Single-Source Publishing for Multiple Audiences with FrameMaker+SGML and Perl

by Alan R. Houser

About the Speaker:

Alan R. Houser is principal partner of Group Wellesley, a Pittsburgh, PA-based firm that provides documentation and content management services to technology-based industries. Houser holds degrees in electrical engineering and professional writing from Carnegie Mellon University, Pittsburgh, PA. He has more than ten years of technical writing experience, including five years working with FrameBuilder and FrameMaker+SGML. He has maintained templates, SGML applications, and documentation production tools for the past three years. Houser can be reached at arh@groupwellesley.com, or at 412-363-3481.

About this paper

You should consider this paper (and accompanying presentation) to be a case study. All but the most trivial forms of single-source publishing present problems that are difficult and time-consuming to solve. For example, the documentation process upon which this paper is based took several person-months to design and implement.

This paper attempts to accomplish the following:

- Provide an overview of one particular single-source publishing solution.
- Provide enough information to assess whether FrameMaker+SGML and Perl offer a viable solution for your single-source publishing needs or your SGML-to-HTML conversion needs.
- Provide enough information so that you might use components of our process to meet simpler publishing needs. For example, the Perl processing that I describe here is unnecessary if you are not generating HTML, or are generating HTML via a different mechanism.
- Provide enough information so that your implementation might proceed more quickly than did ours.

To use the material presented here, you will need the following requisite knowledge:

- Experience as a FrameMaker+SGML Application Developer — the ability to create and maintain SGML applications in FrameMaker+SGML. You must have the requisite knowledge to maintain, at minimum, the following files: FrameMaker+SGML read/write rules, SGML document type definition (DTD), FrameMaker+SGML element definition document (EDD), and FrameMaker template.
- Some programming experience is necessary to implement what I discuss here. Before you use Perl, you will have to learn (at least some) Perl. Access to somebody with Perl experience would be most helpful.

The processes and capabilities that I describe here do *not* require the use of the Adobe Frame Developers' Kit (FDK). The Frame Developers' Kit is an application programming interface for customizing FrameMaker and FrameMaker+SGML.

Problem Statement

My employer (CLARITECH Corporation, Pittsburgh, PA) has developed a C++ Toolkit for creating text search, summarization, and routing applications. The Toolkit offers capabilities in such areas as natural-language processing of text queries, query augmentation (automatically adding related terms to a user's query), and automatic extraction of keywords from large bodies of text. The Toolkit consists of more than 400 public header files, which define more than 600 public classes and more than 1000 public methods.

Documentation for this Toolkit must serve several different audiences. At one end of the audience spectrum are customers who wish to develop relatively straightforward applications. These customers will use existing Toolkit functions. For advanced programmers, the Toolkit offers the option of customizing the Toolkit for particular needs. These advanced programmers will need a much more thorough set of Toolkit documentation, that describes many C++ classes and methods that the "off-the-shelf" customers will not need.

The documentation for this Toolkit must also serve the company's internal developers — those who are responsible for maintaining and extending the Toolkit's functionality. These developers must be provided implementation details about the Toolkit; information which is considered proprietary and company confidential. This information must be available to internal developers, but must not be published to external customers.

Goals

My company and documentation group had the following goals with respect to our single-source publishing initiative:

- Maintain document sources in SGML.
- Use "open" (non-proprietary) solutions as much as possible.
- Maintain all Toolkit documentation in a single source tree.
- Maintain (typically reference) documentation for any particular Toolkit function in a single source file.
- Publish in HTML (our main delivery mechanism) or hardcopy/PDF.
- Support publishing documentation versions for several distinct audience types.

Solution

The following is a summary of our single-source publishing process:

- Maintain documentation source files in SGML. Use Microsoft Visual SourceSafe as a source-control repository.
- Use FrameMaker+SGML to author and maintain documentation source files. Use an EDD/DTD that has been modified to allow writers to identify different types of information within a single document.
- Use customized read/write rules to publish hardcopy/PDF.
- Use Perl scripts to convert SGML source files to HTML. Design Perl scripts to include or exclude information depending on type of audience for whom we are publishing.

