Creating Custom SGML DTDs for Documentation Products

Bradley C. Watson, Keith Shafer
OCLC Online Computer Library Center, Inc.
watson@oclc.org, shafer@oclc.org

Abstract

A case is presented for cost-effectively creating custom DTDs for an organization by non-SGML experts using tools that automatically create DTDs from tagged text. Such tools make it practical and easy to create DTDs without having to hire consultants or invest heavily to develop internal SGML expertise. The specific tool focused on is the *SGML Document Grammar Builder*, a tool developed at OCLC Online Computer Library Center, Inc.

Background

The Standard Generalized Markup Language (SGML) is a meta language for writing Document Type Definitions (DTD) [2]. A DTD describes how a document conforming to it should be *marked up*: the structural tags that may occur in the document, the ordering of the tags, and a host of other features. Simply put, a DTD describes a class of tagged documents in a vendor-independent way.

Many organizations have made, or will soon make, the commitment to use SGML for their documentation. Determining to use SGML for all or part of an organization's documentation products is only the first of many decisions that must be made in order to use SGML. One key decision revolves around the use of Document Type Definitions: to use or not to use a standard DTDs?

This is not a simple decision for many reasons. Economically, this decision will impact an organization for many years, potentially costing or saving a large amount of money. Functionally, this decision will impact the ability of an organization to use its text data fully and efficiently.

We believe that custom DTDs are ultimately the superior approach for most organizations. Therefore this paper presents a case for custom DTDs and tools to create them based on the following premises

1) standard DTDs are not sufficient for all documentation needs,
2) custom DTDs can be difficult and costly to develop manually,
3) tagging sample documents is sufficient to allow for the automatic creation of DTDs,
4) automatically created DTDs reduce the cost of creating custom DTDs and thus free organizations to meet their documentation needs using SGML.

SGML and Grammars

In order to understand the nature of the custom DTD problem and our proposed approach to handling it, one must have at least a superficial knowledge of SGML and grammar notation. Specifically, one should know

1) how document structure is marked up in SGML,
2) how tag attributes are specified in SGML,
3) how general entities are used in SGML, and
4) grammar notation for conjunctions, grouping, and repetition.

The following is provided as a background for the rest of the paper. Those familiar with these concepts may want to skip to the next section.

Tagged Document Structure

SGML is used to guide the markup of document *structure* with *tags* that are clearly distinguishable from document *content*. Generally speaking, the beginning of a structure in an SGML document is marked by a *start tag* and the end of the structure is marked by an *end tag*.

A *start tag* is the character '<', followed by the tag name, followed by a '>' (e.g., <author>). An *end tag* is a '<', followed by a '/', followed by the tag name, followed by a '>' (e.g., </author>). For example, one might see this text

DOC 95-10/95 Savannah, Georgia, U S.A
© 1995 ACM 0-89791-713-8/95/0010.. $3 50

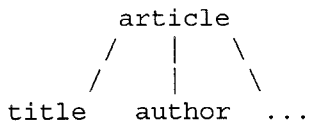
and markup in a document which means that the structure ***author*** has the content *John Smith*:

```
<author>John Smith</author>
```

Structures can be nested. For instance, ***author*** and ***title*** reside inside of ***article*** in the following:

```
<article>
  <title>A Short Document</title>
  <author>John Smith</author>
  ...
</article>
```

A tree representation of the ***article*** looks something like:



Structural components like ***author***, ***title***, ***name***, ***extension***, and ***equation*** in the above are commonly referred to as document ***elements*** in SGML and this paper. To reduce the cost of inserting SGML tags in documents, completely paired SGML tags need not always appear where structure is clear. That is, a DTD may specify which tags can be omitted.

Consider the following markup:

```
<name>John Smith
<extension>5555</extension>
```

Could ***</name>*** be inserted before ***<extension>***? Probably, because the semantic idea of a ***name*** generally encompasses a string like *John Smith*, but not a telephone extension number. For that reason, it makes sense to omit ***</name>***, allowing the ***<extension>*** start tag to imply the ending of the ***name*** structure. Other times, this would not be so.

