Article

A new generation of tools for SGML

R. W. Matzen
Oklahoma State University
Department of Computer Science
EMAIL rmatzen@acm.org

Exceptions are used in many standard DTDs, including HTML, because they add expressive power for DTD authors. However, there is a tradeoff: although they are useful, exceptions add significantly to the complexity of DTDs. Authoring DTDs is a difficult task, and existing tools are of limited use because of the lack of a suitable formal model for exceptions. This paper describes methods for constructing a static model that completely and precisely describes DTDs with exceptions. A software tool has been written to implement the methods and to demonstrate some practical applications. Examples are shown of how the tool is used for DTD authoring, and some useful extensions of the tool are described. For one example DTD, the output of the tool is converted into a regular expression grammar. Preliminary studies indicate that general case algorithms can be developed for this conversion. This would allow existing theory for the context free languages to be used in developing SGML applications. Statistical results are shown from running the software tool on a number of industry and government DTDs and for three successive versions of HTML. The results illustrate that the complexity of DTDs in practice is approaching, or has exceeded, manageable limits with existing tools. The formal model and its applications are needed for SGML and continued development of these methods may impact the evolution of HTML, XML, and related web publishing standards. Some specific projects are proposed, where continued development of the model can result in more powerful tools and new kinds of applications for SGML.

Introduction

Exceptions have a dual personality; they are a powerful tool, but they also cause problems. They are used in many DTDs, including HTML, because they are useful for DTD authors: 1. They add to the expressive power of SGML by providing a concise representation for complex content models; 2. They provide a method for controlling recursion introduced by model groups; and 3. They add to the language power of SGML; there are document types defined by DTDs with exceptions that cannot be defined by a DTD without exceptions.

Although exceptions are useful for DTD authors, there is a tradeoff; they add significantly to the complexity of DTDs. As the use of exceptions increases, it

Markup Languages: Theory & Practice 1.1 (1999): 47–74

becomes more difficult to understand the DTD. In practice, DTDs can be so complex that even the DTD author(s) do not fully understand them; they may contain errors and/or be limited in scope. This complexity also implies higher costs for DTD design and subsequent document processing. These problems are primarily due to a lack of a suitable static model for DTDs with exceptions. In ISO 8879 the effect of exceptions is defined on the model groups associated with elements in particular contexts in document instances. This operational definition is easy to implement at run time; SGML parsers maintain a stack of currently applicable exceptions while parsing the document instances. However, this is not a complete static model and therefore is of limited use for reasoning about DTDs.

This paper provides a formal foundation for a new generation of tools for SGML: a static formal model that provides a complete and precise view of DTDs with exceptions. A prototype software tool has been developed to implement the model and to illustrate its potential. The output of the software tool is extended in several ways to provide information for understanding (viewing) DTDs with exceptions and for detecting and correcting errors caused by exceptions. One form of the output is converted into a regular expression grammar for a DTD. Preliminary studies indicate that a general case algorithm can be developed for this conversion. This would allow the existing theory of the context free languages to be applied to SGML. Even without a general case solution the model supports new kinds of applications for SGML.

This paper assumes that the reader has the necessary background in SGML: an understanding of DTDs and exceptions. In the next section, "Definitions", new terms are defined that illustrate the properties of exceptions and that support the methods that follow; these definitions are consistent with the standard [ISO, "SGML"]. The "Methods" section describes a static formal model of DTDs with exceptions and a prototype software tool that constructs this model. Examples in "New tools for DTD design and analysis" show how to use and extend the output of the software tool. "Extending the model" shows an example of converting a DTD with exceptions into two important forms: a pseudo-equivalent DTD without exceptions and an equivalent regular expression grammar. Related work on formal language models for DTDs with exceptions is also described. The "Results" section shows the results from applying the software tool to some industry and government DTDs and to HTML. These results illustrate the complexity of DTDs with exceptions and demonstrate the need for new, more powerful tools for SGML. The results also show that the model and the software tool work for large DTDs currently in use. "Publishing on the World Wide Web" discusses current alternatives for web based publishing, and it describes how the methods shown in the previous sections can be useful in each of these scenarios. The final two sections state the conclusions of this paper, and describe specific

directions for continued work, where the results could significantly reduce the costs of implementing SGML.

Definitions

The definitions in this section illustrate the properties of exceptions and provide a foundation for the remainder of the paper. They are consistent with the definitions in the standard [ISO, "SGML"]. Example 1, a simple DTD and document instance, are used to illustrate the definitions.

Example 1. A DTD with exceptions and a corresponding document instance.

```
<!DOCTYPE  book  [
<!ELEMENT  book     (header,(chapter)+)  +(pagebrk)>
<!ELEMENT  chapter  (header?,(para)+)             >
<!ELEMENT  header   (#PCDATA)   -(pagebrk) +(bold) >
<!ELEMENT  para     (para | #PCDATA)*      +(bold) >
<!ELEMENT  bold     (#PCDATA)              -(bold) >
<!ELEMENT  pagebrk   EMPTY   >
]>
```

A document instance for the DTD (the indentation is for illustration only):

```
<book>
   <header>A <bold>Really</bold> Good Book</header>
   <chapter>
      <header>Chapter 1<header>
      <para>It was a <bold>dark and stormy<bold> night</para>
   </chapter>
</book>
```

Definition 1. Declared exceptions. The declared exceptions for an element A in some document instance are the exceptions declared in the content model of A. The declared exceptions consist of two sets: the declared inclusions and the declared exclusions. Either set (or both) may be empty. The declared exceptions are denoted by +( ) and -( ).

In the DTD in Example 1 the declared inclusions for book are +(pagebrk) and the declared exclusions are -( ), the empty set. The declared exceptions for chapter are +( ) and -( ), both empty sets. The declared exceptions for header are +(bold), -(pagebrk), for para they are +(bold), -( ), and for bold they are +( ), -(bold). Element type pagebrk has declared content; element types with declared content can contain no subelements, and thus can have no declared exceptions, which also is denoted as: +( ), -( ).

Definition 2. Inherited exceptions. The inherited inclusions of an element A occurring in a document instance, are the union of the declared inclusions of all ancestors of this occurrence of A. The inherited exclusions of A are the union of the declared exclusions of all ancestors of A. The notation used for inherited exceptions is the same as that used for declared exceptions.

The inherited exceptions for a particular element depend on the context in which the element occurs. In the following description of inherited exceptions we refer to particular elements in Example 1. The same statements apply to any element of the same type that occurs in the same context (same ancestors) in any document instance. The inherited exceptions for the book element are empty, because it has no ancestors. The inherited exceptions for the header immediately within the book are +(pagebrk), -( ), and the inherited exceptions for the bold within the para within the chapter within the book are +(pagebrk, bold), -( ).

Definition 3. Applicable exceptions. The applicable exceptions of an element A in a document instance are the inherited exceptions of A unioned with the declared exceptions of A:

```
applicable exclusions  =  inherited exclusions  ∪  declared exclusions
applicable inclusions  =  inherited inclusions  ∪  declared inclusions
```

The definition of applicable exceptions does not consider the precedence of exclusions over inclusions. This is given in Definition 4. For the document instance in Example 1, the applicable exceptions for the book element are +(pagebrk), -( ). The applicable exceptions for the book level header are +(pagebrk, bold), -(pagebrk), and for the bold within the para within the chapter within the book are +(pagebrk, bold), -(bold).

