ELECTRONIC PUBLISHING, VOL. 4(1), 27–42 (MARCH 1991)

Considerations for the preparation of SGML document type definitions

SANDRA A. MAMRAK[1]
Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210

J. A. BARNES
Department of Computer Science
Bowling Green State University
Bowling Green, Ohio 43403

SUMMARY

The Standard Generalized Markup Language, SGML, is being adopted by various international organizations as the medium for exchange of electronically encoded documents. An exchange is accomplished by way of a Document Type Definition, DTD, that describes the content of documents targeted for an exchange. In this paper we suggest considerations for the designers of SGML DTDs. The considerations emphasize uniformity and simplicity without sacrificing expressive power. The considerations are not comprehensive: they address minimization features, attributes, inclusion and exclusion exceptions, and the CONCUR feature of SGML.

KEY WORDS Data representation Document Type Definitions Standard Generalized Markup Language

1 INTRODUCTION

There exists a huge store of electronically encoded data comprising a broad and varied collection of documents. A primary goal of creating electronic stores of these documents is to make them accessible to a wide audience for a variety of activities such as queries, online data delivery and display, data exchange, and data analysis. An obstacle to achieving this goal is the diversity and complexity of the underlying data representations.

A potential for significant improvement in this situation has arrived, with the advent of standardized data representations of documents as encouraged, for example, by the Standard Generalized Markup Language [1], SGML. SGML requires that a class of documents be specified by way of a Document Type Declaration, sometimes also called a Document Type Definition, or simply a DTD. Basically, a DTD defines the structural elements of a document, including rules for the ordering and occurrence of the elements.

Companies, government agencies, publishers, and academia are now defining DTDs. Companies such as Boeing, The World Bank Group, Chemical Abstracts Service, Mead Data Central, OCLC, Inc., Science Application International Corporation, Shell, and EDS are developing DTDs for inhouse use and for possible use by their suppliers as well.

The US Department of Defense STARS and CALS projects are both committed to encoding data using SGML. The CALS project is working with document types for weapon system logistic technical information [2]. The national medical community is considering adopting SGML as a standard representation for medical records, with support from the

[1] This work has been supported in part by a grant from the Applied Information Technologies Research Center, Columbus, Ohio

0894–3982/91/010027–16$8.00

Received 8 November 1989
Revised 21 November 1990

IEEE Medix, ACR/NEMA, and ASTM medical standards groups [3]. The Canadian Ministry of Defense has specified that DTDs be used for certain data exchanges. Members of the European community, as well, are using SGML to exchange data among themselves.

The Association of American Publishers has developed and published DTDs for documents, including one for document type article [4], one for mathematical equations [5], and one for tables [6]. Other publishers such as Elsevier and Springer-Verlag are using SGML.

In academia, the linguistics and humanities domains are using DTDs to define different sets of data. For example, an initiative is under way to create DTDs for types such as dictionaries, thesauri and corpora [7]. For each of these types, a separate DTD must be developed. Furthermore, within each type, there are many subtypes (e.g., the Thesaurus Linguae Graecae [8], Dictionary of the Old Spanish Language [9], and the Comprehensive Aramaic Lexicon [10]) that represent different sets of data to be described.

In response to this emerging international interest in SGML, vendors are beginning to provide tools to create and process SGML DTDs. Some vendors, e.g., ArborText, provide translators from their own internal format to an SGML format. Others, e.g., Datalogics, IBM, SoftQuad, Software Exoterica, and Yard Systems, provide tools for creating and validating DTDs. An industrial consortium has been formed to develop and distribute freely licensed SGML software [11].

Unfortunately, data access problems are not guaranteed to be totally eliminated by the use of SGML. SGML can be utilized in such a way as to leave data representations as diverse and complex as before. In particular, the representation for an SGML document as specified through a DTD allows for considerable, and we believe sometimes too much, flexibility.

In this paper we suggest considerations for the designer of DTDs that constrain certain allowable options. The considerations emphasize uniformity and simplicity in the data representation. The considerations are not comprehensive: they do not address all of the features of SGML. They are based on our extensive experience gleaned from writing software to process existing DTDs [12] and from consulting with agencies who are developing DTDs.

A primary principle that has been applied in the formulation of these considerations derives from software engineering: identify separate functions and design modular, possibly layered, systems to implement them. Widespread distribution, discussion, and possible acceptance of these considerations may encourage the application of this principle to other features of SGML. It may also support the ultimate goal of the standard, namely, easy and correct data access and exchange.

2 CONSIDERATIONS FOR DESIGNING SGML DTDS

Below we present a set of considerations for the use of minimization features, attributes, inclusion and exclusion exceptions, and CONCUR. The considerations are intended to assist those faced with the task of creating DTDs. A discussion and justification of each consideration is provided, though some discussions are longer than others. In particular, the discussions on our recommendations not to use minimization features and not to use attributes are lengthy because these recommendations are the most controversial.