For instance, one might have a math equation inside a paragraph as an inline structure, and also allow an equation outside of a paragraph as a display structure. If the same tag, ***<equation>***, is used to mark all equations, one cannot automatically assume that a paragraph has ended because an equation has begun. For this reason, and others, it is often best to fully tag documents with start and end tags.

SGML Attributes

Essentially, SGML tag *attributes* can be thought of as values that are associated with a given element start tag. For example, the tag ***author*** in the following has the attribute *primary* with the value of *yes*.

```
<author primary=yes>John Smith</author>
```

This markup shows that *John Smith* is the *primary author*. It does not indicate what an application should do with this additional information. For instance, an application might make this name appear first, or extract it for some abstracting service.

SGML Entities

SGML entities are like textual variables and pointers. Basically, an SGML entity begins with an ampersand '&' and ends with a semi-colon ';'. For instance, a document could contain the entity reference ***&food;*** which might be replaced by the word *pizza* in the translated text.

Grammar Notation

The structure of a document can be written as a *grammar* where each *element definition* is a *grammar rule*. In this paper, we will write element definitions (grammar rules) as:

```
ELEMENT  ⇒  DEFINITION
```

An element definition may include ANDs, ORs, and parentheses where AND is implied by juxtaposition and OR is signified by a vertical bar '|'. We assume that AND has a higher precedence than OR and thus will often not place parentheses around ANDs. For example, if A's definition is

```
( B AND C ) OR ( D AND E )
```

we will write:

```
A  ⇒ B C  |  D E
```

Finally, we will refer to a complex element definition (rule) as being made up of *subrules*, structural components between conjunctions. For example, 'B C' and 'D E' are subrules in A's definition above. Furthermore, 'B' and 'C' are subrules of 'B C' and 'D' and 'E' are subrules in 'D E'.

An element definition may use three different repetition symbols to signify that the structure in question has *zero or one*, *one or more*, or *zero or more* repetitions: ?, +, and *, respectively. For instance, assume that an ***article*** has a ***title***, one or more ***authors***, and a ***body***. The structural definition

```
<!DOCTYPE ARTICLE [
<!ELEMENT ARTICLE  - - ( TITLE, AUTHOR+, BODY )>
<!ELEMENT TITLE    - - (#PCDATA)>
<!ELEMENT AUTHOR   - - (#PCDATA)>
<!ELEMENT BODY     - - (#PCDATA)>
]>
```

Figure 1 - Article DTD

of ***article*** can be written as:

```
article ⇒  title  author+  body
```

As a point of comparison, a DTD corresponding to the ***article*** definition above is presented in Figure 1. The document type (***doctype***) described in the DTD fragment is ***article***. An ***article*** contains a ***title***, one or more ***authors***, and a ***body***. The ***title***, ***author***, and ***body*** contain text data.

Since a goal of this paper is to get people to think about document structures in grammar rule terms, we will not spend much time describing DTD syntax. Generating a DTD is simply a syntactic change from grammar rule notation to DTD notation. Those that want to edit DTDs to incorporate more SGML functionality are assumed to already understand DTD syntax.

That said, we would like to note that ANDs are represented by commas in DTDs, and the "- -" in the above DTD fragment tells us that the *start tags* and *end tags* of the elements are not optional. That is, they are required. Finally, **#PCDATA** is an SGML keyword that basically means data content.

Document Type Definitions -- Costly, Necessary

Standards committees are producing DTDs for a variety of text types. Yet many organizations are creating their own DTDs to fit their specific text products. The goal of the former activity is to eliminate the latter activity. The rationale is that, one, DTD construction is a costly and difficult process and, two, standardization of text data across organizations is worthwhile.

But this standardization does not appear to be a practical goal. Standard DTDs are rarely sufficient for local needs, thus utilization of a standard DTD may lessen an organization's ability to fully and effectively use its text data, which is the primary motivation for most organizations' using SGML. However, the cost of custom DTD creation is not trivial, in time or money. Traditionally, a DTD is created after a long period of manual analysis of sample documents to be covered by the DTD. This is followed by a period in which the actual DTD is created based on the structural components identified in the analysis phase. Both processes are best done by people who understand both the documents and SGML. However, few organizations have staff with both sets of expertise when they make the decision to go with SGML. One solution is to hire consultants to work with in-house staff to perform the analysis and the DTD creation. The alternative is to train in-house staff in the details of SGML.

