) = count (*)">
    The 'menu' element cannot contain elements other than 'item'.
  </assert>
</rule>

<rule context="item">
  <assert test="label" diagnostics="label-element">
    A 'label' element is missing.
  </assert>
  <report test="count(label) &gt; 1" diagnostics="label-element">
    There is an extra 'label' element.
  </report>
  <assert test="page">
    At least one 'page' element should be specified within an 'item'.
  </assert>
</rule>
```

```
<rule context="menu//page">
  <assert test="@src">
    Each 'page' must have an 'src' attribute.
  </assert>
  <assert test="@id">
    Each 'page' must have a unique 'id' attribute.
  </assert>
  <assert test="count(//page/@id[. = current()/@id]) = 1">
    The 'id' attribute value of a 'page' must be unique.
  </assert>
</rule>

<!-- Abstract rule to check 'transformation' children: -->
<rule abstract="true" id="lang-check">
  <assert test="count(translation) = count(//languages/lang)">
    The number of 'translation' children in '<name/>' must correspond
    to the number of defined languages. If this element does not
    exist in one of the languages, use an empty 'translation' element.
  </assert>
  <assert test="count(translation) = count(*)">
    There must be no child elements here other than 'translation'.
  </assert>
</rule>

<!-- Applying the abstract rule to all bilingual elements: -->
<rule context="label | html-title | copyright
               | language-switch | contact-webmaster | button">
  <extends rule="lang-check"/>
</rule>

<rule context="translation">
  <assert test="@lang">
    Each 'translation' must have a 'lang' attribute.
  </assert>
  <assert test="@lang = //languages/lang/text()">
    The value of the 'lang' attribute must correspond to one of the
    defined languages.
  </assert>
  <report test="@lang = preceding-sibling::translation/@lang">
    There is another 'translation' element under this parent with the
    same value of the 'lang' attribute.
  </report>
</rule>
```

```
<rule context="blocks">
  <report test="*[not(self::block or self::block-process)]">
    A 'blocks' element must only contain one or more 'block' or
    'block-process' elements.
  </report>
</rule>

<rule context="blocks/block">
  <assert test="@id and @src">
    A 'block' defined in the master document must have both 'id' and
    'src' attributes.
  </assert>
  <assert test="count(//blocks/block/@id[. = current()/@id]) = 1">
    The 'id' attribute value of a 'block' must be unique.
  </assert>
</rule>

</pattern>

<!-- Checks for page documents: -->
<pattern name="page">

<rule context="/page">
  <assert test="@keywords">
    Please consider adding a list of keywords to the page. Use a
    'keywords' attribute for that.
  </assert>
  <assert test="title">
    Each 'page' must have a 'title'.
  </assert>
  <assert test="count(title) &lt; 2">
    A 'page' may have only one 'title'.
  </assert>
  <assert test="block">
    Each 'page' must have at least one 'block'.
  </assert>
</rule>

<rule context="page//block">
  <assert test="@idref or *">
    A block must have either an 'idref' attribute (referring to an
    orthogonal block) or children.
  </assert>
```

```
  <report test="@idref and *">
    A block cannot have both an 'idref' attribute and children.
  </report>
  <report test="count(p | section) &lt; count(*)">
    A block can only have 'p' or 'section' children.
  </report>
</rule>

<rule context="section">
  <assert test="head">
    A section must have a 'head'.
  </assert>
  <assert test="p">
    A section must have at least one 'p' (paragraph).
  </assert>
  <assert test="normalize-space(text()) = ''">
    A section cannot contain text. Use a 'p' element to include a
    paragraph of text.
  </assert>
</rule>

</pattern>

<!-- Rules common for master and page documents: -->
<pattern name="common">

<rule context="int | link[@linktype='internal']">
  <assert test="@link">
    An internal link must use a 'link' attribute to specify the
    page being linked.
  </assert>
</rule>

<rule context="p">
  <report test="(normalize-space(text()) = '') and not(*)">
    A paragraph cannot be empty. If you want to increase vertical
    spacing here, modify the stylesheet.
  </report>
</rule>
```

```
<!-- Dummy rule listing all defined element types: -->
<rule context="
    block | block-process | blocks | button | buttons |
    contact-webmaster | copyright | environment | em | ext | head |
    html-title | img-path | int | item | label | lang |
    language-switch | languages | link | mailto | menu | os |
    out-path | p | page | page-footer | site | section | src-path |
    subhead | target-path | title | translation"/>

<!-- Report error if an element was not matched by the above: -->
<rule context="*">
  <report test="true()">
    Unrecognized element: '<name/>'.
  </report>
</rule>

</pattern>

<diagnostics>
  <diagnostic id="label-element">
    Every 'item' element must contain exactly one 'label' element
    specifying the corresponding top menu label.
  </diagnostic>
</diagnostics>

</schema>
```