Definition 4. Net exceptions. The net exceptions are the exceptions that are active for a particular element in a particular context. They are the same as the applicable exceptions, except that exclusions (either declared or inherited) override inclusions (either declared or inherited). The net exceptions of an element are defined by:

```
net exclusions  =   inherited exclusions  ∪  declared exclusions
                =   applicable exclusions
net inclusions  =   (inherited inclusions ∪ declared inclusions)
                  –  (inherited exclusions ∪ declared exclusions)
                =    applicable inclusions  –  applicable exclusions
```

The notation used for net exceptions is curly braces, +{} and -{}. The net exclusions for all elements are the same as the applicable exclusions. For the DTD and document instance in Example 1, the net inclusions for the book element are unchanged because there is no intersection between the applicable inclusions and

the applicable exclusions. The net inclusions for the book level header are +{bold} and for the bold within the within the book are +{pagebrk}.

SGML parsers are two-stage parsers; they parse an input DTD and then construct a parser for the valid document instances. While parsing the document instance they dynamically compute the net exceptions using a run time stack. The net exceptions for an element may be calculated in two ways: direct from the equations shown above or by :

```
net exclusions  =  net exclusions of parent  ∪  declared exclusions
net inclusions  =  (net inclusions of parent  ∪  declared inclusions)
                   −  net exclusions
```

This second method is more efficient because the intermediate inclusion sets are usually smaller: the net inclusions of an element's parent will always be a subset of the inherited inclusions of the element. Equivalence between these two sets of equations can be shown by simple proofs using the definitions of exceptions and some elementary properties of sets.

For elements with content models, the content of the element in some context in a document is defined by the model group and the net exceptions. Definitions *5* and *6* describe this property of document instances in terms of DTDs.

Definition 5. Dynamic content model. For a DTD, D, a dynamic content model (DCM) for an element type A defined in D, is the model group for A and a set of net exceptions that apply to some occurrence of an A element in some document instance defined by D. For each context in which an element can occur, the DCM (the model group and the net exceptions) completely defines the allowed content of the element. For the purposes of this definition, all element types are assumed to be defined by a content model: for element types with declared content of CDATA or RCDATA the model group is equivalent to (#PCDATA), and the model group for element types with declared content of "EMPTY" is NULL. Because exceptions do not apply to elements with declared content, the net exceptions for all DCMs with declared content are empty. Therefore, element types with declared content have exactly one DCM.

Each element in a DTD has a finite number of DCMs. Let I be the set of all possible sets of inclusions for a DTD and let E be the set of all possible sets of exclusions; then I and E are both the power set of the set of elements defined in the DTD. Therefore they must be finite sets, and (I × E), the possible pairs of inclusions and exclusions, is also a finite set. Then, since each element type defined in the DTD has a finite number of DCMs, the number of DCMs for the DTD is finite. The DCMs for an element type are distinguished from each other by their respective sets of net exceptions. A unique version number (index) is assigned to each DCM to distinguish it from other DCMs of the same element

type (Example 2, Table 1). Definition 6 illustrates how each DCM occurs in specific context with other DCMs.

Example 2. Dynamic content models (DCMs). The DCMs for the following DTD are shown in Table 1.

```
<!DOCTYPE  A  [
<!ELEMENT  A   (B | C)  >
<!ELEMENT  B   (C)        +(X) >
<!ELEMENT  C   (#PCDATA)       >
<!ELEMENT  X   (#PCDATA)  -(X) >
]>
```

Table 1 Dynamic content models for Example 2.

element. version	model group	net inclusions	net exclusions
A.1	(B | C)	{}	{}
B.1	(C)	{X}	{}
C.1	(#PCDATA)	{}	{}
C.2	(#PCDATA)	{X}	{}
X.1	(#PCDATA)	{}	{X}

Definition 6 DCM tree. A DCM tree for a DTD is a tree in which each node represents a DCM of the DTD. The root node is the DCM for the document level (top) occurrence of the DOCTYPE element. The children DCMs (nodes) are derived from the parent as follows: a. The element types of the children are determined by the model group and the net exceptions of the parent DCM. b. Each child's net exceptions are determined by the exceptions inherited from the parent and from the child's declared exceptions.

A DCM tree shows the context in which each DCM can occur in relation to other DCMs of the DTD. The name of each node in the tree is the element name plus a version number for the DCM; each node also has labels (attributes) for the net inclusions and the net exclusions. Note that the version numbers of each node will vary depending on the traversal order. The leaf nodes of a DCM tree are DCMs that have no children. This can occur in one of the following ways:

1. The element type has declared content.
2. The element type has a model group that contains no element names, and there are no net inclusions.
3. All elements in the model group of the element and any included elements are excluded by net exclusions. A DCM tree for the DTD in Example 2 is shown in Figure 1. This tree was constructed in breadth first order. A depth first construction results in exactly the same tree except that the version numbers of the two C nodes are reversed.

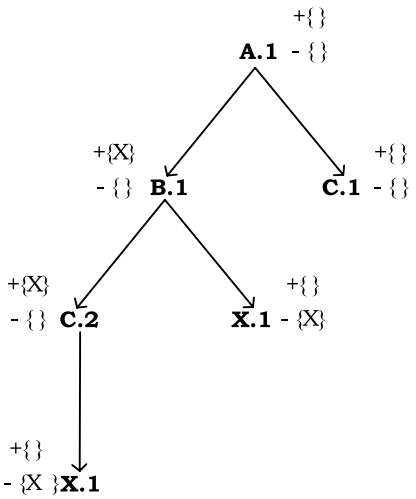

Figure 1 A DCM tree for the DTD in Example 2.

Recursion can be introduced into DTDs by model groups and also by inclusions; at least some paths of DCM trees for recursive DTDs are nonterminating. For the DTD in Example 2, the inclusion of X for B naturally introduces recursion (Xs within Xs), but the exclusion of X for X nullifies it. Because there is no recursion, all paths of the DCM tree for this DTD terminate with leaf nodes that have no children.

Methods

All DCM trees have abbreviated representations in which all paths terminate. This is accomplished by terminating a path whenever a DCM occurs that has already occurred somewhere else in the tree. All paths are guaranteed to terminate because there are a finite number of DCMs (shown in the previous section). Terminating paths by this method implies that there is some order for constructing the abbreviated tree. A depth first construction will result in a different abbreviated tree than a breadth first construction. The version numbers associated with nodes will be different as illustrated in Figure 1; in a depth first construction the version numbers for C.1 and C.2 would be reversed. Also, for abbreviated trees that terminate some paths using the second occurrence rule, the node configurations of the tree will be different. In either case, the core DCMs (elements, model groups, net exceptions) contained in the tree will be the same.

An abbreviated DCM tree for the DTD in Example 1 (constructed in breadth first order) is shown in Figure 2. '*' denotes a DCM that has already occurred in the construction. DC denotes a leaf node that has declared content, and thus can

have no children. Any leaf node that is not marked by DC or * has no children for reasons 2 or 3 in Definition 6.

Abbreviated DCM trees have the following properties:

1. The abbreviated tree shows the correct context for each DCM that appears in the tree. This is direct from the construction, which applies the SGML definitions for determining the content of an element.
2. All DCMs of the DTD will be in the abbreviated tree. This is direct from the definitions of SGML and from the rules for terminating paths.
3. There are a finite number of nodes in the abbreviated DCM tree (all paths terminate).
4. The abbreviated tree completely represents the entire (unabbreviated) DCM tree. A simple proof shows that the subtree of a DCM node must be the same as the subtree of any other node of the same DCM.