Minimization features ease the task of creating and reading documents. Attributes give the DTD creator a means of dealing with semantic issues in a document. We would like

to clarify at the outset that when we advocate not using these features *as a part of the specification of a DTD*, we do not imply that the functions that these features are intended to serve are not important. On the contrary, these functions are so important that they should be addressed by separate mechanisms *outside of the DTD specification*. This approach not only controls the complexity of the DTD itself, but provides for more direct and effective implementation of the desired, added functionality.

2.1 Use a software development environment

The creation of DTDs must be undertaken as a cooperative effort by those who are expert in the application area for which the DTD is being defined, and those who are expert in the use of software tools and testing methods for specifications based on formal languages.

A DTD must enable the correct and useful encoding of information required by the text type, disciplinary domain, and particular processing task. Linguistic expertise, for example, will be required in writing DTDs for the encoding of linguistic analyses, and publishing experience is required to ensure that DTDs meet the needs of publishers. Though necessary, however, such domain-specific expertise is not sufficient to ensure the high quality of the DTD defined.

DTDs are intended as detailed machine-processable specifications, and so should pass the tests we commonly apply to other such specifications, notably computer programs. In our experience with DTDs written by the Association of American Publishers and others, we discovered that DTDs can easily become twice as long and twice as complex as some programming languages [13]. Furuta has also recognizes this degree of complexity in Reference [14].

For example, in order to compare the size and complexity of a DTD to a programming language, one DTD [4,5,6] was rewritten in the same notation as is the programming language Pascal. The notation uses productions, terminals and nonterminals instead of elements. A sample rewrite is shown below for an excerpt from a DTD describing a list as one or more items.

```
The DTD notation:

<!--       ELEMENT MIN  CONTENT  -->
<!ELEMENT  list    - -  item+      >
<!ELEMENT  item    - -  CDATA      >

The rewritten-DTD grammar:

list : "<list>"  item_plus  "</list>"
item_plus : item_plus  item
          | item
item : "<item>"  CDATA  "</item>"
```

The rewritten DTD was simplified to describe only the elements and their logical connections. It did not include other information like minimization specifications or inclusion and exclusion clauses. The rewritten-DTD grammar has 368 productions, 358 terminals and 236 nonterminals. This compares to 191 productions, 152 terminals and

108 nonterminals in Pascal. Further, a nonrecursive nesting level to reach a terminal from the root of the rewritten-DTD grammar can be as deep as 26, compared to 19 in Pascal. And some rewritten-DTD terminals and nonterminals are referenced in as many as 68 productions, compared to 17 in Pascal.

Because they can become long and complex, steps should be taken to control the complexity of DTDs and keep them comprehensible. Their syntax should be verifiably correct, and their semantics should be tested to ensure that the DTD defines what it is intended to define.

Software tools and testing methods have been established to deal with syntactic and semantic problems in computer programs. Compilers exist to verify syntax, and test data are run to raise confidence that the semantics are correct, i.e., that the program does what was intended by its author. These techniques are not foolproof, but they help considerably to eliminate all the obvious problems, and some of the more subtle problems, that arise when humans deal with complex objects.

If DTD definers do not have access to software tools and methods similar to those of the computer programmer, then the quality of their DTDs can be expected to be comparable to that of a computer program that has never been compiled or tested. The DTD is likely to be plagued not only by syntactic errors, but also by semantic ones.

From a syntactic point of view, DTDs are supposed to be written as context-free grammars. As such, they consist of terminal (e.g., CDATA) and nonterminal (e.g., elements like frontmatter, body or backmatter) symbols and a set of rules for the ordering and occurrence of the symbols. Typically, all symbols should be reachable from the start symbol.[2] Also, the grammar should terminate in the sense that there is at least one path to terminal symbols from the start symbol. As a DTD becomes large, it is very difficult for humans to determine if all symbols are reachable, i.e., if there are any symbols that are disjoint from the grammar, and if the grammar terminates.

Semantically, DTDs may be highly error-prone. SGML provides very powerful shorthand notations for specifying rules in a grammar (*parameter entities* are one example). The purpose of these notations is to make the specification of the DTD easier. Often, however, their effect may be to lead definers to write a specification that does not capture their intended meaning. Without some method of testing the DTD, it is very difficult for a definer to know if the DTD really captures the document type that was intended.[3]

Therefore, the successful definition of DTDs requires expertise both from the targeted application area and from the area of software tools and testing methods for formal language specification.

2.2 Do not use minimization features

The DTD definer should use none of the minimization features of SGML. In this section we confine our discussion to the OMITTAG minimization feature. The considerations are similarly applicable to the SHORTREF, DATATAG, and SHORTTAG features.

