Extreme Markup Languages 2003

Montréal, Québec
August 4-8, 2003

Datatype- and namespace-aware DTDs
A minimal extension

Fabio Vitali
University of Bologna

Nicola Amorosi
University of Bologna

Nicola Gessa
University of Bologna

Abstract

DTDs and XML Schema are important validation languages for XML documents. They lie at opposite ends of a spectrum of validation languages in terms of expressive power and readability. Differently from other proposals for validation languages, DTD++ provides a DTD-like syntax to XML Schema constructs, thereby enriching the ease of use and reading of DTDs with the expressive power of XML Schema. An implementation as a pre-processor of a Schema-validating XML parser aids in ensuring wide support for the language.

Datatype- and namespace-aware DTDs

A minimal extension

Table of Contents

EXTREME MARKUP LANGUAGES 2003

Datatype- and namespace-aware DTDs
A minimal extension

Fabio Vitali, Nicola Amorosi, and Nicola Gessa

§ Introduction

Generalized markup languages allow document designers not only to use arbitrary labels to designate fragments of their documents, but to impose constraints on the use of these labels as well. Validation languages serve for expressing both the list of legal labels and algebraic formulations of the required contraints on these labels. Moreover, validation languages are used not only to validate documents, but also to assist human personnel editing new documents compliant with their validation schemas.

The first widespread generalized markup language, SGML [ISO86], relied on just DTDs [Document Type Definitions] as validation language. Its offspring, XML, has seen the birth on many different languages for this purpose: starting off with just DTDs, as in SGML, but then proceeding to XML Schema [TBMM01], Relax NG [CM01], Schematron [Jel02], and many many others providing a large number of different constructs for similar or different purposes.

XML DTDs have been specified in the very same W3C recommendation that defines the XML metalanguage [BPSM98]. Thus, the concept of validation in this document specifically refers to the verification of the adherence of a document to the constraints expressed in the DTD.

On the other hand, new and unforeseen developments in the usage scenarios of XML and additional related recommendations have made the original feature set of the DTD language inadequate for constraints on modern-day XML documents.

For instance, implementers of data interchange applications needed mechanisms to express constraints not just on element structures, but also on content values (*e.g.*, to require that a given element contain a valid date between 1/1/2003 and 31/12/2003), and the (rushed?) introduction of the W3C Namespace recommendation [BHL99] made no provision for changing the namespace-unaware syntax of DTDs. The final result was that whoever wanted to create namespace-aware XML documents had to either give up validation altogether, or employ a number of undignified hacks to allow qualified names (with, unfortunately, predefined prefixes) in DTD specifications.

For this reason, the W3C and many others have started initiatives for developing new and sophisticated validation languages for XML. Among the common features of these languages we can find the switch to an XML-based syntax, the introduction of (more or less) sophisticated data types, and the support for namespaces.

An important example is XML Schema, a W3C recommendation for a complex validation language based on a sophisticated data typing mechanism and an XML-based syntax [BM01]. XML Schema provides full namespace support, sophisticated constraints on markup structures and content values, subtleties and oddities in its specifications, and an incredibly verbose syntax that doubles or triples documents' length compared to DTD. A simple example can be seen with XHTML: the normative DTD [W3Ca] for XHTML strict is 578 lines long, while the corresponding non-normative XML Schema [W3Cb], providing minor additional constraints, lies at 1349 lines in length.

Actually, all of the proposed validation languages tend a bit towards the lengthy side. A sad but true effect of the adoption of XML syntax is the sudden and huge rise in length of all related documents. Another immediate drawback of these languages is that there is no support for general entities, although they are still rather useful and used.

Unfortunately, this immediately leads to problems, for instance, when trying to use general entities in a document that needs to be validated with XML Schema; since entities need to be defined in a DTD subset, several modern XML parsers, upon encountering a DTD, will try to validate the document against it, and will supply a huge number of errors trying to find the definition of elements and attributes, which are not defined there, but in the XML Schema.

Call us traditionalists, but we quite like the syntax of DTDs. We find it a good compromise between expressiveness and simplicity, in our view two of the main aspects to consider in the definition of a schema language for XML documents. DTDs are lean, compact, easy to learn, and easy to study documents that can be read, if needed, by naked eyes, without resorting to specialized tools of dubious help. We believe, in summary, that DTDs' worst shortcomings are in the feature set, rather than in its syntax.