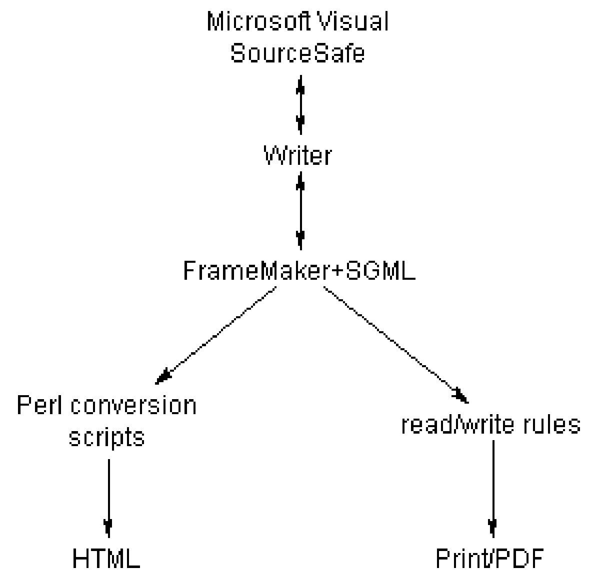

Figure 1 Publishing Work-flow

Adding Audience Information to your SGML

Options for Identifying Audience-Specific Information in SGML

Consider the SGML fragment <p>Here is a secret</p>, which we wish to label as "confidential." SGML provides three ways to do this:

- Marked sections. The SGML specification defines a special construct called a marked section. A SGML marked section begins with "<![" and ends with "]>". The SGML specification allows you to name particular marked sections, and either include or exclude named marked sections from your final document. For example, the following marked section could be included or excluded when an

SGML document is processed, depending on whether you wished to include or exclude sections labeled “confidential”:

```
<![ %confidential; [<p>Here is a
secret</p>]]>
```

- Wrapper Elements. Define SGML elements that correspond to each audience for which you are publishing. For example, the following fragment shows the use of a wrapper element “confidential”:

```
<confidential><p>Here is a
secret<p></confidential>
```

- Element Attributes. Element attributes provide a mechanism for embedding additional information about an element or the contents of an element in your SGML source file. Element attributes can be used for labeling audience-specific text. For example, the following <p> element has an attribute named “audience”, which is set to the value “confidential”:

```
<p audience="confidential">Here is a
secret</p>
```

Advantages and Disadvantages of each Method

Each of these methods for “marking” text has particular advantages and disadvantages:

- Marked sections — present perhaps the easiest way to identify audience-specific sections in an SGML document. However, FrameMaker+SGML does not support the import or export of marked sections.
- Wrapper elements — are good for identifying arbitrary groups of elements, such as one or more consecutive paragraphs. Wrapper elements are generally very easy for authors to apply in FrameMaker+SGML. However, wrapper elements can complicate your DTD if you allow them in too many contexts.
- Element attributes — are a good solution for identifying top-level elements, or elements that encapsulate relatively large sections (including many child elements). However, element attributes are somewhat tedious to set or change in FrameMaker+SGML. For example, your authors are likely to rise against you if they must set an audience-identifying attribute on every paragraph.

Our Solution

We chose a combination of wrapper elements and element attributes, which we thought were optimal for our writers.

We chose to add an “audience” attribute to the following elements:

- The highest-level SGML element that defines each class reference page.
- The SGML element that wraps functions defined by a class.
- The SGML element that wraps data types defined by a class.

Thus, we could label an entire reference page for a particular audience by setting a single attribute value on the parent element for the reference page. Likewise, we could label any particular function or data type defined by a reference page by setting a single attribute value on the element that wraps the function or data type.

For labeling individual or consecutive paragraphs, we chose to define wrapper elements. By “wrapping” a paragraph or consecutive paragraphs in a wrapper element, the writer would define those paragraphs as being appropriate for a particular audience.

Modifying Your DTD and EDD to Include Audience Information

If you are designing a single-source, multiple-audience publishing system based on FrameMaker+SGML, you must modify your DTD and EDD to include your new audience-specific wrapper elements and/or element attributes.

Adding Wrapper Elements to your DTD/EDD

When adding a wrapper element to your DTD and EDD, you must define an SGML *content model* for the new element. (*Content model* is SGML terminology for the definition of the element's contents.) You must also modify the content models of legal parents of the new wrapper element. The following questions will help you to define the content model of any wrapper elements:

- In what contexts (parent elements) can the wrapper element appear?
- What child elements are legal within the wrapper element?

In addition to these questions, you must pay particular attention to the following constraint: When you remove the wrapper element and its contents from a document, the document must still be valid!

Although authoring and maintaining an SGML DTD and FrameMaker+SGML EDD is beyond the scope of this paper, the following DTD fragment illustrates a possible content model for the element "confidential":

```
<!element confidential o o ( p+ )>
```

This DTD fragment defines the SGML element "confidential", which may contain one or more <p> elements. Remember that you must modify the content model for elements that might *contain* the <confidential> element!