The minimization features in SGML have been included in the standard to ease the burden of document markup, inspection, and editing. An attempt to address the

[2] SGML, in an effort to support reuse of DTDs, does not require that all symbols be reachable from the root.

[3] In the AAP documentation of their grammar for tables [4], for example, the markup they use for their examples is not legal according to their formal specifications. In the formal specification they indicate that each cell in a table must contain a paragraph, but in the sample markup they do not include the required tags for paragraphs.

problem of document markup and viewing *in the specification of the standard itself* is counter to the software engineering principle of modularity. By this principle, the problem of creating user-friendly environments for document markup and viewing should be addressed separately from the already complex task of creating a definition for a document type. Thus, although minimization is an important and pressing issue, it should not be handled along with the creation of the DTD itself. Indeed, it is precisely and only outside of the SGML standard, in an array of auxiliary support tools for processing SGML, that such user-oriented, computing-environment issues can be addressed adequately.

The technology to support easy input of documents is rapidly changing. The minimization features in SGML were designed under the assumption that a particular kind of markup technology is in place. The assumption is that the markup system will require the author of a document to enter all the descriptive information about the content of the document, i.e., tags and possibly attributes of tags, in addition to the content itself, completely manually, and that only one view of the document will be available, that of the markup actually entered by the author. This assumption is based on a 'typewriter' view of computer systems that might be developed to support the generation of documents. Basically a 'blank sheet', i.e., empty display screen, is envisioned as confronting the author, and a keyboard is envisioned as the sole source of input to lay characters, both markup and content itself, on the blank sheet.

But a technology has been developed, and is constantly improving, that makes this older technology obsolete. Tag-driven, menu-based, document preparation systems that do not require an author to input any tags and that provide various views of the document, e.g., with tags or without tags, are available commercially.

Further, although these systems are relatively powerful, they are still viewed by some knowledgeable and skillful creators of SGML documents as too awkward to use and a bane to productivity. Thus, even these systems will eventually be streamlined, and even personally customizable, in order to accommodate the most adept data-entry personnel.

In view of the clear separation of the tasks of creating a DTD and of providing for an optimized environment to create instances of it, and in view of the emergence of a sophisticated technology to address the latter problem, we believe it is a design error to include the specification of minimization features in DTDs.

Moreover, because the task of minimization can be addressed only in the most peripheral manner in a DTD, the use of the minimization features can barely achieve the goals for which they were created, if at all. We illustrate the complexity involved in using and specifying the OMITTAG feature below.

The OMITTAG feature states that in some circumstances, an entire tag may be omitted from the marked-up document. There are four conditions that must prevail for a tag to be omissible. First, the tag must be omissible according to three SGML-specific rules. A somewhat simplified version of the SGML-specific rules is:

1. An end-tag may be omitted if its element's content is followed by the start-tag of an element that can not occur within it.
2. An end-tag may be omitted if it is followed by the end-tag of an element that contains it.
3. A start-tag may be omitted if the element is contextually required and any other element types that could occur are contextually optional.

Second, the omission of the tag may not create an ambiguity.[4] Third, the minimization specification for this tag must allow omission. Finally, 'OMITTAG YES' must be specified. The author of a DTD assumes the task of determining which tags are omissible and the minimization is specified in the DTD.[5] Determination of whether a tag is omissible may depend not only on that element's declaration but also on where that element is used in other element declarations.

2.2.1 Complexities in using the OMITTAG feature

OMITTAG specifications can be difficult to use if one does not understand the omission conditions completely. For example, consider the following definition of a list:

```
<!--      ELEMENT MIN  CONTENT           -->
<!ELEMENT list    - -  (item)+             >
<!ELEMENT item    O O  (#PCDATA | list*)   >
```

The string `- -` means that both the start-tag and end-tag are required for the element(s) being defined. The string `- O` means that the start-tag is required and that the end-tag may be omitted, and so on.

The following is a sample, fully marked-up list:

```
<list>
<item>First item</item>
<item>Second item</item>
<item>Last item</item>
</list>
```

To some users, the specification may appear to indicate that all start-tags and all end-tags are omissible on the element 'item', leading to the following possible markup:

```
<list>
First item
Second item
Last item
<\list>
```

A user with some experience may realize that this may not be the correct interpretation of the specification, because there is no way to unambiguously distinguish the items. A newline character has been used here, but newline characters may themselves appear in the contents of an item.

In fact, to correctly decide which 'item' tags are omissible, each decision about omission must be done on a tag-by-tag basis by applying the three conditions for omissibility.

[4] The reader may notice the 'catch-all' nature of this second condition. In fact it was added to later versions of the standard, probably in response to the fact that the SGML-specific rules do allow for ambiguous markup.

[5] It is important to note that a creator of a document instance is not entirely bound to comply with the minimization specified by the creator of the DTD. If the creator of an instance chooses to omit tags other than those specified, the result may be as innocuous as a simple warning from the parser.