Furthermore, we are puzzled by the fact that working groups and individuals alike have found it irresistible, when creating their own new validation language, to come up with not only a list of supported features and an XML-based syntax, but also new and peculiar validation paradigms that require undue care, attention, and time to master and use appropriately.

We find it pointless and confusing to create — as has been done — new validation languages that, prior to the availability of an improved feature set, require document architects to learn a new syntax, a new validation paradigm, and new quirks and new peculiarities, and to throw away experience, craft, and a number of legacy applications and DTD instances.

Having a good DTD to validate documents, there are two perceived ways to enhance the capability to manage datatypes:

1. Transform the DTD into a XML Schema document, with the resulting loss of readability
2. Transform the DTD into another validation language, with the need for a new validation paradigm

On the contrary, we have set forth to create DTD++, our own validation language for XML documents, to provide the additional features that no one denies being necessary for modern-day XML validation, but to refrain as much as possible from imposing new syntax and new validation paradigms to document architects.

In particular, our design goals have been:

- To create a language that is a direct extension of a DTD, meaning that:
  - Every legal DTD is also a legal DTD++.
  - Every legal DTD++ can be typographically stripped of its DTD++-specific syntax extensions to obtain a legal DTD providing a (obviously less stringent) validation for the same document class.
  - Syntactical extensions to DTD syntax are kept to an absolute minimum
- To use XML Schema as the source for the most reasonable set of extended features, meaning that:
  - The largest possible set of XML Schema features is directly available within DTD++.
  - Every legal DTD++ is semantically equivalent to a legal XML Schema.
  - Validation of XML documents against DTD++ is identical to the validation of the same document against its corresponding XML Schema, minus the conversion of error messages.

Indeed, the last design goal actually allows us to provide widespread support for DTD++, since DTD++ documents can be converted on the fly to XML Schema and use any Schema-validating XML parser to provide validation. Thus, the DTD++ validator we have created is just an XML Schema front-end, which converts DTD++ files into XML Schema DOMs, activates the Schema validation on XML documents, and converts XML Schema validation errors back into DTD++ terms.

In the next section, we will report a few related activities and discussions about validation languages that may provide further illustrations of the finer points of our proposal. In the subsequent section, we will illustrate the extended syntax of DTD++, providing a few examples and comparisons with XML Schema. A few converted examples taken from well-known XML Schemata and DTDs complete the paper.

§ Related works

The literature describing the relative merits of validation languages for XML documents is huge and varied.

An important early work is surely MSL [BFRW01], an attempt to model and formalize the core aspects of XML Schema to aid in the process of evolving its specification and tools. MSL, which uses inference rules notation, has managed to provide a concise and precise description of XML Schema (in spite of the original specification that requires a couple of hundred pages to describe) that captures the essence of these features. This study has led to the creation of a list of problems with the XML Schema recommendation such as complexity, transitivity, arbitrary uses of rules of restriction, and use of wildcards with equivalence classes.

Rick Jelliffe [Jel01] has described the state of the art of schema language for XML documents as of the end of 2001. He considers eight (8) different schema languages, including DTD and XML Schema, and points out a few controversial aspects of the XML Schema structure specification that need further development. On the other hand, XML Schema datatypes are found to be elegant and widely accepted. Again, complexity jumps out as the main weakness of XML Schema, whereas DTD has the advantage of its greater terseness.

Lee and Chu [LC00] have made a comparative analysis of six XML schema languages: XML DTD, XML Schema, XDR, SOX, Schematron, and DSD. Their study classifies these languages with respect to a few points of view , such as “ease of use”, “language characteristics”, “expressive power”, and “requirements for dbms”. DTD tends to have the weakest expressive power because of its shortcomings in datatypes and constraints and its minimal schema structures, while XML Schema appears to have the strongest expressive power. On the other hand, DTD, due to its simple language specification, is described as the easiest schema language to learn, especially compared to XML Schema.

A few interesting proposals do refer to extensions to DTDs. The first and oldest actually dates back to SGML; the HyTime `lextype` attribute [DD94] allowed authors to specify a datatype for attributes and simple elements, thereby creating a HyTime error (although not a validation error) whenever an attribute contained a value not matching the rules expressed in the `lextype` attribute.

