Extreme Markup Languages 2005®

Montréal, Québec
August 1-5, 2005

Design patterns for descriptive document substructures

Fabio Vitali
University of Bologna

Angelo Di Iorio
University of Bologna

Daniele Gubellini
University of Bologna

Abstract

Existing schema languages can lead to overdesign. They offer more choices than are necessary for purely descriptive (as contrasted with prescriptive) situations. A potential solution is to design based on “patterns” from real DTDs. Using three example situations, alternatives, repeatable homogeneous elements, and mixed content models, we derived a group of patterns sufficient to express all required structures in a descriptive environment. To provide a meaningful example, we propose a new instance-based schema language, DTD--, that derives schemas from tagged sample instances according to the patterns. Since there are few patterns, every document can be represented by a simple grammar where grammar rules can be directly inferred from the document, without any ambiguity.

A Conference of IDEAlliance
Extreme Markup 2005 Languages®

Design patterns for descriptive document substructures

Table of Contents

Introduction....1
The need for patterns: three examples....2
- Alternatives....2
- Repeatable homogeneous elements....2
- Mixed content models....3

Some patterns for document structures....4
- Marker and Atom....4
- Block and inline elements....4
- Record....5
- Table....5
- Additive and Subtractive Context....5

DTD--: Limiting possible structures to patterns only....6
- Atom, Record and Table....7
- Block, Marker and Subtractive Context....7
- Additive Context....8

Processing DTD--....8
Discussion and merits of DTD--....11
Conclusions....13
Bibliography....13
The Authors....15

A Conference of IDEAlliance
Extreme Markup Languages® 2005

Design patterns for descriptive document substructures

Fabio Vitali, Angelo Di Iorio, and Daniele Gubellini

§ Introduction

Within the SGML/XML communities, a topic that has characterized the field and separated it from others dealing with data management, description and storage has been the distinction between "descriptive" (or "generic") and "procedural" markup. This is meant to help flexibility in downstream applications (i.e., those that handle the data after it has been marked up) in that the documents contains labels that tells things about what a document fragment is and what is its role, which never changes regardless of the applications, rather than labels that tells how each document fragment can be used, and thus is only adapt for just one downstream application to the detriment of all others.

But besides this distinction, another is interesting to point out, between "descriptive" and "prescriptive" markup. Although this distinction has been often implicitly or explicitly made with regard to document models (i.e., schemas and DTDs [Document Type Definitions]), we still think there is room for public reflection on the issues. In this dichotomy, descriptive is used to refer to markup that simply states some quality about each text fragment, without trying to impose any rule on how and where it should appear, while prescriptive markup would, besides simply providing names for the labels to use in the markup, also impose constraints and structural rules on the use and positioning of labels.

If marking up a document is expressing semantics, in that some meanings that can be drawn from some content are made explicit, we can in a way state that prescriptive models give the most expressive power to the document designers, and make document authors subject to the power of the constraints, while descriptive models reflect the fact that sometimes document authors work and have worked independently of the desires of the document designers, and thus the latter have to accommodate variations, exceptions, differences, etc.

Working in a descriptive environment in this second sense (opposed to prescriptive, rather than to procedural) imposes a number of constraints to the activities of the document designers. The temptation to over-design (i.e., to impose too many constraints on document structure, as if we were in a prescriptive situation) is strong, and may lead to situations where actual documents cannot fit the structure because they are too different from the "natural" candidates. On the other hand, the temptation to under-design (i.e. To give up and say "anything goes") is also to be fought, because this would lead to major differences in markup of the same documents given by the lack of absolute standards to refer to.

It is our contention that for descriptive markup the syntactic choices available in validation languages such as DTDs, XML Schema [TBMM01] and Relax NG [CM01] are too powerful, and easily lead to over-design. On the other hand, just creating a flat vocabulary of element names with `ANY` as the generic content model would constitute an extreme of under-design to be avoided as well.

In our mind an adequate solution can be found in describing "best practices", or even better in identifying those "patterns" that are oftentimes expressed in real DTDs and that pinpoint some more fundamental truths on common structures that all other structures expressible in the DTD syntax.

As a matter of example we consider three examples: alternatives, wrappers on repeatable elements and mixed content models. Reflecting on these three examples allows us to draw the conclusion that some patterns can be derived from well-thought-out examples and some anti-patterns from bad examples. In particular, we have devised seven patterns that hopefully (and with internal and controlled variations) can express all the structures that are possibly required in descriptive situations. From this, and the awareness that the best theory can only be described by a meaningful example, we extol the virtues of specification by example, proposing a new schema language, DTD--, that is basically an instance of a document that the language is meant to control. Tools have been designed for converting XML Schema and DTDs back and from DTD--, and for validating documents against these schemas.