Using Rule 1 for tag omissibility, the list can be marked up as:

```
<list>
<item>First item
<item>Second item
<item>Last item</item>
</list>
```

and using Rule 2, this markup can be reduced to:

```
<list>
<item>First item
<item>Second item
<item>Last item
</list>
```

The specification does not indicate that all the item start-tags are omissible. By examining the declaration of list, we see that only the first item is contextually required. The other items are contextually optional. The notation implies that only the start-tag for the first item is omissible. Thus Rule 3 reduces the markup to:

```
<list>
First item
<item>Second item
<item>Last item
</list>
```

Note that this is not the only possibility for minimal markup. Another possible minimal, unambiguous markup is:

```
<list>
First item</item>
Second item</item>
Last item
</list>
```

However, this markup is not allowable because it does not meet the first condition for omissibility, i.e., it is not allowable by the three SGML-specific rules. In particular, it is not permissible to omit the start-tag on the second and third items.[6]

One other problem that may occur with the use of tag omissions is that modifications to existing text may be made incorrectly because insufficient attention is paid to tag rules. For example, if an author wanted to add an item to the beginning of the list above, a common tendency would be to generate the new, illegal list as follows, without concern for putting a begin-tag on the now second item:

[6] M. J. Kaelbling points out in Reference [15]p. 53, that it is not possible to achieve *absolute* minimal markup in SGML because the conditions under which an SGML start-tag is omissible require tag inclusion, even when a parser could unambiguously infer the omitted tag.

```
<list>
New first item
First item
<item>Second item
<item>Last item
</list>
```

As can be seen by this example, determining the legal use of omissible tags can be a complex undertaking, involving knowledge of not only the immediate minimization specification for a given element, but the examination of several declarations in a given DTD, and careful consideration of the current instance being created or modified.

2.2.2 Complexity in specifying omissible tags

We now illustrate the complexity involved in specifying which tags may or may not be omitted. We also illustrate that in order to achieve the most intuitively pleasing omissions, it may be necessary to go back and change the element definitions in an already existing DTD.

We use the following mathematical formula as an example. The DTD element definition that we will be using is taken from Reference [5]. The definition without the specification of minimization is:

```
<!--      ELEMENT       MIN     CONTENT     -->
<!ELEMENT fr            ??      (nu,de)       >
<!ELEMENT (nu,de)       ??      (%f-butxt;)*  >
```

This declaration says that the content of a fraction consists of a numerator followed by a denominator. The content of the numerator and the denominator are defined to be built-up text, which is defined elsewhere. Built-up text *does* include fractions.

Assume you are a specifier of a DTD who now wishes to embellish the element definition with the specification of which tags may be omitted. Your first impulse is likely to be to determine the minimum markup by examining instances of possible declarations of fractions. Rather quickly, you might discover that the string `<fr>numerator<de>denominator</fr>` represents minimal markup (using tag omission only, and not the SHORTTAG feature).

Next, you would attempt to add the minimization specification to the DTD. Upon examination, you would discover that it is not possible to specify this markup exactly, without rewriting the element definitions, because there is no way using the current specification to indicate that only the denominator start-tag is required. So, you might rewrite the element definition, specifying the minimization suggested by the sample string:

```
<!-- ELEMENT MIN CONTENT           -->
<!ELEMENT fr - - (nu,de) >
<!ELEMENT nu 0 0 (%f-butxt;)* >
<!ELEMENT de - 0 (%f-butxt;)* >
```

At this point, it may occur to you that there is another string that represents minimum markup, i.e., `<fr>numerator</nu>denominator</fr>`, and that it is likely that creators of individual instantiations of DTDs may try to use this minimization instead of

the one specified. Such a minimization would, at the very least, cause a warning message to be generated by the parser of the DTD if the specification above were the prevailing one.

So, you might attempt to determine a way to allow for both specifications. In fact, the following specification allows for both forms of minimization, if the creator of the instance applies the conditions for omissibility properly when declaring the DTD instance.

```
<!-- ELEMENT MIN CONTENT       -->
<!ELEMENT fr - - (nu,de) >
<!ELEMENT nu 0 0 (%f-butxt;)* >
<!ELEMENT de 0 0 (%f-butxt;)* >
```

By application of the one condition for omissibility, i.e., the three SGML-specific rules, the minimal markup `<fr>numerator denominator</fr>` is allowable.[7] By application of another condition, i.e., the markup cannot be ambiguous, it is discovered that some tag must be included between the numerator and the denominator. If we choose the numerator end-tag, the desired minimization is achieved.