DT4DTD [BGP00] is a submission to the W3C providing a mechanism to specify datatypes for elements and attributes that does not break the plain DTD syntax. Two additional attributes are foreseen for each element in the DTD; `e-dtype` has as its fixed value the name of the datatype for the element, while `a-dtype` has as its fixed value the list of the datatypes for each attribute of the element, in the order in which they are presented in the `ATTLIST` element. Datatypes are named through notation declarations that reserve the name and associate it to a unique URI, which allows for the specification of user-defined types. No provision is made for namespace support. DT4DTD is meant as a migration path towards XML Schema using just plain DTD syntax.

In [SM00], Sperberg-McQueen proposes a method for expressing a limited form of context-sensitive rules in XML Schema (*e.g.*, to impose that the content model of a `P` element in HTML can contain `INPUT` elements if `P` is within a `FORM` element, and not otherwise). While some kinds of context-sensitive constraints can be expressed in XML Schema, they cannot be provided in XML DTDs (SGML DTDs were slightly better at that, too). Thus, an imaginary syntactical extension to DTDs is proposed to take care of these problems, allowing types to be defined (through a `TYPE` construct) and type-related conditions to be expressed in element content models.

DSDL (ISO/IEC 19757, [DSDL]) is an on-going effort of ISO/IEC aimed at creating “a framework within which multiple validation tasks of different types can be applied to an XML document in order to achieve more complete validation results than just the application of a single technology”. Originally, the project encompassed all four of the major schema languages, *i.e.*, RELAX NG, Schematron, DTD, and W3C Schema. In more recent times, it has become clear that the emphasis of the working group is toward support for publishing scenarios (to the detriment, we assume, of data interchange use cases), and that support could be relinquished for XML Schema (except for its data types) and DTDs.

In fact, although foreseen in the organization of the activities of the working group, DSDL part 9, called “Datatype- and namespace-aware DTDs”, appears to this date an empty box. In particular, in [Hol02], G. Ken Holman, one of the editors of the ISO proposal, has been actually wondering whether there is any sense in pursuing the topic further, given the impression that most users of the DSDL proposal may probably switch *en masse* to Relax NG anyway.

This somewhat disappointing confession came out as a reply to a post by John Cowan [Cow02], which provided a few ideas he would like to see implemented within part 9 of the DSDL framework as extensions of the DTD syntax. Holman’s answer is surprising, especially considering the amount of comments that Cowan’s post raised in just a few days.

Cowan's post provided syntactical suggestions for the following issues:

1. Namespaces declaration
2. Datatypes for attributes
3. Simple datatypes for elements
4. Datatype lists
5. Datatype choice
6. Restoring the SGML & connector
7. Abandoning SGML 1-ambiguity rules
8. Restoring SGML multiple element and attribute definitions
9. Supporting fixed element content
10. Restoring SGML inline comments (in a later post)

The subsequent posts included a number of further suggestions and critiques, some of which provided additional or alternative suggestions to Cowan's own. In Cowan's post, as well as in many of its responses, the temptation was strong to include new features (which were present in SGML but abandoned in XML, or never even proposed anywhere), so to create a new validation language. In other responses, the attitude was more conservative, aiming at identifying the minimal superset of DTD (*e.g.*, namespace + simple predefined datatypes) that could be devised, and suggesting the use of other languages (*e.g.*, Relax NG) for more sophisticated constraints.

§ DTD++

As mentioned in the introduction, we find DTD syntax too easy to learn, remember, and write to be abandoned lightly. Imposing XML Schema as the natural alternative to DTDs seems a tough and steep change, especially in situations where enhancing an existing DTD with a few stricter constraints could be useful, but not at the cost of completely turning the specification around and basically rewriting it.

As has been mentioned, XML Schema simply has too many ways to do the same thing, and many of its subtleties and quirks have led experts [Kaw01] to suggest that document architects avoid using some of its features (such as types) and stick to simpler constructs (*e.g.*, element groups). Furthermore, the XML Schema specification is simply too verbose, and is rather hard to read without the aid of a specialized tool, self-evidence and readability being strong advantages of DTDs, instead.