A software tool has been developed to construct abbreviated DCM trees and output them in list form. The output for the DTD in Example 1 is shown in Table 2. The algorithm for constructing the tree is straightforward, given the definition of DCM trees and the rules for abbreviating paths. The software tool implements a breadth first construction. This has a distinct advantages over a depth first construction: it results in a tree that is wide rather than deep and in practice this has been shown to be more effective for viewing the results.

Table 2 Output DCM tree (list form) with input DTD from Example 1.

element. version	model group	net exceptions	children
book.1	(header, (chapter)+)	+{pagebrk} -{}	header.1, chapter.1, pagebrk.1
header.1	(#PCDATA)	+{bold} -{pagebrk}	bold.1
chapter.1	(header?, (para)+)	+{pagebrk} -{}	header.1, para.1, pagebrk.1
pagebrk.1	EMPTY	+{} -{}	
bold.1	(#PCDATA)	+{} -{pagebrk, bold}	
para.1	(para | #PCDATA)*	+{pagebrk, bold} -{}	para.1, pagebrk.1, bold.2
bold.2	(#PCDATA)	+{pagebrk} -{bold}	pagebrk.1

The software tool has been modified to annotate the names of the model group elements and the net inclusions with their DCM version numbers as follows. For each DCM encountered in the traversal of the tree there is an element name and an associated version number. The children of each DCM are derived from the model group elements plus the net inclusions minus the net exclusions. Therefore, for each DCM there is a version number associated with each model group element and with each element in the net inclusions (The net

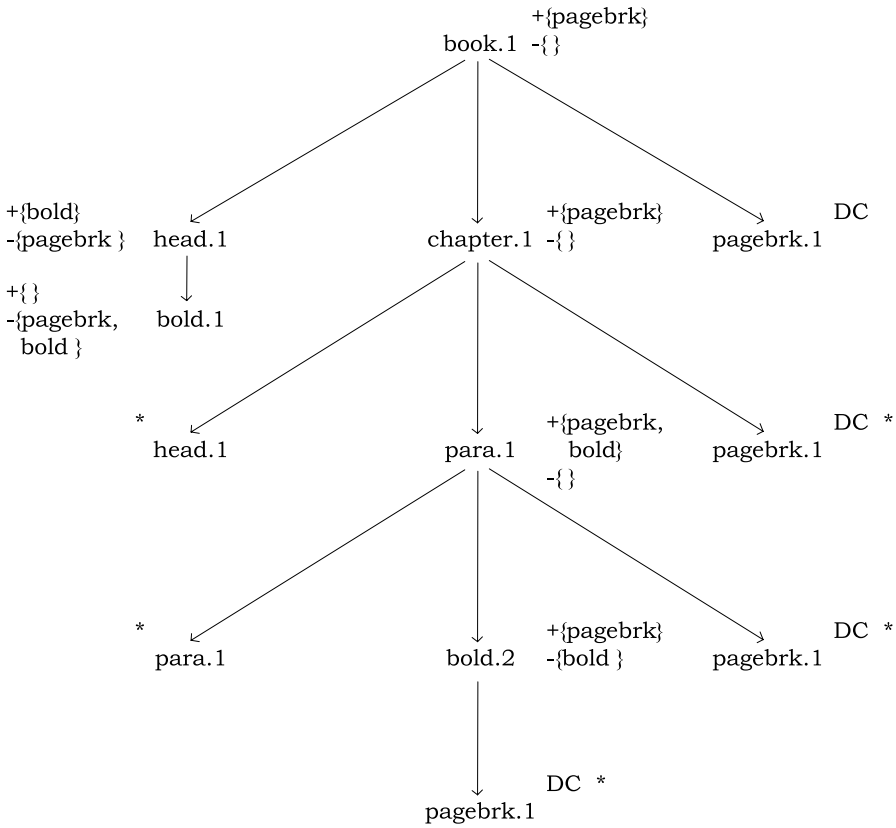

Figure 2 An abbreviated DCM tree for the DTD in Example 1.

exclusions have no associated version number because they are not children of the DCM). This gives a complete description of the DCM tree in a more concise format than in Table 2 or in a graphic representation of the tree. This format is called an expanded DTD and is shown in Table 3. Applications for expanded DTDs are described in "Extending the model".

Table 3 Expanded DTD for the input DTD from Example 1.

element. version	annotated model group	annotated net exceptions
book.1	(header.1, (chapter.1)+)	+{pagebrk.1} -{}
header.1	(#PCDATA)	+{bold.1} -{pagebrk}
chapter.1	(header.1?, (para.1)+)	+{pagebrk.1} -{}
pagebrk.1	EMPTY	
bold.1	(#PCDATA)	+{} -{pagebrk, bold}
para.1	(para.1 | #PCDATA)*	+{pagebrk.1,bold.2} -{}
bold.2	(#PCDATA)	+{pagebrk.1} -{bold}

New tools for DTD design and analysis

DTD design is a complex, expensive, and error prone process. Exceptions are a big part of the complexity problem, but they are used in many DTDs because they add significant expressive power for DTD authors. The software tool described in the previous section constructs abbreviated DCM trees. Some features have been added to the tool to support DTD design and analysis; these and other proposed extensions to the tool are described in this section. The output for all of the examples in this paper was generated by the software tool, except for formatting and where otherwise noted. This is the first tool to provide a complete and precise view of DTDs with exceptions. Other available tools for DTD design typically provide a method of viewing parent/child relationships between elements via a tree or some other graphic method [Pepper, "Whirlwind Guide"]. However, these trees are not abbreviated (finite) and their utility in reasoning about DTDs is limited.

Example 1 is a simple DTD with limited use of exceptions, and the results shown could be derived manually. However, most practical DTDs are much larger, and there are no previously existing algorithms to derive the abbreviated DCM trees. The statistics in the "Results" section illustrate that without effective automated tools it is very difficult to understand the scope of typical DTDs with exceptions or to determine if the use of exceptions has caused errors. In particular, it is difficult to determine if the DTD actually defines the desired document type: Are any elements included in contexts in which they are not intended? Are any elements excluded from, or not included in, contexts in which they are intended? Several extensions to the output of the software tool are shown in this section to help answer these questions.

The DTD in Example 3 illustrates how exceptions can cause errors; we assume that author of this DTD intended to implement the following: Revision elements (rev) are used to mark sections of a document that have been revised. Revisions can occur directly within chapters and/or paragraphs (para); nested revisions (revisions within revisions at any level of nesting) are not allowed. Nested lists and nested paras are also not allowed. Paras are allowed within items within lists, but any other nesting of paras with lists is not allowed. The exclusion of para from para was used to implement this restriction on mutual nesting of paras and lists (unfortunately it has an undesirable side effect). Bold elements are allowed as immediate subelements of either paras or revs nested within paras, but not in any other context, including nested bolds.

The abbreviated DCM tree for the DTD in Example 3 is shown in list form in Table 4. The DCMs for each element type are listed together, rather than in order of occurrence in the traversal; this makes it easier to view all DCMs for one element type. Figure 3 shows a graphic version of the abbreviated DCM tree (for brevity, it does not show the exceptions for each DCM). Both forms of the tree

are useful for viewing the DTD, and for detecting elements included in contexts not intended and detecting elements excluded from contexts in which they are intended. Both of these types of errors occur in Example 3: 1. Bold elements can occur in contexts not intended, as immediate subelements of lists (list.3) and items (item.2). 2. Paras are excluded from a context in which they are intended; a side effect of excluding paras from paras is that some items (item.2) cannot contain any paras.