You may now observe that the initial element declaration was okay, and change the element definitions once again to their original form, with the minimization specification added:

```
<!--      ELEMENT MIN CONTENT     -->
<!ELEMENT fr      - - (nu,de)       >
<!ELEMENT (nu,de) O O (%f-butxt;)*  >
```

This specification for tag minimization is comparable to the list example just discussed, and is unintuitive to the inexperienced creator of document instances. Again, the apparent ambiguity must be resolved in exactly the same way as for the list example, by methodically applying the conditions for omissibility when creating an actual instance of a fraction.

Also, we observe that the most logical action for a specifier of a DTD who wants to allow for maximum flexibility under the SGML minimization rules, is to simply specify all minimization as `O O`. This approach relieves the DTD specifier of any complexity in the specification, but in turn simultaneously places a tremendously complex burden on the creator of a document instance.

In reality, the effective function of the minimization specification is to indicate, among those tags which may be omissible, which tags *must* be included. This is a strange function indeed, because such a specification of required tags is necessarily due to some personal, subjective judgments in the mind of the specifier of the DTD, having nothing to do with possible ambiguity. Such judgments are unlikely to be shared by creators of instantiations of DTDs, who will find these rules arbitrary and unintuitive (as they are), and thus difficult to apply.

This is a rather simple example, involving only two elements. As was observed above, DTDs may involve hundreds of elements that interact in complex ways. Thus, the specification of omit tags can be expected in general to be considerably more complicated than that discussed here, and proportionately more error-prone.

[7] This allowable markup is deduced by applying rule 2 above to the end-tags and rule 3 to the start-tags. It is ambiguous.

Again we might consider the task of using the mechanism of tag omission, in addition to using the SHORTTAG feature, in the actual markup of an instance of a mathematical equation. Consider the following:

$$\frac{1+\frac{1}{1+\frac{1}{x}}}{x}$$

A completely marked-up version of the above formula is:

```
<fr><nu>1+<fr><nu>1</nu><de>1+<fr><nu>1</nu>
<de>x</de></fr></de></fr></nu><de>x</de></fr>
```

Using tag omission only, the markup would be the following:

```
<fr>1+<fr>1</nu>1+<fr>1</nu>x</fr></fr></nu>x</fr>
```

The fraction end-tag is not omissible because a fraction may contain another fraction.

By adding empty end-tag minimization, the markup would be the following:

```
<fr>1+<fr>1</>1+<fr>1</>x</></></></>x</></></>
```

Without a doubt, this markup is considerably shorter than the fully marked-up version. The real question is: is it easier to generate and is it easier to understand? In this example, the start-tags of the numerators and denominators are missing but are understood to be there. The placement of the empty end-tags is similar to matching right parentheses with left parentheses in an expression where the left parentheses (i.e., start-tags) are invisible. Authors of programming languages have long acknowledged the difficult problem of matching nested parentheses, but their case is considerably simpler than this one because they always have the explicit left parentheses. Other, select tags may be entered to better display the intended markup in this case, but that process is itself subject to error because the creator of a document instance must specify only those shortened versions of markup that are allowable both under SGML rules and according to the DTD.[8]

2.2.3 Summary

The tasks of creating a DTD and of creating and viewing instances of a DTD are clearly separate and distinct. Thus, separate and distinct mechanisms should be implemented to support these tasks. The use of the minimization features in the specification of the DTD has the effect of combining the two tasks. This not only unnecessarily complicates the task of defining a DTD, it allows for only an incomplete and awkward mechanism for easing the task of creating and viewing document instances. Thus, we recommend that the minimization features be not used when creating DTDs. We also recommend that vendors continue in their efforts to create friendly, customizable environments that support the creation and viewing of document instances.

[8] The real issue here is that users should not have to deal with this markup form of the mathematical expression at all, but that software support systems should be in place to allow for direct manipulation of the layout form of the expression, with automatic generation of the markup form for exchange.

2.3 Do not use attributes

SGML provides two mechanisms for marking information in a DTD: tags and attributes. Tags are used to mark the begin and end of each element that is defined in a DTD. Attributes are associated with particular elements. Their notation is different from that of tags. They cannot be decomposed hierarchically, and they can be restricted to contain only certain values.

Attributes are provided to serve a function different from that of tags. Loosely speaking, they allow a definer of a DTD to provide meta-information about the elements of a document that are described using the tag mechanism. The relationship between tags and attributes has been variously described as follows: tags mark things, attributes mark information about the things; tags mark objects, attributes mark features of the objects; or tags carry hierarchical information about a document, attributes carry all other information.

A problem arises for a DTD designer when deciding what should be encoded with tags and what should be encoded with attributes. Ultimately the *meaning* of a piece of data is strictly in the eye of the beholder [16]. What is a 'thing' to one may be 'information about the thing' to another. What is an 'object' to one may be 'a feature of an object' to another. So, DTD definers must deal not only with the complexity in defining the DTD elements and their orderings, i.e., the grammar of the DTD. Additionally, they are faced with an equally challenging task of anticipating the views of the document that will be taken by various 'beholders' and trying to accommodate these with the correct set of tags, attributes and even attribute values.