© 2005 Fabio Vitali, Angelo Di Iorio, & Daniele Gubellini

§ The need for patterns: three examples

Alternatives

Let us consider a possible *either/or* situation: for instance, in an address, a document designer might decide that an address either has a P.O. Box or a street address. In a DTD like syntax, this could be rendered in a rule such as:

Figure 1: Expressing alternatives in a DTD-like syntax

```
<!ELEMENT address (name, (pobox | street), city, ZIP, state) >
```

In a prescriptive document factory, this rule effectively inhibits incorrect structures to be created, and ensures homogeneity in the created documents. In a descriptive environment, on the other hand, there is no homogeneity to be sought for documents (they exist already), but rather it is important that all existing documents are marked up at best and without ambiguities.

Now two things may happen: if in the document set there is no example of a simultaneous presence of P.O. Box and street address, then this is a constraint that has no practical effect on reality, one additional check that was not needed. If, on the other hand, a document exists that has both a street address and a P.O. Box, then the rule does not allow a correct markup, and forces the document editor to find a hack around the constraints of the DTD.

A corresponding descriptive rule would therefore be:

Figure 2: Expressing alternatives with a descriptive rule

```
<!ELEMENT address (name, pobox?, street?, city, ZIP, state) >
```

where the alternative has been transformed into a sequence of optional elements. This rule has no effect on the final markup, exposes exactly the same meanings for documents that naturally follow the stricter rule, **but** allows for the exception in case one exists.

Alternatives do not capture additional semantics with respect to a sequence of optional elements, but *a priori* exclude some situations to occur. Thus in a descriptive environment they are useless in the best cases (where all occurrences naturally follow the alternation) or a nuisance and an obstacle if an exception happens.

Repeatable homogeneous elements

It is sometimes tempting to insert a repeatable element within a sequence of different elements. For instance an address may include any number of telephone and fax numbers. One such rule could be:

Figure 3: Expressing repeatable elements in a DTD-like syntax

```
<!ELEMENT address (name, ..., state, (telephone|fax)*) >
```

It is difficult to extract any meaning from the presence of several such elements directly within the address element. Certainly they have not the same role and importance of name, street, zip or state elements. Should they be taken individually or cumulatively? Does the order of appearance have an importance?

In fact, we believe that this form of rule is just an erroneous shorthand for the real structure, which should be in our mind:

Figure 4: Expressing repeatable elements with a descriptive rule

```
<!ELEMENT address (name, ..., state, telephones?) >
<!ELEMENT telephones (telephone|fax)+ >
```

The telephones element (in its plural form) already hints that there will be one or many individual telephone elements inside, each of which should be considered as an autonomous piece of information.

Wrappers help in creating a strong structure and separation of concerns, give more clarity and visibility to the inter-relations among elements, and simplify the readability of the DTD. As a counter-example, consider the following definition from TEI [Text Encoding Initiative], and how it could be improved by the use of wrappers.

Figure 5: The definition of the `DIV` element in TEI

```
<!ELEMENT div ((argument | byline | dateline | docAuthor |
docDate | epigraph | head | opener | salute | signed | anchor | gap | index
| interp | interpGrp | lb | milestone | pb)*, (((div | divGen), (anchor |
gap | index | interp | interpGrp | lb | milestone | pb)*)+ | (((eg | bibl |
biblFull | ab | l | lg | p | sp | figure | cit | q | label | list | listBibl
| note | stage | table), (anchor | gap | index | interp | interpGrp | lb |
milestone | pb)*)+, ((div | divGen), (anchor | gap | index | interp |
interpGrp | lb | milestone | pb)*)*)), ((byline | closer | dateline |
epigraph | salute | signed | trailer), (anchor | gap | index | interp |
interpGrp | lb | milestone | pb)*)*) >
```

Mixed content models

Mixed content models are by definition used when describing semi-structured text flows that are part of larger contexts. Paragraphs that have meaningful subparts inside are natural candidates for mixed content models.

Each individual subelement of a paragraph specifies some special meaning or style on the wrapped text. For this reason, it seems just natural to assume that all text within a sub-element of a paragraph is also part of the paragraph. We believe that subelements should not be allowed to contain as elements data that is not part of the paragraph text flow, since this could be difficult to identify without precise advance knowledge of the meaning of the subelement itself and its further subparts.

Thus the only allowable forms of mixed content models should be:

Figure 6: Defining a mixed-content model to model paragraphs

```
<!ENTITY % inline "(#PCDATA | a | b | ... | z)*">
<!ELEMENT para     %inline; >
<!ELEMENT a        %inline; >
<!ELEMENT b        %inline; >
...
<!ELEMENT z        %inline; >
```

or, at most, if we want to exclude further nesting inside subelements,

Figure 7: Defining a mixed-content model to model paragraphs, excluding further nesting inside subelements

```
<!ENTITY % inline "(#PCDATA | a | b | ... | z)*">
<!ELEMENT para     %inline; >
<!ELEMENT a        (#PCDATA) >
<!ELEMENT b        (#PCDATA) >
...
<!ELEMENT z        (#PCDATA) >
```

This for is meant to specify that the content model of all elements of a mixed content are mixed content themselves (or simple text in the simplest cases), and that a block element is the only mixed content element whose content model list does not include itself (i.e., there is no para inside the inline entity).