Example 3. A DTD with exceptions.

```
<!DOCTYPE  book  [
<!ELEMENT  book    (chapter+) >
<!ELEMENT  chapter (rev | para | list)+ >
<!ELEMENT  para    (rev | #PCDATA)*  -(para) +(bold)>
<!ELEMENT  list    (item+)           -(list)>
<!ELEMENT  item    (para*) >
<!ELEMENT  rev     (para | list | #PCDATA)*  -(rev) >
<!ELEMENT  bold    (#PCDATA)         -(bold) >
]>
```

Table 4 Element ordered abbreviated DCM tree (list form) for the DTD in Example 3.

Element. version	model group	net exceptions	children
book.1	(chapter+)	+{} -{}	chapter.1
chapter.1	(rev | para | list)+	+{} -{}	rev.1,para.1,list.1
para.1	(rev | #PCDATA)*	+{bold} -{para}	rev.2, bold.3
para.2	(rev | #PCDATA)*	+{bold} -{rev, para}	bold.1
para.3	(rev | #PCDATA)*	+{bold} -{rev, para, list}	bold.2
para.4	(rev | #PCDATA)*	+{bold} -{para, list}	bold.4, rev.3
list.1	(item+)	+{} -{list}	item.3
list.2	(item+)	+{} -{rev, list}	item.1
list.3	(item+)	+{bold} -{rev, para, list}	bold.2, item.2
item.1	(para*)	+{} -{rev, list}	para.3
item.2	(para*)	+{bold} -{rev, para, list}	bold.2
item.3	(para*)	+{} -{list}	para.4
rev.1	(para | list | #PCDATA)*	+{} -{rev}	para.2, list.2
rev.2	(para | list | #PCDATA)*	+{bold} -{rev, para}	bold.1, list.3
rev.3	(para | list | #PCDATA)*	+{bold} -{rev, para, list}	bold.2
bold.1	(#PCDATA)	+{} -{rev, para, bold}	
bold.2	(#PCDATA)	+{} -{rev, para, list, bold}	
bold.3	(#PCDATA)	+{} -{para, bold}	
bold.4	(#PCDATA)	+{} -{para, list, bold}	

Table 4 provides a useful reordering of the DCM list for the particular problems of determining inclusion/ exclusion correctness. However, the output is still too verbose for some kinds of DTD analysis. In particular, many of the net exclusions have no effect on the model group. For example, all of the net exclusions for all DCMs of the bold element type have no direct effect on the model group; there are no model group elements to exclude. This is an important concept for analyzing DTDs with exceptions, and it is formalized in Definition 7.

Definition 7. The local effective exceptions are the net exceptions that have some effect on the model group of an element in context, but not necessarily on its subelements. Most inclusions are locally effective; even if an element occurs in the model group, including the element will usually extend the context in which it can occur in the containing element. Many exclusions are not locally effective; they have no effect on the model group of the element, even though they may affect the content of subelements within it. A net exclusion is effective if and only if the excluded element is in the model group. Local effective exceptions are denoted by +[ ] and -[ ]. Note that most of the net exceptions shown in Table 4 are not effective; they simply "pass through" elements without effecting their model groups, such as -{para} for all the para DCMs. In some cases such as bold.1–bold.4, the net exceptions also happen to have no affect on any children. Thus, the effective exceptions are a much more accurate measure of how exceptions actually affect the DTD. This property is used in the next section as a metric for DTD complexity.

Table 5 DCMs with local effective exceptions.

Element version	model group	local effective exceptions
para	(rev | #PCDATA)*	
para.1		+[bold] -[]
para.2		+[bold] -[rev]
para.3		+[bold] -[rev]
para.4		+[bold] -[]
list	(item+)	
list.3		+[bold] -[]
item	(para*)	
item.2		+[bold] -[para]
rev	(para | list | #PCDATA)*	
rev.2		+[bold] -[para]
rev.3		+[bold] -[para,list]

Table 5 shows the DCM tree from Table 4 in a reduced form, with only the local effective exceptions listed; the output is now focused on the exceptions that immediately affect the content of the elements, not necessarily their subelements.

The output is reduced further by listing the model groups only once. With these reductions there is much less output to examine to answer the questions about correctness: Are any elements included in contexts not intended? Are any elements excluded from contexts in which they are intended?

The output in Table 5 is useful, but for large DTDs it is still too verbose for some purposes (for the HTML 4.0 DTD there are 2,481 DCMs with effective exceptions). In Table 6, the local effective exceptions of all DCMs for each particular element type are unioned together. This format is the most effective for detecting the two errors in the DTD of Example 3 because it reduces the information to a minimum: the bold elements are included in some contexts not intended (items and lists) and the para elements are excluded from a context in which they are intended (some item). Note that some information is lost in this format, and it is not suitable for certain types of questions. For example, it can not be used to answer the question: is the para element excluded from all intended contexts? This would require the information given in Table 5, or Figure 3. The para element should be excluded from rev.2 and rev.3, but not from rev.1.

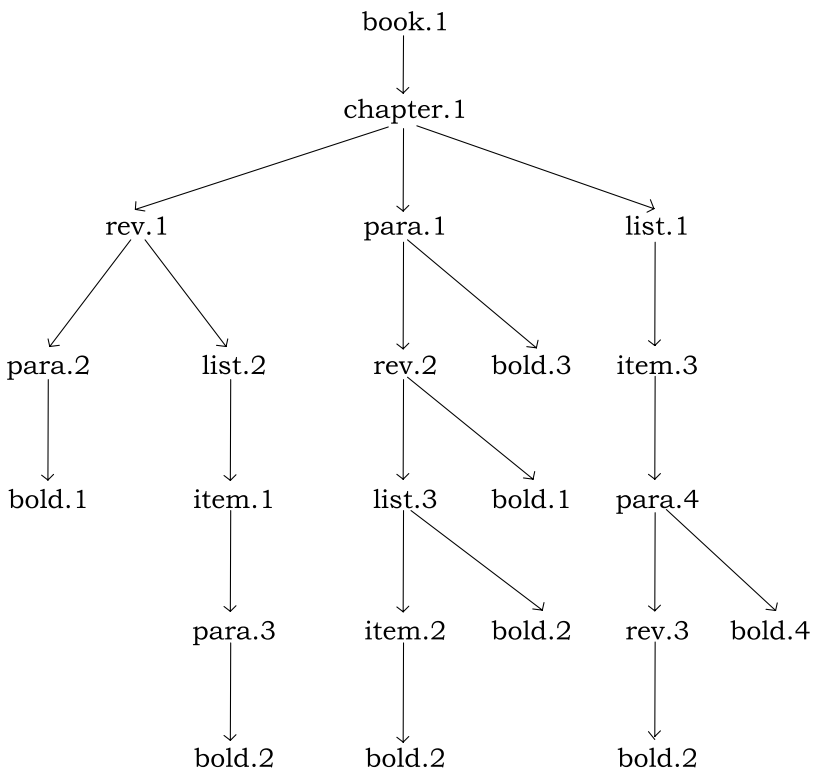

Figure 3 Abbreviated DCM tree for DTD in Example 3.