When faced with the prospect of hiring consultants or ramping up their own expertise in SGML, most organizations find neither alternative attractive. Three or more months to create a DTD is not unusual. Consultants are costly for that kind of time. Add in the senior staff support time, and the cost quickly goes up. Conversely, training staff members in DTD construction can be equally costly, because it will take even more senior staff time away from the task of producing text, which is their real job.

The trade-off between need and cost is never a simple one to negotiate. Yet not putting an organization's documentation or other text data into a standard, logical markup based on SGML will also be a costly decision. It appears that the best tools for creating, storing, distributing, and presenting text will all be SGML-capable. Even simple word processing systems will soon be SGML-capable. Not using these tools will be very expensive in the long run.

DTDs -- The Needs at OCLC and IDI

OCLC Online Computer Library Center, Inc. (OCLC), a nonprofit computer library service and research organization, and its for-profit subsidiary, Information Dimensions, Inc. (IDI) are both committed to using SGML. However, this commitment is tempered by the problem of costs related to DTDs.

OCLC -- Its DTD Needs

At OCLC, production cost is a very important determinant, since the members of OCLC are all libraries, which have traditionally tight budgets. However, libraries have a need for high quality products, given the expected life-span of their holdings and the usage needs of their patrons.

OCLC's mission is to provide libraries a variety of products and services with the goal of lowering the rate of rise of the costs incurred by libraries while sustaining or expanding their capability to serve their patrons' needs. Electronic documents are fast becoming a necessary part of every library's set of offerings, so OCLC is endeavoring to meet this need as inexpensively, but with as high a quality, as possible. SGML has been adopted as part of that strategy. However, while SGML certainly aids in reaching the quality goal, it can be expensive.

For example, a large amount of the data that comes to OCLC for electronic distribution is in some form of tagged text, but frequently, there are no DTDs associated with the documents. Thus, the cost of quickly and efficiently creating DTDs that are correct for each set of documents is a large consideration. Using a standard DTD is out of the question, since most of the document collections do not conform even remotely to one, even if their structure is such that they could have been marked in accordance to a standard DTD. OCLC must use the documents as they are, not as they could have been.

Clearly OCLC needs a way to cost effectively create DTDs after the tagged text has been created. In other words, OCLC needs a tool that makes a DTD match a set of tagged documents.

IDI -- Its DTD Needs

Like OCLC, IDI's commitment to SGML is a strong one. First, IDI is a leading vendor of database products and applications that support SGML documents. Second, IDI views SGML as a superior approach for delivering its system documentation in multiple presentation formats. The latter meant, of course, that a move to SGML for the IDI documentation department was a top priority.

However, when IDI first made the decision to move their documentation to SGML, the problem of standard versus custom DTD was very much in evidence. A standard DTD, such as ISO 12083, [3] was tempting. Implementing the ISO 12083 DTD guarantees compatibility with all other documents and systems that conform to it. More importantly, the costly effort of designing and building a DTD from scratch is saved. However, no matter how much thought a committee puts into designing a DTD, it cannot be a perfect fit for every organization's documents. A standard DTD is the product of compromise, so important elements or structures vital to one's own documents are inevitably not included. Designing a custom DTD ensures that an organization's documentation needs will be met.

Thus, IDI leaned towards implementing SGML via custom DTDs for their documentation products. However, before committing to that path, they wanted to do a prototype DTD to get a feel for the effort involved and the value gained over using standard DTDs.

As it happened, because of OCLC's needs as discussed above, OCLC recently invented a tool that can make DTD creation reasonable, both in cost and time.

DTDs -- Their Automatic Creation

Motivated by OCLC's problem of tagged text with missing DTDs, the OCLC Office of Research has an ongoing effort, the *SGML Document Grammar Builder* project, studying the manipulation of tagged text [5]. This project has resulted in the construction of a C++ engine library, the Grammar Builder Engine ("GB-Engine"), that can be used to automatically create reduced structural representations of tagged text (DTDs), translate tagged text, automate database creation, and automate interface design — all from sample tagged text.