Appealing to our model of separate layers of software to handle separate functions, we recommend that only one type of information should be specified in a standard description of a DTD, to be marked with tags. Any other types of information that are needed to process the DTD, i.e., the information typically described using the attribute mechanism, should be specified and dealt with outside the standard by those groups wishing to impose a particular view on a given document.

Even if one allows for the use of attributes in DTDs, the attribute mechanism is not suited to provide the full functionality that is needed to address the needs of all software applications that may wish to process instances of documents. We elaborate on these points below.

2.3.1 Semantic issues

From a semantic point of view, there is no universal and unambiguous way to distinguish tags from attributes. At best, distinguishing criteria have been suggested that are application-dependent. These criteria are not generally applicable across all domains and even lead to contradictory distinctions within a single domain.

For example, in the publishing domain, the concept of 'hierarchy' has been proposed as a distinguishing one for tags and attributes. All those pieces of information in a manuscript that are clearly hierarchical are to be marked using tags. All other information is to be marked using attributes. Thus, the manuscript components frontmatter, body and rearmatter would be tagged. But nonhierarchical items like references and citations would be marked using attributes.

The problem with the 'hierarchy' criterion is that the concept very quickly blurs as DTDs are defined in other domains. For example, suppose in the linguistics domain it is

desired to define a DTD to mark parts of speech, like verb, noun, adverb and so on. Further, assume that for each part it is desired to mark information like gender, person, number, tense and so on. In this case, neither the parts of speech themselves, nor the information about them, is hierarchical in any clear sense.

Another concept that has been applied in the publishing domain to distinguish tags from attributes is that of 'content'. Content is defined as all those substantive pieces of information that will actually appear on a printed page, i.e., the content of a manuscript. These are to be marked with tags. All else, e.g., formatting or rendering information, are to be marked using attributes. Note that using this criterion, references and citations would now be marked using tags, while using the hierarchy criterion they would be marked using attributes, even though both criteria stem from the same application.

The concept of 'content' can quickly blur, however. For example, for tables, should column separators be considered to be content or formatting? A character will appear as a column separator, be it a vertical bar, space, or some other, on a printed page of a table. An application that processes tables may be written to evaluate visual cues that readers use to distinguish items in a table. In this sense, column separators appear on a page and are considered substantive, and so should be specified using tags.

However, column separators may be thought of simply as formatting conventions. From the point of view of an application that wishes to find the average of all the values in the columns of the table, column separators are extraneous and not substantive. In this case, column separators would be considered not to constitute the content of a manuscript and so should be specified using attributes.

2.3.2 Syntactic issues

From a syntactic point of view, tags are more powerful than attributes in the sense that tags can be further decomposed into subcomponents. On the other hand, certain values, or ranges of allowable values, can be associated with attributes, but not with tags. So, a case may be made for using attributes if a case can be made that it is desirable to specify allowable values of certain strings when a DTD is defined.

In practice, the value of *any* terminal string in the document is strictly a matter of interest or concern only to a particular application that may be processing instantiations of the document. This is equally true for those strings that may be delimited using tags or those that may be delimited using the attribute mechanism. Because it is in the application that the interest originates, it should rightly be in the application that the necessary work to pursue the interest, i.e., specifying values and constraints on values, is done.

For example, consider the 'value' of zip code information. One application may wish to generate mailing lists only for those entries with postal zones indicating residence in Oregon and Washington. Another may wish to verify that all the zip codes are valid based on an official US Postal Service listing of such codes. Or, alternatively, an application may search for all zip codes that are invalid based on such a listing. Still another may wish to analyze the postal codes in order to determine a geographic distribution of entries in the southern portion of the USA. Each of these applications is interested in a constrained 'value' of the string that is used to represent the zip code. However, there is no obvious way to impose constraints on the value of the zip code when the DTD is defined such that all of the interests of these varied applications would be served. Further, it would be a bad

design policy to burden the DTD itself with a set of constraints that are of interest to only one or a few applications.

Or, consider the 'value' of a 'column separator' associated with a table element. Suppose a list of allowable values for the separator has been specified in the DTD. This list is likely to be based on some application for instantiations of the DTD, say a formatter. Now assume that someone is creating an instance of this DTD by marking an already existing manuscript. Suppose that the manuscript has a value for a column separator that does not occur on the list of allowable values. The creator is faced with the choice of illegal markup or no markup at all. On the other hand, if 'column separator' had been marked using tags, the definer would simply fill in the string corresponding to the true value of the column separator.