Likewise, the following EDD fragments define the same FrameMaker+SGML element "Confidential":

Element (Container): Confidential
General rule: (P)+

Adding Element Attributes to your DTD/EDD

The questions that you must address when adding element attributes to your DTD/EDD are slightly different than those for adding wrapper elements. You will not modify the content model of the elements to which you add audience-identifying attributes. You will, however, have to determine the following characteristics for the attribute that you are adding:

- possible values for the attribute
- whether the attribute is required or optional
- the default value of the attribute, if you wish to specify a default value

Just as for wrapper elements: when you remove the element that you are identifying with a special attribute, the document must remain valid! If it is possible that removing an audience-specific element will cause a document to no longer be valid, you may need to modify the content model of any parent elements, or reconsider to which elements you will add an audience-specific attribute.

Using Read/Write Rules to Process Your SGML Source

FrameMaker+SGML allows you to modify the default behavior when importing or exporting SGML, via read/write rules. These rules are typically defined in the file *rules* in your SGML application directory. You can use customized read/write rules to publish your SGML source files for a particular audience, as follows:

1. Define wrapper elements and/or element attributes to identify your audience-specific information.
2. Define an SGML application for each audience for whom you wish to publish. Each SGML application will be the same as your "normal" SGML application, except that it will include an appropriate "drop" rule in the read/write rules.
3. Use the modified SGML application to import SGML source files for publishing.

For example, the following read/write rule will drop the element tags and contents (including child elements) of the "confidential" element:

```
element "confidential"
 {
 drop;
 }
```

You might include this read/write rule in a special SGML application (perhaps called "no_secrets"). SGML files that are imported into FrameMaker+SGML using this special SGML application will not include content that had been tagged "confidential".

This option presents a simpler, although perhaps less flexible, mechanism for single-source publishing than using Perl to process your SGML files. However, it does offer the capability to publish both hardcopy/PDF files, as well as to publish for the World-Wide Web via FrameMaker+SGML's HTML export capabilities.

Converting SGML to HTML with Perl

This section describes some of the characteristics that make Perl processing of SGML relatively straightforward, and provides a procedure for identifying the mapping between your SGML elements and HTML tags. I discuss specifics of the Perl language in later sections of this paper.

Considerations

HTML is an SGML document type definition — the "rules" for legal HTML are specified in SGML. Because HTML is an SGML document type, HTML and SGML share the following characteristics:

- Elements that (should) identify semantics of text within a document. Elements are (or should be) used within a particular context.
- Each element has a particular begin tag (i.e. <body>) and end tag (i.e. </body>).
- Elements may be nested, but end tags must be in inverse order of begin tags. The SGML fragment "<body>some text</body>" is legal; the SGML fragment "<body>some text</body>" is not (note that the closing "</body>" and "" tags are transposed in the second fragment).

FrameMaker+SGML and Markup Minimization

The SGML specification allows for something called "markup minimization." Each element defined in an SGML DTD can allow or forbid markup minimization. If an element allows markup minimization, that element can be closed by a "minimized" (empty, except for the backslash) tag: "</>". Markup minimization can decrease the length of SGML source files, but complicates writing parsers.

For example, the following is a legal markup-minimized SGML fragment:

```
<sentence>The <thing>cow</>
<action>jumped</> over the
<thing>moon</></>.
```

FrameMaker+SGML does not support markup minimization; it will not export markup-minimized SGML. (FrameMaker+SGML will, however, successfully import markup-minimized SGML). FrameMaker+SGML will close each start tag with a full end tag. For example, FrameMaker+SGML will export the previous SGML fragment as:

```
<sentence>The <thing>cow</thing>
<action>jumped</action> over the
<thing>moon</thing></sentence>.
```

If you are processing "markup-minimized" SGML, like the previous example, your processor must determine to which start tag each "</>" tag belongs. Because FrameMaker+SGML does not export "markup-minimized" SGML, you will never have to figure out which start tag a minimized end tag belongs.

Identify mapping between SGML and HTML tags

We used the following process to create a mapping between SGML element names and HTML tags:

1. Mark up an example document in HTML. Create (by hand-tagging or by an GUI-based HTML authoring tool) an example version of a typical document in HTML.
2. Compare the SGML source for your sample document to the HTML source that you have created. Identify mappings between the SGML and HTML tags, as follows:
   - Identify one-to-one mappings. These will be simple to transform in Perl.
   - Identify context-sensitive mappings. These will require more complex Perl programming.

- Identify cases in which attribute values are important to your HTML layout. These will also require more complex Perl programming.