Thus, we set about to create DTD++, a new validation language that inherits the ease of learning, remembering, and writing of DTDs. At the same time, we decided that we needed to refrain from providing yet another validation paradigm and yet another feature set. XML Schema seemed to offer the most natural set of features to choose from; backed by the W3C, well-known and discussed, with plenty of implementations around, XML Schema provided just the right collection of features to support.

In a way, we wanted to create a DTD-like syntax providing the same constructs of XML Schema (or a reasonably wide subset thereof).

From a syntactical point of view, the DTD++ language has to be as close as possible to DTDs; in particular, it needs to be a real superset of DTD, so that every legal DTD is a valid schema in this language, too. Furthermore, as much as possible, we want the extensions to be non-intrusive into the well-known DTD syntax. To this end, the extensions need to be typographically removable from the specification, thereby leaving a legal DTD that validates pretty much the same class of XML documents (of course, without the additional constraints). Finally, as much as possible, we wanted to use and reuse the existing constructs of DTDs, and provide as little new syntax to the language as possible . Although this led us to debatable and possibly inelegant solutions, we have ended up with a syntax that has one new construct (`TARGETNS`) and a few syntactical expressions interspersed within classical DTD constructs.

From a semantic point of view, we decided that we needed to keep the DTD validation paradigm while at the same time enriching it by importing those XML Schema constructs that we felt compatible with it, and skipping some unnecessary (in our minds) features of XML Schema. A design goal we felt important was providing a syntax that could create specifications semantically equivalent to some reasonable and useful subset of XML Schema. This has had two advantages: on one hand, it has deferred discussions about the correctness and completeness of our language to XML Schema, which has wider

Figure 1: Defining namespaces with the TARGETNS construct

```
<!TARGETNS "http://www.foo.org">
<!TARGETNS ns "http://www.bar.org">
<!ELEMENT name (ns:firstname)>
<!ELEMENT ns:firstname (#PCDATA)>
```

shoulders than anything we could create with our own limited strengths; on the other, it has allowed us to make the DTD++ engine a simple pre-processor to an XML Schema validator, thus limiting the development efforts and the risk of inaccuracies in our implementation.

DTD++ currently supports namespaces, predefined simple datatypes, facets, occurrence constraints, user-defined simple and complex datatypes, the ANY content model, namespace-constrained ANY content models, the XML Schema version of the ALL content model, the XML Schema version of the mixed content model, and a number of other minor extensions.

However, no support is provided for local definitions of elements and attributes (with the corresponding madness of choosing one among Russian Dolls, Salami Slices, and Venetian Blinds styles of type definitions [Cos03]), locally unqualified elements, locally qualified attributes, imports, inclusions, redefines, and few other intricacies. As for keys and unique values, no support exists for them yet, but we plan to look into this topic soon.

Namespace declaration

Differently from XML Schema and other validation languages, DTD++ specifications are not XML files. Thus, namespaces are required by the DTD++ syntax, but only as a means to provide a context for rules and constraints. Thus, DTD++ only defines namespaces as target namespaces, for which rules are expressed by the specification.

To enforce this distinction, the TARGETNS construct has been introduced. This is the only new construct introduced by the DTD++ syntax. TARGETNS contains the URI of the namespace and (possibly) a prefix used locally (*i.e.*, in the DTD++ specification) to associate elements and attributes to the namespace.

The example in Figure 1 shows the definition of two namespaces and two elements. The element name is bound to the namespace "http://www.foo.org", while the second is bound to "http://www.bar.org". Of course, the prefixes are scoped within the specification: document instances will need to explicitly contain the xmlns specification as usual, and they may associate different prefixes to these namespaces.

It is important to note that XML Schema allows exactly one target namespace per file, and requires specifications for multiple namespaces to be separated in different resources and imported, while DTD++ allows any number of TARGETNS constructs to be specified in the same document. The DTD++ pre-processor will, in fact, create as many dummy XML Schema specifications as there are TARGETNS, and import them as needed.

Simple types

Predefined simple types

In XML Schema, simple types are a mechanism to impose constraints to the values of an attribute or an element having character content (*i.e.*, a way to constrain strings to some set of values). XML Schema provides a long list of predefined simple datatypes, as well as the possibility of deriving new types from existing ones by restriction, union, and list.

DTDs, however, only have one datatype for character elements (#PCDATA) and a handful of datatypes for attributes (CDATA, enumerations, IDs, IDREFs, and so on). Please also note that all these DTD datatypes are available, with the same name, within XML Schema.