Even if a case could be made for assigning values in the DTD, the mechanism provided by SGML would be likely to prove inadequate for the need. Attributes are declared in SGML using an attribute-declaration statement that specifies the element name to which the attribute applies, the attribute name, a 'declared value' for the attribute and a default value. The declared value is intended to specify the possible range of values from which the actual value of the attribute may be chosen.

The SGML standard has attempted to anticipate the possible range of values that would be used by definers of DTDs by restricting the 'declared value' parameter of the attribute declaration to a prespecified set of classes. For example, they can have a declared value of NAME which restricts the attribute value to a string in which the first character is a letter and the rest are either letters, digits, a period or a hyphen. Or attributes can have a declared value of NAMES which restricts the value to a list of NAMEs. All of the prespecified classes allow attributes to take on only one type of value, or a list of one type of value.

The AAP in its DTD for mathematical formulas found these classes to be inadequate to fully specify the desired restrictions on some of the attributes they wanted to declare. For example, to describe the value of an attribute *column separators* for an array, they wished to specify a list of ordered pairs as the attribute value. The first element of the pair defines the column number and the second element defines the allowable separator, e.g., single line, double line, blank, and so on. Because the standard does not provide sufficient expressive power for this specification, the AAP specified its desired restriction in a lengthy comment following the formal declaration for the attribute (see Reference [5]p. 49). Clearly, such a description is outside the standard, and no software that has been designed to validate or analyze this DTD will be able to deal with this type of declaration.

2.3.3 Summary

The creator of a DTD is currently faced with specifying two types of information in a DTD: (1) using tags, the set of elements that comprise the DTD, including rules for their order and occurrence, and (2) using attributes, information about the elements that will facilitate certain types of processing of the DTD. Each of these tasks is complex and potentially error-prone, especially as a DTD grows large.

Appealing to the principle of modular separation of functionality, we recommend that these two tasks be handled separately. While creating the DTD, tags should be used exclusively to describe the relevant information in a document. Later, in other layers of

software, information relevant to processing the DTD, e.g., inspection or assignment of values for certain elements, can be encoded.

Even if the creator of a DTD wishes to use attributes to assign or constrain values of certain elements, such constraints cannot be specified in a general way, and can be more effectively handled outside the standard specification, in the application software that actually requires this kind of information.

2.4 Avoid the use of inclusion and exclusion exceptions

Inclusion and exclusion exceptions in SGML are powerful shorthand notations. They indicate that a certain content model should be included or excluded over sets of elements in a DTD, rather than indicating this information on an element-by-element basis. For example, it is possible to use an inclusion exception to indicate that a footnote or figure can occur anywhere in an entire document.

Our experience with these exception mechanisms is that they are so powerful that the DTD definer does not always readily understand the full implications of the exceptions that are specified. For example, in the case of a figure occurring anywhere, this implies that figures can occur in header lines, in footnotes, and in bibliographic citations. Typically, a DTD definer would not intend this. Similarly, for a footnote, indicating that it can occur anywhere in the manuscript would imply that it could occur after a begin-tag for an author string, but before the author string itself. Again, this would be very strange usage in most common documents.

Further, if the definer does understand the implications of an inclusion exception and wants to limit its scope, then an exclusion exception can be used. A problem occurs when inclusion and exclusion exceptions begin to be piled atop one another.

As an example from the AAP DTD for mathematical equations, a content model for 'special characters or strings,' (`%f-scs`), is defined as

```
(%f-scs;) ((%f-cstxt;)|sup|inf)* -(tu|%limits;|%f-bu|%f-ph;)
```

In this case, the definer explicitly includes the sup and inf elements in the content, but then removes them with the exclusion of `%f-ph`, because it is defined as

```
(%f-ph;)  (box|unl|ovl|sup|inf)
```

The actual interpretation of what is intended may become so complex that the only way to understand what is legal in any given context is to provide a full expansion of the shorthand inclusion/exclusion notation. This is what we recommend be done in the first place.

Inclusion exceptions could be used in a restrained fashion to indicate only *floating* elements in a manuscript. Exclusion exceptions could be used in a restrained way only at lower levels of the manuscript hierarchy. Observation of these restraints would render these features of SGML more manageable.

2.5 Use the CONCUR feature

The CONCUR feature of SGML provides a mechanism for more than one simultaneous, hierarchical markup of the same data stream. The main advantage of this mechanism is that

it supports different structural views of the same document. Such views may be required by different analyses.

For example, consider applications that analyze scenes of a play. A play can be viewed as a structure of acts, each made up of scenes. Each scene in turn may be viewed as composed of lines spoken by a character. Alternatively, scenes may be viewed as collections of sentences. These two views may not fit into the same hierarchical structure because sentences may be formed as lines that are delivered by several characters, but lines may include parts of more than one sentence:

```
Character1: I'm sick and tired . . .
Character2: . . . of you completing my . . .
Character1: . . . sentences.  And I'm sick and tired . .
Character2: . . . of you completing my sentences.
```

A similar situation can be found in the markup for the Thesaurus Linguae Graecae [8]. In this case, it is desired to mark up documents indicating both logical and physical elements. The logical elements contain font tags. The physical elements contain page tags. The logical and physical views of the document cannot fit into the same hierarchical structure because pages may contain fonts, but fonts can span more than one page. The same would be true for logical paragraph elements and physical page elements in other styles of markup.