3. Identify any "boilerplate" information that you want to include in the header and/or footer of your HTML file. Examples include: HTML declaration, HTML style sheet information, company name, copyright date, software version information, and document source file creation or modification date. You will probably add boilerplate information to your HTML files via simple Perl "print" statements.
4. Begin writing your Perl conversion script. It is probably best if you start with simple tasks, such as opening your source file, performing one-to-one transformations, and writing the results to an output file. This way you can see encouraging (we hope!) results quickly.

Extending your Processing Scripts

As you develop your conversion scripts and become more proficient with Perl, you may want to add additional functionality to your Perl conversion scripts. For example, you may wish to:

- Automatically create index pages (table of contents, list of reference pages, list of functions).
- Resolve cross-references. This may require that you parse each SGML file twice (once to build a list of cross-reference targets; the second time to insert the cross-reference text and target link).

You are likely to have other ideas for extending your Perl conversion scripts as you develop them.

About Perl

Perl (an acronym for Pattern Extraction and Replacement Language, but by convention is presented as proper noun) was designed to identify, extract, and replace patterns in text files. Because Perl was originally designed to process text files, Perl handles tasks such as opening existing files, reading file contents, replacing text patterns within a file, and writing output to new files fairly easily.

Perl Regular Expressions

Perhaps Perl's greatest utility comes from its ability to identify and replace *regular expressions*. Regular expressions can be thought of as patterns or rules that match character strings in a text file. When a regular expression "matches" a string, your Perl program can perform an action on that string. For example, your Perl script might delete the string, replace the string with a different string, or even save the string for later use. Any Perl script that performs text conversion (such as an SGML-to-HTML converter) will be based on Perl's regular expression capabilities.

A regular expression can include the following types of characters:

- Literal characters (example: the string "<body>").
- Special characters, used to identify (for example):
  - categories of characters (alphabetical, numerical, white space)
  - negation of characters (example: any non-numeric character)
  - ranges of characters (example: any digit between 3 and 7, inclusive)
  - negation of character ranges (example: any character that is not a digit between 3 and 7)
  - number of characters (zero or one, zero or more, or at least one)

The following is Perl syntax for matching a regular expression and replacing the matched text:

```
s/pattern/replacement/modifiers;
```

For example:

```
s/<body>/<p>/g;
```

will replace <body> with <p>. The "s" at the beginning of the instruction tells Perl to replace the matched text; the "g" at the end of the instruction specifies to replace all instances of <body>. Without the "g", the instruction would only replace the first occurrence of <body>.

Perl is freely available (and free!) on several computing platforms. For Perl availability, see the Comprehensive Perl Archive Network (CPAN) web site, at www.cpan.org. A complete Perl distribution is also included on CD-ROM with several Perl manuals, including *Perl for Dummies* (Paul Hoffman, 1998). Perl has one of the larger user communities of any programming language, so there is likely to be a Perl guru available near you.

It is also notable that Perl is interpreted, not compiled, at run-time. This means that you can test changes to your Perl scripts right away, without waiting for your Perl program to compile. (Compiled languages do tend to run faster than interpreted languages, but I have rarely felt that a Perl script ran too slowly).

Perl Gotchas

This section discusses some of the characteristics of Perl that can be particularly troublesome for beginners.

There's More Than One Way To Do It

You may see the letters "TMTOWTDI" occasionally in Perl documentation. These letters stand for "There's more than one way to do it" and mean exactly that. As with the UNIX text-processing tools that preceded Perl, there is usually more than one way to accomplish any given task in Perl. In fact, you may discover different, or even easier, ways to accomplish the tasks described in this paper.

Yes, I really mean "/"

Perl uses the backslash ("/") character to separate the "match" and "replacement" strings in a transformation operation. (This is actually not always true—remember TMTOWTDI?). But suppose you want your match or replacement strings to include a backslash? You do so by escaping the backslash, which means to precede the backslash with a forward slash ("\/"). The forward slash tells Perl to treat the following backslash as a real character, not a Perl separation character.

Greedy vs. non-greedy matches

By default, a Perl regular expression matches the longest possible string. For example, assume that we apply the following regular expression operation:

```
s/<body.*>/<p>/g;
```

(Match the string "<body", followed by zero or more of any other characters, followed by ">". Replace match with "<p>".)

to the following SGML fragment:

```
<body align="right">Mary had a little
lamb</body>.
```

The result would be:

```
<p>.
```

That is correct! The regular expression matches the *entire* fragment, and replaces the matched text with "<p>".