To automatically create DTDs, the GB-Engine translates sample document structures into corresponding grammar rules. These grammar rules are then simplified through a series of reductions. Basically, these reductions attempt to decrease the number and complexity of *subrules* in each *rule*, while leaving intact the semantics captured in the more complex representation. The resulting generalized grammar is then output in DTD format.

The GB-Engine is embedded in a number of systems, with *Fred* currently being the most popular. *Fred* is an extended Tcl/Tk interpreter. Tcl is a complete string command language with variables, strings, and lists, while Tk is an X-based toolkit [4]. As a result, Fred is a complete interpreter/shell that has access to the GB-Engine objects that can be used to easily build X interfaces. It is important to note that Fred's basic functionality is actually provided by the GB-Engine and is not restricted to Fred. For instance, the GB-Engine has been ported to the PC and there is some possibility of using Perl [8] and/or Scheme [1] as an

alternative to the Tcl portion of Fred. Fred is used for a number of translation tasks at OCLC [7] and several Fred services, including automatic DTD creation, are freely available via the WWW [6].

The desirability of the GB-Engine for OCLC with its large sets of tagged texts without DTDs is clear. However, such a tool is not only useful in the situation where tagged text exists, but even when no tagged or even electronic text exists as yet, as in the IDI prototype DTD project.

The reason one wants to tag text in SGML is to take advantage of all the document handling tools that are being designed to work with SGML-compliant tools, thus avoiding the costs and inconvenience of building, maintaining, and using proprietary software systems that will not work with other such systems. However, to use those SGML tools, such as databases and viewers, documents need to be tagged in accordance to a DTD.

Traditionally, a DTD is created for a document collection *before* the collection is built. This assumes that some sort of paper analysis has been performed to first identify important structure. As it turns out, however, this is actually a more expensive, time consuming approach than doing it the other way.

It is relatively simple for writers and editors to tag their documents to make information structure apparent using tags they *invent* as they go along, such as ***paragraph*** and ***section*** and ***title*** tags. Thus, given a tool such as the GB-Engine, the DTD can be extracted from a set of documents chosen to represent the collection and then tagged. This allows an organization to construct a custom DTD at much less cost and difficulty than could be done previously.

These premises were tested by OCLC's Office of Research as a result of IDI's need for a prototype documentation DTD. In fact, building the IDI prototype DTD turned out to be one of the earliest uses of the GB-Engine outside of the Office of Research. A prototype DTD was built for IDI with the GB-Engine using four sample documents supplied by the IDI documentation department. The next section describes the process involved.

Building a Prototype DTD Using the GB-Engine

To create a prototype DTD for IDI, four manuals of varying focus and style were chosen as the test data. Two manuals were descriptions of software application programming interfaces (APIs), one was a systems administration guide, and the fourth was a software program user guide.

To use the GB-Engine to automatically create a DTD, one or more document instances must be marked up with a target set of SGML elements. This set of elements may or may not already exist in whole or in part. Usually it makes sense to use the core elements from a standard DTD, such as the ISO12083, adding new ones only when necessary.

While IDI did not have a core set of standard tags, OCLC had prepared, some years before, a set of standard tags for use within OCLC, which served the purpose. This set of tags included elements for structures such as table of contents, section headers, paragraphs, and other very common logical structures.

Analyzing the test documents consisted of simply typing in the appropriate tags from the core set, while creating new tags to mark those logical structures that were unique to the IDI documents. Finding and creating tags for such structures is a large part of any document analysis process. Before GB-Engine, this was done manually, using such devices as sticky tabs to mark structures in the hard copy and note the suggested tags for each. Then one had to distill the information contained in the sticky tabs and notes into a semantically and syntactically correct DTD in relation to the documents. This process is very tedious and error-prone.

The difference when using the GB-Engine approach is that one simply needs to place tags, both start and end, in one or more electronic documents in their appropriate positions, then process the document(s) with the GB-Engine. If the tagging makes grammatical sense and captures all of the appropriate structures, a correct DTD will be generated. Thus, with the GB-Engine approach, one only has to find structures and tag them wherever they occur in the sample texts. The GB-Engine will get the logic of the DTD correct.