Figure 2: Anonymous and named simple types

```
<!ELEMENT position (#INTEGER[0,99999]) >
<!ENTITY # myInteger "(#INTEGER[0,99999])" >
<!ELEMENT position #myInteger; >
```

DTD++ provides all the predefined types of XML Schema, with no distinction between attributes and character elements. This means that one may use `CDATA` for elements and `#PCDATA` for attributes with no ill consequences. All predefined types are given a name in the same vein as in DTDs: `#STRING`, `#INTEGER`, `#DATE`, `#FLOAT`, and so on.

User-defined simple datatypes

Besides predefined datatypes, XML Schema allows document architects to create new datatypes by deriving existing ones. Derivation usually happens by restriction (only a subset of the supertype values are legal in the subtype), union (the combined set of the legal values of two or more base types are legal in the derived type), and list (a space-separated list of legal values of the base type is legal in the derived type).

User-defined datatypes can be anonymous (*i.e.*, used directly, with no specification of names, in the definition of an element or an attribute) or named (*i.e.*, with a separate definition and referred to by name by element and attribute definitions).

DTD++ introduces user-defined simple types directly deriving from best practices in DTDs: content models and parameter entities. Thus, an anonymous user-defined datatype for an element looks just like the content model of an element, while named datatype are introduced by a special kind of parameter entity (whose only difference lies in the use of the # character instead of %).

Figure 2 shows two equivalent definitions of an element, the first using an anonymous datatype, and the second a named datatype (`#myInteger`), constrained to be an integer ranging from 0 to 99999. The decision to use a special `ENTITY` reflects the widespread habit of using parameter entities in DTDs as a way to separate shared content models from the actual element and attribute definitions.

Derivation by restriction and facets

A simple datatype derived by restriction is in XML Schema a new datatype that allows the subset of legal values for the supertype that satisfy a few additional constraints called “facets”.

XML Schema defines a number of facets, some of which are valid for all base datatypes, and some only for a few of them. Among them are `length`, `maxLength`, `minLength`, `maxInclusive`, `maxExclusive`, `pattern`, `enumeration`, etc.

In order to remain aligned with the minimality of the DTD syntax, and to keep definition of user-derived types well within a single line, we decided to derive some syntax from regular expressions, as is vaguely done in other parts of DTD syntax.

Thus, the specification of facets in a subtype is done via the juxtaposition of short expressions separated by a number of brackets and other typographical paraphernalia.

Derivation by list and union

List types in XML Schema are whitespace-separated collections of values, each individually legal according to the base type. DTD++ uses the “+” character after the base type name to specify that a list of one or more values are legal for the derived type. The first line of Figure 4 defines a list type of integers.

Union types in XML Schema are derived from two or more base types; a legal value of the derived type is any value that is valid according to one of its base types. DTD++ uses the “|” character to specify the choice of values from either datatype. The second part of Figure 4 shows the definition of a union type based on two user-defined types.

Figure 3: Facets in user-defined simple types

```
#INTEGER[0,100]     <xsd:minInclusive value="0"/>
        <xsd:maxInclusive value="100"/>

#INTEGER]0,100[     <xsd:minExclusive value="0"/>
        <xsd:maxExclusive value="100"/>

#STRING{10}       <xsd:length value="10"/>

#STRING{10,}      <xsd:minLength value="10"/>

#STRING{10,20}      <xsd:minLength value="10"/>
        <xsd:maxLength value="20"/>

#STRING(AK|AL|AR)or(AK|AL|AR) <xsd:enumeration value="AK"/>
        <xsd:enumeration value="AL"/>
        <xsd:enumeration value="AR"/>

#STRING/\d{3}-[A-Z]{2}/   <xsd:pattern value="\d{3}-[A-Z]{2}"/>
```

Figure 4: List and union type definitions

```
<!ENTITY # listOfMyIntType (#myInteger;+)>
<!ENTITY # USStateList (#STRING(AK|AL|AR)+)>

<!ENTITY # zipUnion ((#USState;)|(#listOfMyType;))>
```

Figure 5: A named complex type

```
<!ENTITY @ USAddress "(name,street,city)" "country (#NMTOKEN) 'US' ">
```

It should be noted that the syntax is not ambiguous in its use of the "+" and "|" characters for derived simple types; the other uses of these characters in structured content models can be easily identified since they do not allow names of simple types in their definitions.