To disable greedy matching, use the "?" modifier after character classes in your regular expression. For example, the previous example would become:

```
s/<body.*?>/<p>/g;
```

(Match the string "<body", followed by zero or more of any other characters, followed by ">". Replace match with "<p>".)

The result of the non-greedy operation would be:

```
<p>Mary had a little lamb</body>.
```

Your Perl script can then transform the closing </body> tag by the following operation:

```
s/<\/body>/<\/p>/g;
```

Resulting in:

```
<p>Mary had a little lamb</p>.
```

Perl Examples

This section provides example fragments from a Perl SGML-to-HTML conversion script.

Open the text file refpage.sgm for processing

```
open (IN, refpage.sgm);
```

Replace all instances of SGML tag <body> with HTML tag <p>:

```
s/<body>/<p>/g;
```

Replace all instances of SGML tag </body> with HTML tag </p> (note the escaped backslash character):

```
s/<\/body>/<\/p>/g;
```

Replace SGML tag <body>, which may have one or more attributes, with HTML equivalent <p>. Do not translate attributes or take action based on attribute values:

```
s/<body.*?>/<p>/g;
```

The following is a somewhat more complex regular expression. It essentially says: match the string "<confidential>", followed by the minimum number of characters that do *not* match the string "<confidential>", followed by the string "<\/confidential>". Simply put, this regular expression will strip the <confidential> begin and end tags and their contents:

```
s/<confidential>
   (?:(?!<confidential>).)*?
   <\/confidential>//msg;
```

This regular expression is similar to the previous example. Check attribute value on element <function>. If attribute "audience" is set to "confidential", remove element and contents.

```
s/(<function ((?:(?!audience ?=
   ?).)*audience ?=
   ?\"confidential\".*?)
   <\/function>)//msg;
```

Complete Perl Conversion Example

Consider the following SGML fragment, which contains the marked-up text of a (trivial) function reference page:

SGML fragment

```
<function>
<funcname>printstring</funcname>
<synopsis>printstring "string"
</synopsis>
<description>
<para>The printstring function prints a
string.</para>
</description>
```

The following Perl regular expression transformations would convert the reference page to valid HTML:

Perl processing script

```
# replace begin tags
s/<function>/<html>\/n<body>/g;
s/<funcname>/<h1>/g;
s/<synopsis>/<pre>/g;
s/<description>/<hr>/g;
s/<para>/<p>/g;
# replace end tags
s/<\/function>/<\/body>\/n<\/html>/g;
s/<\/funcname>/<\/h1>/g;
s/<\/synopsis>/<\/pre>/g;
s/<\/description>//g;
s/<\/para>/<\/p>/g;
```

And the resulting HTML:

HTML result

```
<html>
<body>
<h1>printstring</h1>
<pre>printstring "string"/pre>
<hr>
<p>The printstring function prints a
string.</p>
</body>
</html>
```

There are a few notable things about this example:

- Note the one-to-one correspondence between SGML and HTML tags. This example did not require any sort of context-specific processing. In real-world applications, you will probably find that most SGML tags have a single HTML equivalent, but you are likely to have instances in which the replacement HTML depends on the context of the SGML tag.
- We did not do any audience-specific processing here. In other words, we did not exclude any text based on an audience-identifying wrapper element or element attribute value.
- Note that the <description> element was replaced by an HTML rule element (<hr>). Note also that the closing <\/description> element was simply deleted. In most instances, the replacement for the beginning and ending tags will be identical (except for the backslash character). However, you will occasionally want to perform a different transformation operation on the beginning tag than on the ending tag of an element.

Resources

IDG Books Worldwide

- *Perl for Dummies* by Paul Hoffman, Second Edition, 1998

O'Reilly & Associates

- *Learning Perl* (the Llama Book) by Randal L. Schwartz and Tom Christiansen, Second Edition, 1997
- *Perl Programming* (the Camel Book) by Larry Wall, Tom Christiansen, and Randal L. Schwartz, Second Edition, 1996
- *Perl Cookbook* (the Ram Book) by Tom Christiansen and Nathan Torkington, 1998
- *Mastering Regular Expressions* by Jeffrey Friedl, 1997

Wordware Publishing, Inc.

- *Practical Guide to SGML/XML Filters* by Norman E. Smith, 1998

Internet Resources

CPAN (Comprehensive Perl Archive Network)

- www.cpan.org

Perl home page

- www.perl.com/perl

Perl Institute

- www.perl.org

Internet Newsgroups

- comp.lang.perl.announce
- comp.lang.perl.misc