Table 6 Local effective exceptions: unioned together for each element.

element	model group	all local effective exceptions
para	(rev | #PCDATA)*	+[bold] -[rev]
list	(item+)	+[bold] -[]
item	(para*)	+[bold] -[para]
rev	(para | list | #PCDATA)*	+[bold] -[para,list]

Other useful DTD analysis features can be implemented by extending or modifying the software tool:

1. The abbreviated DCM tree for the DTD in Example 3 has only 23 nodes, but the number of nodes for HTML 4.0 exceeds 80,000 (see Table 9). Clearly, the batch format output of the software tool is not suitable for large DCM trees; interactive access to the results is important to continued development. Traversing a graphic version of the abbreviated DCM tree would be very useful for general purpose DTD analysis with some limitations imposed by the size of typical abbreviated trees; the user should be able to focus the traversals based on particular exceptions, particular element types, and other criteria. Also, it would be useful to annotate the nodes with the declared and effective exceptions, in addition to the net exceptions, and allow the user to view any of these on request.
2. Determine if infinite recursion exists in the DTD and show where it occurs; infinite recursion occurs at each leaf node that is a repeated DCM and that also has children. Some DTDs are designed to allow infinite recursion even though this generally causes confusion for applications that process SGML documents. This formal specification for the recursion in DTDs provides a foundation for resolving this confusion.
3. For each element type, construct parent/child lists: complete listings of all parents and all children of the element in all its possible contexts. This is typically done manually for DTD analysis and documentation purposes; for large DTDs with exceptions it is difficult and is likely to result in errors. A useful extension of this feature would be to output complete ancestor and descendant lists, which is not feasible to do manually.
4. Detect DTDs that allow exclusion errors, rather than detecting actual occurrences of these errors while parsing document instances. This is an important feature that can eliminate serious problems, since detecting exclusion errors while parsing documents is analogous to detecting bugs in a software program after it has been released for use.
5. Detect inaccessible element types: those elements that cannot occur in any document, because they do not occur in a model group, or because they have been excluded in all possible contexts. These are currently detected by the software tool; if there is no DCM for the element type, then it is inaccessible.

Extending the model

Expanded DTDs are a compact list form of abbreviated DCM trees (Table 3). In an expanded DTD each DCM has its own modified element declaration; the element type for each declaration is the element name plus the version number. In Table 7, the expanded DTD for Example 3 is shown in a slightly modified form that is output by the software tool; the exceptions for each modified element declaration are the local effective exceptions of the DCM, rather than the net exceptions. The model group elements and the inclusions for each modified element declaration are annotated with their respective version numbers (directly from the DCM tree). The model group elements marked with an 'X' for a version are those that have an effective exclusion and cannot be children. The first subsection of this section shows how expanded DTDs can be used to convert DTDs with exceptions into pseudo-equivalent DTDs without exceptions, and the second shows how to derive a context free specification for a DTD with exceptions.

Table 7 Expanded DTD for Example 3.

element. version	annotated model group	local effective exceptions
book.1	(chapter.1+)	
chapter.1	(rev.1 | para.1 | list.1)+	
rev.1	(para.2 | list.2 | #PCDATA)*	
para.1	(rev.2 | #PCDATA)*	+[bold.3]
list.1	(item.3+)	
para.2	(rev.X | #PCDATA)*	-[rev] +[bold.1]
list.2	(item.1+)	
bold.1	(#PCDATA)	
item.1	(para.3*)	
para.3	(rev.X | #PCDATA)*	-[rev] +[bold.2]
bold.2	(#PCDATA)	
rev.2	(para.X | list.3 | #PCDATA)*	-[para] +[bold.1]
bold.3	(#PCDATA)	
list.3	(item.2+)	+[bold.2]
item.2	(para.X*)	-[para] +[bold.2]
item.3	(para.4*)	
para.4	(rev.3 | #PCDATA)*	+[bold.4]
rev.3	(para.X | list.X | #PCDATA)*	-[para,list] +[bold.2]
bold.4	(#PCDATA)	

Removing exceptions from DTDs

For the general case, it is not possible to convert a DTD with exceptions into an exactly equivalent DTD without exceptions. However, it is possible to convert

DTDs with exceptions into useful, pseudo-equivalent DTDs without exceptions. In a pseudo-equivalent DTD, the different DCMs for an element type each have their own element declaration as shown in Table 8. The most difficult part of this conversion is already accomplished by the software tool (Table 7); methods for the remainder of the conversion are described below.

In expanded DTDs the exceptions affect only the model group of the element types (DCMs) to which they apply. Thus, the exceptions can be merged into (applied to) the model groups. The result is an expanded DTD without exceptions, that is pseudo-equivalent to the original DTD. For each modified element declaration, perform the following steps:

1. Modify the model group to reflect the effect of any exclusions.
2. Determine if there are any required elements in the model group for which there are corresponding effective exclusions; these are precisely the situations where exclusion errors can occur in document instances of the original (non-expanded) DTD. For the normal case when there are no exclusion errors proceed with Steps 3–4.
3. Modify the model group to reflect the effect of any inclusions.
4. Simplify the model groups (if necessary) so that they are not ambiguous as defined and prohibited by the standard [ISO, "SGML"]. This definition of ambiguity is generally equivalent to nondeterminism in regular expressions [Brüggemann-Klein and Wood, "Validation"].

For many cases Steps 1–4 can be performed with little effort and with no special handling. For example, the DTD in Table 8 was derived from Table 7 by heuristically applying Steps 1–4. It is equivalent to the DTD in Table 7, and it is pseudo-equivalent to the original DTD in Example 3. The heuristic approach is useful because it allows the user to view the original DTD at a level of detail not previously available and provides opportunities to correct errors and delete unnecessary declarations. For example, the logical errors in the DTD from Example 3 can be corrected by changing the model group for the list.3 production in Table 8 to (item.2)+, and the model group for item.2 to (para.2*). These changes prevent bolds from occurring in contexts not intended (lists and items) and allow paras where needed (item.2).

These error corrections can alternatively be accomplished by simply changing the list.3 model group to (item.1)+ and then removing the useless declaration for item.2. In general, the expanded DTDs are not optimized; they may contain unnecessary element declarations. For example, in Table 8 the productions for the bold DCMs all have the same model group, (#PCDATA). If all occurrences of the bold.2–bold.4 DCM names in model groups are replaced with bold.1, then the

Table 8 An equivalent expanded DTD without exceptions.

element. version	annotated model group (with exceptions applied)
book.1	(chapter.1+)
chapter.1	(rev.1 | para.1 | list.1)+
rev.1	(para.2 | list.2 | #PCDATA)*
para.1	(rev.2 | #PCDATA | bold.3)*
list.1	(item.3+)
para.2	(#PCDATA | bold.1)*
list.2	(item.1+)
bold.1	(#PCDATA)
item.1	(para.3*)
para.3	(#PCDATA | bold.2)*
bold.2	(#PCDATA)
rev.2	(bold.1 | list.3 | #PCDATA)*
bold.3	(#PCDATA)
list.3	(bold.2*, (item.2, bold.2*)+)
item.2	(bold.2*)
item.3	(para.4*)
para.4	(rev.3 | #PCDATA | bold.4)*
rev.3	(#PCDATA | bold.2)*
bold.4	(#PCDATA)

declarations for bold.2–bold.4 can be removed. Future work should include developing an efficient algorithm to minimize the number of element declarations in an expanded DTD.

The pseudo-equivalent DTDs are a useful form. As described above, they allow authors to design DTDs using the expressive power of exceptions while managing their undesirable side-effects. They can also be used to replace an existing DTD; tools can be developed that map the base element names in the document instances to their appropriate DCM versions. This process could support conversion to XML, an SGML based processing standard for the web that does not allow exceptions (see "Publishing on the World Wide Web"). [Maler, "Exceptions"] describes some heuristic techniques for removing exceptions from DTDs. The expanded DTDs (Table 7) can be a useful tool in this approach.

There is previous work on applying exceptions to model groups (Steps 1 and 3 above) and detecting exclusion errors (Step 2). The results are mostly theoretical and omit some details ([Matzen, "Model"]; [Kilpeläinen and Wood, "SGML and Exceptions"]). Previous work, special cases, and remaining open problems are described below.