The CONCUR feature supports the definition of simple, straightforward DTDs that are likely to be well-understood by their definers and thus likely to completely and correctly capture their intentions. A potential disadvantage of using the CONCUR feature is that it may be difficult to process a DTD instance with multiple tag sets when the SHORTREF feature has been used [17]. Because we are recommending that this feature should not be used, this disadvantage will not affect the DTD when applying our considerations.

Another consideration that affects the use of the CONCUR feature is that there currently exists little experience in the SGML community regarding the *interactions* of the various hierarchical views of a single document as provided by this feature. These interactions may themselves be quite complex and require further techniques or toolsets to support their correct specification.

3 CONCLUSION

This is an exciting period in the development of electronic documents because the potential for widespread distribution and access of documents is being realized. In the past, distribution and access has been hindered by a proliferation of different data representations in each document type. The arrival of a standardized data representation like SGML can go far to alleviate this hindrance. However, care must be taken in the design of SGML DTDs to ensure that they remain accessible to both humans and machines. This paper suggests a set of considerations for designing DTDs that keeps them uniform and relatively simple, and thus more easily accessible.

We have developed an environment to support creation of DTDs that are constrained by our considerations [13]. We are currently using the environment to create DTDs in the domains of electronic publishing and medical records.

ACKNOWLEDGEMENTS

We would like to acknowledge the contributions to this manuscript of M. J. Kaelbling and C. S. O'Connell.

REFERENCES

1. *Information Processing—Text and Office Systems—Standard Generalized Markup Language (SGML)*, October 1986. ISO 8879-1986 (E).
2. Automated Logistics Systems. Report to the Committees on Appropriations of the United States House of Representatives and the United States Senate: Executive Summary, March 1986.
3. S. A. Mamrak. Patient record exchange project: Specification and software tools. Proposal submitted to the Agency for Health Care Policy and Research, May 1990.
4. Association of American Publishers, *Standard for Electronic Manuscript Preparation and Markup*, 1986. Electronic Manuscript Series.
5. Association of American Publishers, *Markup of Mathematical Formulas*, April 1986. Electronic Manuscript Series.
6. Association of American Publishers, *Markup of Tabular Material*, April 1986. Electronic Manuscript Series.
7. C. M. Sperberg-McQueen. Text encoding initiative: Guidelines for the encoding and interchange of machine-readable texts. Proposal to National Endowment for the Humanities, January 1988.
8. University of California, Irvine, California, *Thesaurus Linguae Graecae: Beta Manual*, 1988.
9. David Mackenzie and Victoria A. Burrus, *A Manual of Manuscript Transcription for the Dictionary of the Old Spanish Language*, The Hispanic Seminary of Medieval Studies, Ltd., Madison, fourth edition, 1986.
10. Stephen A. Kaufman, *The Comprehensive Aramaic Lexicon Text Entry and Format Manual*, The Department of Near Eastern Studies of The Johns Hopkins University, Baltimore, 1987.
11. S. A. Mamrak. A proposal for an integrated chameleon architecture. Proposal submitted to potential industrial members of a consortium, April 1990.
12. S. A. Mamrak, J. Barnes, J. Bushek, and C. K. Nicholas, 'Translation between content-oriented text formatters: Scribe, LaTeX and Troff', Technical Report OSU-CISRC-8/88-TR23, Department of Computer and Information Science, The Ohio State University (August 1988).
13. Conleth S. O'Connell Jr., 'Supporting the development of grammar descriptions for multiple applications', Technical Report OSU-CISRC-7/90-TR20, Department of Computer and Information Science, The Ohio State University (1990).
14. Richard Furuta, 'Complexity in structured documents: User interface issues', in *PROTEXT IV: Proceedings of the Fourth International Conference on Text Processing Systems*, ed., J.J.H. Miller, pp. 7–22. Boole Press (1987).
15. Michael J. Kaelbling, *Braced Languages and a Model of Translation for Context-Free Strings: Theory and Practice*, PhD thesis, The Ohio State University, 1987. Available as University Microfilms International, Inc., No. 8804059.
16. Dennis S. Arnon and Alan J. Perlis, 'Distributed electronic mathematics' (November 1989). Draft.
17. David Barnard, Ron Hayter, Maria Karababa, George Logan, and John McFadden, 'SGML-based markup for literary texts: Two problems and some solutions', *Computers and the Humanities*, **22**, 265–276 (1988).