Complex types

A syntactical issue in porting complex types to a DTD-like syntax is the fact that XML Schema complex types specify both contained elements and attributes, while a DTD uses two different constructs for content models and attribute lists.

DTD++ uses an intermediate approach: anonymous complex types in DTD++ are absolutely equivalent `ELEMENT` and `ATTLIST` constructs for structured content models in DTDs. Named complex types, on the other hand, use another version of the parameter entities of DTDs that is identified by the letter "@".

As shown in Figure 5, complex type definitions in DTD++ have two parts, one for the element content model, and the second for the attribute content model. Both are optional and can be omitted when they are not needed.

Occurrence constraints

XML Schema provides much finer control for the repeatability and optionality of elements and structures in a content model than DTDs, allowing numerical specification of the minimum and maximum number of occurrence of each structure. DTD++ provides a construct to specify these constraints, as shown in Figure 6. A missing first value indicates a required presence, while a missing second value indicates an "unbounded" number of elements.

Figure 6: Elements constrained in occurrence

```
<!ELEMENT PurchaseOrder (shipTo,billTo,comment[0,5],items)>
```

Figure 7: A derivation by restriction

```
<!ENTITY @ NewPurchaseOrderType @PurchaseOrderType; "(shipTo,billTo,comment,
items)">
```

Figure 8: A derivation by extension

```
<!ENTITY @ Address "(name,street,city)">
<!ENTITY @ USAddress "@Address;,(state,zip)">
```

Of course, standard DTD occurrence specifications, such as “+”, “?”, and “*”, can still be used to mean, respectively, “[1,]”, “[0,1]”, and “[0,]”.

Deriving complex types

Complex types can be derived as well in XML Schema. Derivation is either by restriction or extension.

Derivation by restriction implies that every legal instance of the subtype satisfies also the constraint of the supertype. Syntactically, the specification of a restricted complex type in XML Schema must repeat all of the elements of the base type, plus all additional constraints. From another point of view, we could think of a restricted type as a complete type definition, with the specification of a dependency relationship to another type.

Derivation by extension, on the other hand, appends new elements or attributes to an existing type. No complex structures can be defined that connect the elements of the base type with the ones added in the derivation, since they must be connected by a sequence operator.

DTD++ cannot derive complex types anonymously (unless only attributes are added). Derivation must happen within a @ entity construct. A derivation by restriction simply specifies the base complex type before providing the actual content model. A derivation by extension, on the other hand, includes the base complex type name in a sequence before the new elements.

Mixed content models

Mixed content models allow character content to be interspersed with elements. Paragraphs in HTML are a good example for this kind of construct. DTDs have a very simple and limited format of content model, according to which no constraint in order, number, or position can be attached to elements specified in a mixed content.

XML Schema proposes a relaxation of these limitations by allowing any constraint and structure of the elements of a mixed content, while still allowing characters anywhere within the content.

Thus, DTD++ needs to provide a more extended syntax than DTDs for mixed content models. This is done by prefixing any structured content model with the keyword `#PCDATA`, as shown in Figure 9.

`ALL` content model

SGML had an additional operator for a structured content model, the so-called “&” operator, which signaled that an element had to be present in the content, but in any position. This operator disappeared in the XML DTD syntax, but the “`all`” element, a similar construct although with limitations, was reintroduced in XML Schema.

Figure 9: An extended mixed content model

```
<!ELEMENT letterBody (#PCDATA (salutation,quantity,productName,
shipDate[0,1]))>
```

Figure 10: Classical and extended syntax for mixed content

```
<!ELEMENT salutation (#PCDATA | name | address)*>
<!ELEMENT salutation (#PCDATA (name | address)*)>
```

Figure 11: Defining an `ALL` content model

```
<!ENTITY @ PurchaseOrderType  "(shipTo & billTo & comment? & items)">
```

Figure 12: Defining a namespace-constrained `ANY` content model

```
<!TARGETNS "http://www.foo.it">
<!TARGETNS ns "http://www.bar.it">

<!ELEMENT book (chapter)>
<!ELEMENT ns:paragraph (#PCDATA)>
<!ELEMENT chapter ANY[1,]{http://www.bar.it}>

<!ELEMENT ns:note (#PCDATA)>
<!ELEMENT footer (#PCDATA)>
```

DTD++ reintroduces the “&” operator that was used by SGML, but imposes the same limitations as XML Schema. Figure 11 shows the DTD++ definition of an element with the “&” operator.