Step 1: Applying effective exclusions to model groups cannot be accomplished by simply removing the elements and their associated operators. For the modified element declaration:

```
<!ELEMENT  A.1   (B | (C, D))   -[C]  >
```

this approach would result in the model group (B | D), which is incorrect because a single D element is not valid content for A.1. Because C is a required element in the subgroup (C, D), the entire subgroup must be excluded. In [Kilpeläinen and Wood, "SGML and Exceptions"] a direct approach is given for applying exclusions to model groups: convert the model group to a regular expression, then replace all occurrences of the excluded element in the expression with ∅, the empty set. The resulting expression defines the model group as affected by the exclusions. Although no implementation details are given, this approach appears to be correct if algorithms are implemented to reduce the resulting expressions using basic properties of regular expressions given in (Aho and Ullman, 1972). Another method for applying exclusions to model groups is described in [Matzen, "Model"]: the model group is converted to a finite state automata (FSA), then all transitions on excluded elements are removed from the FSA.

Regardless of which method is used a problem can occur. Consider the modified element declaration:

```
<!ELEMENT   A.1  (B?)  -[B]  >
```

The result of applying the exclusion is `(B?) = (B | ε) = (∅ | ε) = (ε)`, the empty string, which is not a valid model group, and thus cannot be used in an element declaration. However, no exclusion error occurs because B is an optional element.

Empty content is allowed as the result of applying an exclusion at run time, even though it is not allowed as a model group (empty content in element declarations is expressed as declared content of EMPTY). Changing the content model of A.1 to declared content of EMPTY is not a good solution, because the old document instances will contain end tags, and elements declared as EMPTY cannot have end tags. [Kilpeläinen and Wood, "SGML and Exceptions"] propose interpreting/modifying the standard so that a result of (ε) is an exclusion error. Although this may be a reasonable interpretation, it implies compatibility problems with existing SGML parsers. There is an ad hoc method of handling this case that does not require this modification to the data or changing the standard: define a dummy (unused) element type, D, and use the model group, (Dε) instead of () in the element declaration of A.1.

Step 2: There are two considerations for exclusion errors: how to detect them and what to do when they are detected. The standard gives guidelines for identifying required elements, but it does not require SGML parsers to detect and report exclusion errors (attempts to exclude required elements). Some parsers report them at run time while parsing document instances, but it has not been shown that their implementations are complete and correct. The algorithms described above for applying exclusions can directly be used to detect exclusion errors; they occur anytime that the model group defines no content, ∅, after the exclusions are applied. For example, in the modified declaration

```
<!ELEMENT   A    (B, C)   -[B]  >
```

the model group defines the content of an A as a B followed by C. If all content containing a B is excluded, the resulting model group defines no content, (B, C) = (∅, C) = (∅). Thus, an exclusion error will occur in a document instance. There is an important distinction between no content, ∅, and the set containing only the empty string, {ε}. Note that the original model group does not allow the empty string, and therefore will still not allow it when all strings containing B are excluded.

When exclusion errors are detected, the standard specifically states that there is no defined action: the document instances may be processed by ignoring the exclusion, they may be rejected as invalid documents, or some other action may be taken. One primary reason that the standard allows DTDs that cause exclusion errors is that there previously has been no method for detecting them in the DTD. However, given the results here, the most reasonable way to handle these errors is to detect and reject DTDs that allow them to occur rather than detecting them while parsing document instances, which is analogous to detecting program errors when they occur on certain input values. Also, the lack of defined actions complicates run time detection by encouraging nonstandard implementations.

For the cases where a DTD cannot be rejected, Step 1 can be modified so that the resulting expanded DTD defines the correct document type for the two cases: either accepting or rejecting the document instances that cause the exclusion errors. To accept them simply do not apply the exclusion that causes the error. To reject them remove the offending modified element declaration, then remove any references to it. For example, if the element declaration for some DCM A.1 has an exclusion error, remove the declaration for A.1, then apply an exclusion of A.1 to all element declarations with A.1 in the model group. If applying these exclusions causes further exclusion errors, then continue this process recursively until done. The resulting DTD defines all document instances defined by the original DTD, except those that contain the original excluded element(s) in the contexts that cause exclusion errors.

Step 3: A method for applying inclusions to model groups is outlined in Section 11.2.5.1 of the standard: for a model group M, for all applicable inclusions, (R1, R2, ...RN), for M, for all QX, where Q is either a parenthesized subgroup of M or a token A in M (an element or #PCDATA), and for all occurrence indicators, X (*, ?, +, or null), replace QX with the expression,

```
(R1 | R2 | ... | RN)*,  (Q, (R1 | R2 | ... | RN)* )X
```

This approach redundantly applies the inclusions to the parenthesized subgroups of M; they need only be applied to the tokens. [Kilpeläinen and Wood, "SGML and Exceptions"] show a modified approach that eliminates this redundancy. For each token A in the model group, replace A with:

```
(R1 | R2, ... | RN)*,  A,  (R1 | R2 | ... RN)*
```

Both versions omit an outermost set of parenthesis that can result in a model group containing more than one kind of connector (|, &, or comma), which is a syntax error. The second version omits the parenthesis binding an occurrence indicator X of A to its new operand "`A, (R1 | R2 | ... | RN)`". This can cause problems, as illustrated by the modified element declaration:

```
<!ELEMENT  A   (B+)    +[C]  >
```

the result of applying the inclusion of C using the second version is:

```
<!ELEMENT  A   (  (C)*, B, (C)*+  )    >
```

which will not parse because of adjacent occurrence indicators. Resolving this by ordinary operator precedence results in

```
<!ELEMENT  A   (  (C)*, B, ((C)*)+  )    >
```

which allows C to occur anywhere as intended, but only allows one occurrence of B, which is an error. Adding the parenthesis resolves this problem: for each element A and occurrence indicator X, replace AX with:

```
( (R1 | R2 | ... | RN)*,  (A, (R1 | R2 | ... | RN)* )X )
```

Do the same for A = #PCDATA, but add an explicit * occurrence indicator for X when X is null. Although proof of correctness is not given, this appears to be a solution except for the special case described below. Comments and observations

are invited. [Matzen, "Model"] gives an algorithm for applying inclusions to FSAs constructed from model groups.

[Kilpeläinen and Wood, "SGML and Exceptions"] observe a special case that must be handled, regardless of which method is used: the standard specifies that if an element in a document instance can match both a model group element and an inclusion, it is matched to the model group element. For the declaration:

```
<!ELEMENT  top   (a | b)  +[a] >
```

the revised method for applying inclusions results in:

```
<!ELEMENT  top   ( (a*, (a, a*)) | (a*, (b, a*)) )    >
```

which is incorrect: any string $a^i b a^k$, for $i > 0$, is not valid content for top, because by this special rule, the first a in the string matches the *a* from the original model group. This eliminates any possible string match with the *b*. The revised method described above for applying inclusions can be modified to handle these cases. Index the symbols in the model group: ((a_1*, (a_2, a_3*)) | (a_4*, (b_5, a_6*))). Then using the methods described in [Brüggemann-Klein and Wood, "Validation"], determine if any symbol in any input string can match two indexed symbols, a_i and a_j, where one is from the original model group and the other is an included element. For each such case, remove the a_i* for the included element from the expression. For the above example, the first *a* in a string can match the model group element a_2 as well as two included elements a_1 and a_4. Removing the a_1* and a_4* results in ((a_2, a_3*) | (b_5, a_6*)) = ((a, a*) | (b, a*)) which is correct.