For example, the IDI software API manuals have a chapter containing function descriptions. However, the logical structures that are used in the descriptions vary from description to description and from manual to manual. These variations have to be captured in the logic of the grammar rules for the DTD. See Figure 2, below, for an example of such a description, Figure 3, below, for a tagged example of the text in Figure 2, and Table 1 for a list of the relevant ***elements*** representing each logical structure of the function description.

Looking at Figure 3, one can see that there is a structure tagged ***<fret>***. Not all function descriptions contained such

```
Add_SGML_Text_Data

Description This function adds SGML text data to the database. This function . . .

Syntax      ULONG  AddSGMLData(ID, . . .)

Returns     STAT_OK     No errors
            STAT_FAIL   Error attempting to add

Parameters  (Name)        (Type/Usage)

            ID          CONID Read

            Used to maintain process context.
```

Figure 2 - Untagged Function Description Text

```
<function rid=1>
<ftitle><bxt>Add_SGML_Text_Data</ftitle></bxt>
<fdscr>Description This function adds SGML text data to the database. This function . . .</fdscr>
<fsyn>Syntax ULONG  AddSGMLData(ID,  .)</fsyn>
<fret>Returns STAT_OK No errors STAT_FAIL      Error attempting to add</fret>
<fparms><h1>Parameters (Name)  (Type/Usage)</h1>
<fpparm><fpname>ID</fpname><fptype>CONID</fptype><fpuseage>Read</fpuseage>
<fpdscr>Used to maintain process context./</fpdscr></fpparm></fparms></function>.
```

Figure 3 - Tagged Function Description Text

```
<!doctype USERDOC [
<!ELEMENT USERDOC     - -  (DOCTI,FM,TOC,PF,CHP)  |  (DOCTI,FM,PF,TOC,INTRO,CHP+))>
.
.
.
<!ELEMENT FUNCREF     - -        (#PCDATA)>
<!ELEMENT FUNCTION    - -        (FTITLE,FDSCR,FSYN,FPARMS,FCOMMENT)  |
                                 (FTITLE,FDSCR,FSYN,FRET,FPARMS))
<!ELEMENT FTITLE      - -        (BXT)>
<!ELEMENT FDSCR       - -        (#PCDATA)>
<!ELEMENT FSYN        - -        (#PCDATA)>
<!ELEMENT FPARMS      - -        (#PCDATA,FPPARM+)>
<!ELEMENT FCOMMENT    - -        (#PCDATA)>
<!ELEMENT BXT         - -        (#PCDATA)>
<!ELEMENT FPPARM      - -        (FPNAME,FPTYPE,#PCDATA,FPUSEAGE,FPDSCR)>
<!ELEMENT FPNAME      - -        (#PCDATA)>
<!ELEMENT FPTYPE      - -        (#PCDATA)>
<!ELEMENT FPUSEAGE    - -        (#PCDATA)>
```

Figure 4 - USERDOC DTD

a structure. On the other hand, the description in Figure 3 does not contain a *<fcomment>* tagged structure, while other descriptions did. Figure 4, above, contains a fragment of the final prototype DTD created by the GB-Engine.

The definition for the element ***function*** shows that there are two possible combinations of elements that could be used in the IDI documents to build a function description section. In grammar notation:

```
FUNCTION ⇒
  (FTITLE FDSCR FSYN FPARMS FCOMMENT)
  |(FTITLE FDSCR FSYN FRET FPARMS))
```

To determine that these two variations of ***function*** exist, then describe their logical relationship in a DTD is not simple. To do this over and over through hundreds or even thousands of pages of text and hundreds or thousands of different structure combinations is nearly impossible to get correct, and is tedious, difficult, and time consuming. GB-Engine removes a great deal of these problems.

Table 1 - Function Description Elements

FUNCREF	Function Reference
FUNCTION	Function Description Section
FTITLE	Function Title
FDSCR	Function Description
FSYN	Function Syntax
FRET	Function Return Value
FPARMS	Function Parameters
FCOMMENT	Function Comment
FPPARM	Function Parameter
FPNAME	Function Parameter Name
FPTYPE	Function Parameter Type
FPUSEAGE	Function Parameter Usage
FPDSCR	Function Parameter Description

All together, the analysis/tagging of the four documents took one person approximately sixteen hours, or four hours each. It should be noted that the person doing the tagging had never done any document tagging before, in SGML or any other system, was not familiar with DTD construction or use, and had no familiarity with the documents or the systems they described prior to doing this experiment. This person, did, however, have extensive experience in using, designing, and building software systems, as well as using similar documents in relation to software systems.