`ANY` content models

Both DTDs and XML Schema allow an element’s content model to be defined as `ANY`, which means that any other element is allowed inside it. XML Schema adds support for occurrence constraint and namespace specification in `ANY` content models. It further adds a few predefined namespace constants, such as “`##other`”, “`##targetNamespace`”, and so on.

DTD++ inherits the `ANY` syntax of DTDs and adds a way to specify the occurrence and namespace constraints of XML Schema. Figure 12 shows an example of this.

The `chapter` element is defined as containing one or more elements belonging to the namespace `http://www.bar.it`. Thus, the example in Figure 13 will raise an error, because the `chapter` element contains an element not belonging to that namespace.

Implementation and testing

We have implemented a pre-processor for our DTD++ language. This reads an XML document, finds both the internal and the external sets of the DTD++, creates the corresponding XML Schema (or Schemata, in case the DTD++ refers to more than one namespace), and passes the whole set of documents to the parser for the actual validation. Upon receiving errors, the pre-processor converts the messages so that they can be understood in terms of the original DTD++.

Figure 13: An example of a non-valid use of the ANY content model

```
<!DOCTYPE book SYSTEM "DTD1.dpp">
<book xmlns="http://www.foo.it" xmlns:ns="http://www.bar.it">
  <chapter>
    <ns:paragraph>...</ns:paragraph>

    <!-- Non valid!!! -->
    <footer></footer>

  </chapter>
</book>
```

The implementation has been tested with a number of document classes. In all cases, we started with their XML Schema, wrote a reasonable DTD++, and verified that the documents still validated according to the DTD++. In all cases, we found a quantifiable reduction in size (between two- and three-fold) and a (subjective) vast improvement in readability and usability of the specification.

An example may suffice: the XHTML language is provided with both a DTD and an XML Schema specification. The DTD is (net of comments) 578 lines long, while the XML Schema is (without comments and annotations) 1349 lines. The XML schema provides a few and important additional constraints with respect of the DTD.

We wrote a DTD++ version of the XHTML trying to merge the constraints specified in the XML Schema with the specification style of the DTD, obtaining a DTD++ specification which is semantically equivalent to the XML Schema and is exactly as long as the DTD (577 lines).

§ Conclusions

The literature seems to agree that schema languages for XML documents lie between the two extremes of a DTD, that has maximum terseness and readability, but minimum expressive power, and XML Schema, that has the greatest expressive power but a much lesser clarity and conciseness.

Additionally, coexistence of different schema languages within the same document still are not straightforward. DTD subsets are required when using general entities, and some parsers get overly confused dealing with entities defined in a DTD, and elements and attributes in a different schema document.

Our proposal aims at finding a reasonable compromise between the expressive power of XML Schema and the ease of use and compactness of a DTD. What we decided first was that there was no sense in creating a completely new language; extending an existing syntax with features taken from another existing language seemed, and still seems now, a much better approach.

Of course, the final result is still incomplete and partial. In particular, support for keys and unique values that exist in XML Schema has not been provided yet. Still, the experience so far with the DTD++ language appears to be interesting and rewarding.

Bibliography

[BFRW01] Brown, Allen, Fuchs, Matthew, Robie, Jonathan, and Wadler, Philip. MSL – A model for W3C XML Schema. May 2001. **http://www.research.avayalabs.com/user/wadler/papers/msl/msl.pdf**.

[BGP00] Buck, Lee, Goldfarb, Charles F., and Prescod, Paul. *Datatypes for DTDs (DT4DTD) 1.0*. W3C Note, 13 January 2000. **http://www.w3.org/TR/dt4dtd**.

[BHL99] Bray, Tim, Hollard, Dave, and Layman, Andrew. *Namespaces in XML*. Jan 1999. **http://www.w3.org/TR/REC-xml-names**.

[BM01] Biron, Paul V., and Malhotra, Ashock. *XML Schema Part 2: Datatypes*. May 2001. **http://www.w3.org/TR/xmlschema-2/**.