Step 4: Maintaining unambiguity after inclusions are applied is a more difficult problem; more is required than is shown in Step 3 above. [Kilpeläinen and Wood, "SGML and Exceptions"] state that methods exist that preserve unambiguity of the original content model, but details of implementation are not given for the general case. Heuristic approaches can be useful, as illustrated by Table 8.

Context-free specifications for DTDs with exceptions

The methods described above can be modified to obtain an important general result. For each DTD with exceptions there is an equivalent regular expression grammar; regular expression grammars are equivalent to the context free grammars [Woods, "Augmented transition networks"]. A regular expression grammar is a set of productions: the left side of each production is a nonterminal symbol, the right side is a regular expression over the terminal symbols (tokens) and the nonterminals, and one nonterminal is the start symbol.

A DTD with exceptions can be converted to a regular expression grammar as follows. First construct the expanded DTD with effective exceptions (Table 7). Then perform Steps 1–3 above to derive the expanded DTD without exceptions

(Table 8). Maintaining unambiguity (Step 4) may be desirable, but is not required for this result. Also, the requirements for Steps 2 and 3 are relaxed; they need only to derive a regular expression, not a valid model group. After performing Steps 1–3, modify the resulting regular expressions as follows: prepend a begin tag and append an end tag, but use the original element names in the tags, rather than the annotated element names (DCMs). Thus, the modified element declaration

```
<!ELEMENT  para.1   (rev.2 | #PCDATA | bold.3)*  >
```

becomes the production

```
para.1 → <para>, (rev.2 | #PCDATA | bold.3)*, </para>
```

For clarity, the SGML operators are used here rather than their equivalent regular expression operators. The DCM names are the nonterminals, and the begin tags, end tags, and data characters in #PCDATA are the terminal symbols. The para elements are defined by all of the productions for para DCMs, each providing the correct content definition for para elements in the particular context. The result of applying this step to each modified element declaration is a regular expression grammar that is equivalent to the original DTD with exceptions (Example 3).

There is previous work on context free models of DTDs with exceptions, but the primary focus has been theoretical. In [Matzen, "Model"] methods are outlined for constructing systems of finite automata from DTDs with exceptions and the OMITTAG feature. Systems of finite automata are a context free class of recognizers that are equivalent to regular expression grammars [Woods, "Augmented transition networks"] In [Kilpeläinen and Wood, "SGML and Exceptions"] an algorithm is outlined for converting DTDs with exceptions into structurally equivalent extended context free grammars.

Results

The DTD in Example 3 is a simple DTD; the errors in it could be detected easily by anyone competent in DTD design. However, in practice, DTDs are much larger and more complex. The results shown in this section illustrate the complexity of DTDs currently in use, and they show the need for new tools to assist in DTD design and other SGML applications. The software tool to implement Algorithm 1 was run on seven DTDs:

1. The DTD from Example 3.

2. HTML 2.0. The first version of HTML to be formally defined by an SGML DTD [Connolly, "HTML"].
3. HTML 3.2. The last W3 recommendation for HTML ([Raggett, "HTML 3.2"].
4. Both HTML 2.0 and 3.2 have optional elements. The choices made for this paper did not affect the results in Table 9 significantly (± 5%).
5. HTML 4.0. An early version of HTML 4.0 ([Raggett, "HTML 4.0 (WD)"]. Results for the current W3 recommendation, "frameset" version of HTML 4.0 [Raggett et al., "HTML 4.0"] will be available on the Markup Languages Web site.
6. CALS 38784C. The baseline DTD for the Department of Defense Continuous Acquisition and Lifecycle Support initiative [Department of Defense, "MIL-M-38784C"].
7. J2008. The automobile and truck industry DTD for emission related automotive service information [SAE, "J2008"].
8. RIF-EPC. Railroad Industry Forum Electronics Parts Catalogue [Railroad Industry Forum, "Electronic Parts Catalog DTD"].

No attempts were made to determine the correctness of the above DTDs. This is best answered by the respective DTD authors; it requires understanding the semantic specifications for each DTD, as well as the authors' intents and limitations. Instead, statistics were compiled for each DTD; the results are shown in Table 9. The first three rows reflect the size of the DTD and the use of exceptions. The values in rows 4–6 provide a rough measure of the overall complexity of the DTD and the effects of exceptions on it, and they are proportional to the complexity of the DTD. Rows 4 and 5 are rough measures and in some cases grossly overstate complexity. Row 6 is the most accurate of these values for measuring DTD complexity caused by exceptions.

Table 9 Statistics/Results for DTDs.

	Example 3	HTML 2.0	HTML 3.2	HTML 4.0	CALS	RIF-EPC	J2008
Number of element types in the DTD	7	46	67	89	146	99	130
Element types with declared exceptions	4	8	7	10	18	17	27
Total number of declared exceptions	5	35	44	75	33	39	165
Number of DCMs in the DTD	19	148	282	2887	1037	495	729
Nodes in the abbreviated DCM tree	23	1759	5902	82,236	9909	1102	4242
DCMs with local effective exceptions	8	86	130	2481	1020	76	493

The results in Table 9 clearly show that DTDs currently in use are complex. Implementing SGML applications is expensive, and the cost is proportional to the complexity of the DTD. Future work should include a study of refining the methods shown in Table 9 to provide more accurate metrics for DTD complexity. This would be useful for minimizing the impact of exceptions on DTD complexity and also for estimating costs for SGML applications.

Publishing on the World Wide Web

The growth in the HTML DTD shown in Table 9 has been necessary to support the increased demand for processing by the various parties publishing on the web: more sophisticated formatting (style/presentation), interactive forms, applets, etc. Because these new features require increased representation of structure in the HTML DTD, its complexity is approaching manageable limits with existing tools. There are currently two distinct paths in the evolution of web based publishing:

1. Continue expanding the HTML DTD in response to demands for new processing ([Press, "Not TV"]; [Raggett, "HTML 4.0 WD"]; [Raggett et al., "HTML 4.0"]). Given the rapid growth, this scenario requires the development of new tools for understanding and processing HTML. Without these tools the HTML DTD will become unmanageable in the near future. The methods in this paper can help to extend the lifetime of this single DTD approach.
2. There have been proposals to adopt generalized SGML as a publishing standard for the web ([Press, "Not TV"]; [Sperberg-McQueen and Goldstein, "HTML to the Max"]). These have evolved into the Extensible Markup Language (XML), a new standard for web publishing that is a subset of SGML [Cover, "SGML/XML Web Page"]; it maintains the meta-language capabilities of arbitrary DTDs, but it eliminates many features of SGML that contribute to its complexity. It eliminates exceptions because of the problems with understanding their effects on DTDs and related complexity in processing. The methods in this paper provide an alternate solution to this problem; they are a strong argument for keeping the expressive power of exceptions in XML, and this in turn may contribute to its success. In the event that exceptions are not added to XML the model and tool are still applicable to XML, for modeling DTDs with recursion.

   The methods described for converting DTDs with exceptions into pseudo-equivalent DTDs without exceptions can be used for SGML DTD to XML DTD conversions. However, it is not clear that these conversions will be widely used, because a processing standard called Extensible Style Language

(XSL) [Cover, "SGML/XML Web Page"] is being developed that does not need DTDs to publish (view) the document instances. In XSL, processing instructions are defined for elements in particular contexts, and these can be identified when parsing the document instances. However, there are currently no tools to provide a finite representation of all possible elements in context, particularly for DTDs with exceptions; this makes it difficult to determine if a set of processing specifications is complete and correct. The construction of abbreviated DCM trees can be modified to provide this finite representation, and this will be useful for developing complete XSL specifications when publishing SGML data in an XML/XSL environment.