In other words, using the GB-Engine approach to constructing a DTD does not require a large amount of SGML expertise. It does require a basic understanding of document structure in relation to the use of documents and how tags in the document can be used to mark such structures.

The next step after constructing the marked up documents was to run them through the GB-Engine to construct the desired prototype DTD. This took less than a minute on a Sun Unix Server. Figure 4, above, is a sample of the full DTD relating to the function description tags listed above.

The DTD built by the GB-Engine is not a final DTD, generally speaking. It is, however, a sound basis for making a final DTD. Manual tweaking is sometimes useful. For instance, the structure that the GB-Engine deduced for the ***function*** element was:

```
FUNCTION ⇒
   (FTITLE FDSCR FSYN FPARMS FCOMMENT)
   |(FTITLE FDSCR FSYN FRET FPARMS))
```

A reasonable reduction for this might be:

```
FUNCTION ⇒
   (FTITLE FDSCR FSYN FRET? FPARMS
   FCOMMENT?)
```

On the other hand, it might be that the two disparate element groups for ending a function description are important and must be kept separate. In that case, one might want to simplify the element rule this way:

```
FUNCTION ⇒
   (FTITLE FDSCR FSYN ((FPARMS FCOMMENT)
   |  (FRET FPARMS)))
```

This makes the rule shorter, while maintaining two distinct structural possibilities without allowing any other alternative structures which the first re-write would have allowed.

In terms of producing a final prototype DTD for IDI, many refinements were accomplished automatically by adding and

subtracting element tags from the documents, which were then run through GB-Engine again. Finally, some manual editing of the DTD was found to be necessary, largely to account for known grammatical possibilities that the four documents did not themselves contain. However, it is much simpler to tweak an existing DTD then to create one from scratch. The end result was that IDI had a workable DTD prototype in a very short amount of time, at a minimal cost.

Conclusion

The primary benefit of an automated DTD generation system such as the SGML Document Grammar Builder to documentation creation is that custom DTDs become very affordable. Not everyone has a need for custom DTDs, but for those who do, the cost of creating one manually can certainly be prohibitive. GB-Engine removes the cost roadblock to generating customs DTDs, which in turn allows a documentation organization the flexibility it requires to get the SGML markup of its documents right for their specific needs.

In this paper, besides motivating the need for the automatic creation of DTDs from sample tagged text, we have introduced the GB-Engine and Fred. We have successfully used Fred to analyze several tagged sources, fix markup errors, and translate tagged documents. We expect that others will find similar uses for Fred, or comparable tools, as SGML proliferates.

Software Information

The best way to get a feel for automatic DTD creation is to experiment with examples of your own text. Accordingly, OCLC makes several Fred-based services freely available to the community via the World Wide Web at:

http://www.oclc.org/fred/

Currently, these services include free automatic DTD creation, direct grammar reduction, and arbitrary text translation. After only five months of availability, the DTD creation service has already been used to generate over 2,100 DTDs.

REFERENCES

[1] Harold Abelson, Gerald Jay Sussman, and Julie Abelson. *Structure and Interpretation of Computer Programs*. The MIT Press, 1985.

[2] Information Processing -- Text and Office Systems -- Standard Generalized Markup Language (SGML). International Organization for Standardization. Ref. No. ISO 8879:1986, 1986.

[3] Electronic Manuscript Preparation and Markup. ANSI/NISO/ISO 12083, 1994.

[4] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Publishing Company, 1994.

[5] Keith Shafer. SGML Grammar Structure. In *Annual Review of OCLC Research July 1992-June 1993*, pages 39-40, 1994.

[6] Keith Shafer. Fred: The SGML Grammar Builder. Fred's WWW home page. Accessible at URL:http://www.oclc.org/fred/, 1994.

[7] Keith Shafer and Roger Thompson. Introduction to Translating Tagged Text via the SGML Document Grammar Builder Engine. Accessible at URL:http://www.oclc.org/fred/docs/translations/intro.html, 1995.

[8] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., 1992.