[BPSM98] Bray, Tim, Paoli, Jean, and Sperberg-McQueen, C. M. *Extensible Markup Language (XML) 1.0*. Feb 1998. **http://www.w3.org/TR/REC-xml**.

[CM01] Clark, James, and Makoto, Murata. Relax NG. 03 Dec 2001. **http://relaxng.org/spec-20011203.html**.

[Cos03] Costello, R.L. XML Schemas: Best Practices. 17 Feb 2003. **http://www.xfront.com/BestPracticesHomepage.html**.

[Cow02] Cowan, John. Come On, DTD, Come On! Thoughts on DSDL Part 9, message to: xml-dev@lists.xml.org. Tue, 11 Jun 2002. **http://lists.xml.org/archives/xml-dev/200206/msg00449.html**.

[DD94] DeRose, S. J. and Durand, D. G. *Making Hypermedia Work*. Kluwer Academic, Publishers. 1994.

[DSDL] *ISO/IEC 19757 – DSDL Document Schema Definition Languages*. **http://xml.coverpages.org/dsdl.html**.

[Hol02] Holman, G. Ken. Re: [xml-dev] DSDL part 9: new namespace declarations not needed as part of DTD syntax?, message to: xml-dev@lists.xml.org. Sat, 15 Jun 2002. **http://lists.xml.org/archives/xml-dev/200206/msg00715.html**.

[ISO86] *ISO 8879:1986. Information processing – Text and office systems – Standard Generalized Markup Language (SGML).*

[Jel01] Jelliffe, Rick. The Current State of the Art if Schema Language for XML. 2001. **http://www.planetpublish.com/pdfs/RickJelliffe.pdf**.

[Jel02] Jelliffe, Rick. The Schematron Assertion Language 1.5. 01 Oct 2002. **http://www.ascc.net/xml/resource/schematron/Schematron2000.html**.

[Kaw01] Kawaguchi, Kohsuke. W3C XML Schema: Dos and DON'Ts. 2001. **http://www.kohsuke.org/xmlschema/XMLSchemaDOsAndDONTs.html**.

[LC00] Lee, Dongwon, and Chu, Wesley W. Comparative Analysis of Six XML Schema Languages. Sep 2000. **http://nike.psu.edu/publications/sigmod-record-00.pdf**.

[SM00] Sperberg-McQueen, C. M. *Context-sensitive rules in XML Schema*. 15 February 2000, rev. 25 April 2000. **http://www.w3.org/2000/04/26-csrules.html**.

[TBMM01] Thompson, Henry S., Beech, David, Maloney, Murray, and Mendelsohn, Noah. *XML Schema Part 1: Structures*. May 2001. **http://www.w3.org/TR/xmlschema-1/**.

[W3Ca] W3C. *Xhtml1-strict.dtd*. **http://www.w3.org/2002/08/xhtml/xhtml1-strict.dtd**.

[W3Cb] W3C. *Xhtml1-strict.xsd*. **http://www.w3.org/2002/08/xhtml/xhtml1-strict.xsd**.

The Authors

Fabio Vitali
University of Bologna, Department of Computer Science
Mura A. Zamboni, 7
Bologna
Italy
fabio@cs.unibo.it

Fabio Vitali is a professor at the Department of Computer Science at the University of Bologna. He holds a Laurea degree in Mathematics and a Ph.D. in Computer and Law, both from the University of Bologna. His research interests include markup languages; distributed, coordinated systems; and the World Wide Web. He is the author of several papers on hypertex functionalities, the World Wide Web, and XML.

Nicola Amorosi
University of Bologna, Department of Computer Science
Mura A. Zamboni, 7
Bologna
Italy
amorosi@cs.unibo.it

Nicola Amorosi holds a Laurea degree in Computer Science from the University of Bologna.

Nicola Gessa
University of Bologna, Department of Computer Science
Mura A. Zamboni, 7
Bologna
Italy
gessa@cs.unibo.it

Nicola Gessa holds a Laurea degree in Computer Science from the University of Bologna and has been a Ph.D. student since January 2003.

Extreme Markup Languages 2003
Montréal, Québec, August 4-8, 2003
This paper was formatted from XML source via XSL
by Mulberry Technologies, Inc.