Conclusions

The results in Table 9 illustrate the complexity of DTDs with exceptions, which in turn implies high costs for DTD design and corresponding problems with quality. These results also show that the complexity of some DTDs is approaching (or has exceeded) manageable limits given existing tools for designing and understanding them. There is clearly a need for more powerful tools for DTD design and analysis and for subsequent SGML processing. The software tool described in this paper is useful for understanding (viewing) DTDs with exceptions and for detecting errors caused by the incorrect use of exceptions. Several practical extensions of the tool are described that provide other new capabilities for DTD analysis. Because exceptions are an integral part of SGML, any generalized SGML tool must support them. There are previous theoretical results for formal language models of DTDs with exceptions ([Matzen, "Model"]; [Kilpeläinen and Wood, "SGML and Exceptions"]). However, this is the first description of an implementation, and thus it provides a foundation for a new generation of applications and tools. Some of these are discussed in the section on "Future work".

SGML is used to define the syntax (structure) of documents. It does not directly address the semantics (processing), but it does provide a structural foundation for attaching processing specifications. Standards for processing SGML have not been widely accepted because they are complex and difficult to implement; one of their primary limitations is the lack of a complete static view of SGML structure. The model in this paper provides this view; it will be useful in implementing existing processing standards and important for developing new, more robust approaches to SGML processing.

The expanded DTDs output by the software tool are a powerful extension of the model; these can be used to construct DTDs without exceptions that are pseudo-equivalent to the original DTDs with exceptions. This allows authors to

design DTDs using the expressive power of exceptions while managing their side-effects. Also, the methods shown for converting DTDs with exceptions to regular expression grammars provides a powerful formal foundation, the existing theory for the context free languages, to be used in developing new kinds of SGML applications. The continued development of the methods and tools described in this paper can be a significant factor in the future success of SGML, and they could affect the evolution of HTML, XML, and other standards for the World Wide Web.

Future work

There are specific areas where continued work could result in new tools for existing applications such as DTD design, and also for new kinds of applications. Developing these tools could significantly reduce the costs of implementing SGML.

1. Develop a complete DTD design and analysis tool based on the methods and the software tool described in this paper. Include interactive focused traversals of abbreviated DCM trees, query based features such as examining recursion, and automatic features such as detection of DTDs that allow exclusion errors. Also, add other features based on an extended study of the practical problems encountered in DTD design and analysis.
2. Extend the model to provide a finite representation of all possible contexts for elements, which are the basic units that may have different processing requirements. This can be accomplished by minor modifications to abbreviated DCM trees. This extended model will be generally useful for formally communicating processing specifications for DTDs, and it will be directly applicable to developing XSL processing instructions for XML. Also, this extended model will be useful for studying comprehensive new approaches to SGML processing.
3. Refine the methods presented in "Results", to develop more precise metrics for the structural complexity of DTDs. This will be useful in determining feasibility and for estimating costs for SGML projects.
4. Given the algorithms shown here for constructing context free specifications for DTDs with exceptions, it may be possible to develop a solution to the DTD subset problem: are the documents defined by one DTD a subset of those defined by another. Then, version compatibility could be automatically confirmed. For example, are all HTML 3.2 documents valid under HTML 4.0. Even though the subset problem is not solvable for the general case of the context free languages, begin and end tags may make this a solvable problem for SGML [Sperberg-McQueen, "Complexity"].

Received 22 June 1998
Revised 31 July 1998

Acknowledgments

I would like to acknowledge the contributions to this work by Dr. G. E. Hedrick and Dr. K. M. George of Oklahoma State University, and I wish to express my deep appreciation for their steadfast support.

References

Brüggemann-Klein, A., and D. Wood. "The Validation of SGML Content Models". *Mathematical and Computer Modeling* 25:4 (1997): 73–84.

Connolly, D., and T. Berners-Lee, eds. "Hypertext Markup Language — 2.0". RFC 1866. [Cambridge, Mass.]: MIT/W3C, September 22, 1995. The DTD itself (dated June 6, 1995) is at http://www.w3.org/MarkUp/html-spec/html-s.dtd.

Cover, R., SGML/XML Web Page, http://www.sil.org/sgml/, Dec. 1997.

Hopcroft, J. E., and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Reading: Addison Wesley, 1979, pp. 29–35.

ISO (International Organization for Standardization). *International Standard ISO 8879 Information Processing — Text and office systems — Standard Generalized Markup Language (SGML)*. [Geneva]: ISO, 1986.

Kilpeläinen, P., and D. Wood. "SGML and Exceptions". In D. Wood and C. Nicholas, eds. *PODP 96: Proceedings of the Third International Workshop on Principles of Document Processing* (Palo Alto, California, Sept. 1996) pp. 39–48. Springer-Verlag, Berlin, October, 1997, Lecture Notes in Computer Science, Volume 1293.

Maler, E. "SGML Exceptions and XML". ArborText white paper, 1998. http://www.arbortext.com/sgmlexpt.html.

Matzen, R. W. "A Formal Language Model for Detecting Ambiguity in SGML". Diss. Oklahoma State University, 1993.

Matzen R. W. "Unraveling Exceptions". In *Conference Proceedings: SGML/XML 97*. Washington, D.C.: Graphics Communication Association, December, 1997, pp. 289–295.

Matzen, R. W., K. M. George, and G. E. Hedrick. "A Model for Studying Ambiguity in SGML Element Declarations". In *Proceedings of the 1993 ACM / SIGAPP Symposium on Applied Computing* (February 14–16, Indianapolis, Indiana). New York: ACM, 1993, pp. 668–676.

Pepper, S. "The Whirlwind Guide to SGML Tools and Vendors". Oslo: Falch, 1997. http://www.falch.no/people/pepper/sgmltool.

Press, L. "The Internet is Not TV: Web Publishing". *Communications of the ACM* 38.3 (March 1995): 17–23.

Raggett, D., HTML 3.2. http://www.w3.org/MarkUp/Wilbur/html32.dtd, January 14, 1997.

Raggett, D., HTML 4.0. http://www.w3.org/TR/WD-html40/sgml/html4.dtd, July 8, 1997.

Raggett, D., Le Hors, A., and Jacobs, I. HTML 4.0 http://www.w3.org/TR/REC-html40/frameset.dtd, April 24, 1998.

Railroad Industry Forum, Railroad Industry Forum — Electronic Parts Catalogue DTD, 1996, http://www.eccnet.com/rif/rif-epc.dtd, February 13, 1996

SAE (Society of Automotive Engineers). "J2008 DTD for Interactive technical Manuals". 1993. http://www.sil.org/sgml/gov-apps.html#j2008.

Sperberg-McQueen, C. M., and R. F. Goldstein. "HTML to the max: A manifesto for adding SGML intelligence to the world wide web". In *Proceedings of the Second Web Conference* (Chicago, Oct. 1994). http://www.ncsa.uiuc.edu/SDG/IT94.

Sperberg-McQueen, C. M. “Re: Measuring the complexity of DTDs” post to newsgroup: comp.text.sgml, article # 6469, May 12, 1997.

United States Department of Defense, MIL-M-38784C,ftp://ftp.fedworld.gov/pub/cals/cals.htm (38784C.ent) May, 1991.

Woods, W. A. *Augmented Transition Networks for Natural Language Analysis*. Report No. CS–1, The Aiken Computation Laboratory, Harvard University, Dec. 1969, pp. 60–99.