§ Some patterns for document structures

A pattern-based approach is a vehicle for the simplified creation of well-structured, unambiguous and manageable schemas, where few design patterns are enough to express all the required structures in a descriptive environment. The other constructs produce more complicated and potentially ambiguous schemas, whose meaning could be expressed with the same power by using patterns.

Our approach relies on a basic principle: "spreading" the meta-information over the depth of the document in order to decrease the need for complex constructs. Obviously the documents based on these patterns are more "verbose": this is the cost that must be payed to allow high-level abstraction, but in the meanwhile the abstraction itself eases the understanding of the document. The patterns we propose are briefly described below.

Marker and Atom

A marker is an empty element (in case, enriched with attributes), whose meaning is strictly connected with its position within the context. It is not meant to provide characterization of the text content, but to identify special rules for a given position of the text. Examples of markers can be found in the HTML specs, such as the `hr` element or the `img` element.

Figure 8: The Marker pattern

```
<!ELEMENT hr (EMPTY) >
<!ELEMENT img  (EMPTY) >
<!ATTLIST img src %URL; #REQUIRED>
```

An atom is used to mark-up units of information, unstructured and not further divisible. The content-model of an atom is a sequence of characters, which express a basic content such as a date, a string or a number:

Figure 9: The Atom pattern

```
<!ELEMENT email (#PCDATA) >
```

Note that markers and atoms play different roles, even if a marker can be considered a valid atom without content, from a syntactical point of view.

Block and inline elements

In the fundamental idea of mixed content elements, blocks contain both text and inline structures, and it is the natural flow of the text, rather than some arbitrary rules, that determine the position of each inline structure. Furthermore it often happens that inline structure nest arbitrarily (as is the case of bold and italic elements). This means that is a descriptive environment it is hopeless and erroneous to try to impose any constraint on block elements except the complete identification of the allowable inline elements, that can nest arbitrarily.

Block elements and inline elements, thus, share the same content model, wich is mixed and contains the list of the inline elements. Block elements are distinguishable because they use the same content model, but are not listed in the allowed elements. A simple way to express this is to employ a parameter entity used by both block and inline elements and not containing the block elements.

Figure 10: The block and inline pattern

```
<!ENTITY %   inline "(#PCDATA | i | b)*" >
<!ELEMENT p %inline; >
```

```
<!ELEMENT i %inline; >
<!ELEMENT b %inline; >
```

Record

A record is a container of heterogeneous information, composed of a set of *name-value* pairs. From a syntactical point of view, the content model of a record is a list of elements, that can be atoms, blocks or other records and tables (but not inlines or markers):

Figure 11: The record pattern

```
<!ELEMENT person      (name, address, description) >
<!ELEMENT name        (#PCDATA) >
<!ELEMENT address     (name, pobox?, street?, city, ZIP, state) >
<!ELEMENT description %inline; >
...
```

Records can be used to group simple units of information in more complex structures or to organize data in hierarchical subsets. Records are meant to capture information as it is, rather than as it should be. For this reason, the record pattern lacks alternatives and repeatable subgroups, for the reasons previously explained.

Table

A table is an ordered list of homogeneous elements. Tables can be used to group homogeneous objects into the same structure and, also, to represent repeating tabular data.

Within tables we can expect to find either records or blocks, but never atoms, inlines or markers. A good way to emphasize its role as "set of homogeneous elements" is to name the table with the plural form of the name of the contained element.

Figure 12: The table pattern

```
<!ELEMENT persons (person)+ >
```

Tables are the main way for expressing repetitions. These repetitions are not expressed raw, as a subgroup, within a more complex content model, but protected by a plural-form wrapper that acts as a member of a more fundamental record.

In some situations the table can express the plural form of a category, whose singular forms can express differences within a class. Thus the table actually has an additional form that does not detract from its generality:

Figure 13: The second form of the table pattern

```
<!ELEMENT persons (child | teen | adult | elder)+ >
```

Additive and Subtractive Context

Not all situations we find in descriptive markup can be covered by the previous patterns. Exceptions and special cases abound that can be dealt with difficulty with traditional validation languages, and easily with patterns.

For instance, one may consider allowing in an element other elements already used in other parts of a document, only with a few more elements not found elsewhere. One example is immediate: the `FORM` element of HTML allows all elements in the `&flow;` entity, *plus* the special form elements such as `INPUT`, `TEXTAREA`, etc.

In a word, FORM provides a *context* for these elements. We call a context where a few elements are added in depth to existing elements an *additive context*.

A different example regards re-using a content model already used in other parts of a document, only excluding some elements. Yet again, an example from HTML can be easy: A elements cannot contain other A elements. Similarly, we could define a footnote as a regular paragraph, except that no footnotes can be defined. Here again the A element and the footnote element describe a context where some elements that would normally be allowed make no sense and should be signalled. We call them *subtractive contexts*.

The additive context and subtractive context pattern allow designers to explicitly express these relationships. Unfortunately, with traditional schema languages, it is very difficult to describe either additive or subtractive contexts: special elements can occur (or be excluded) not only directly within the container, but also within other elements inside it. In fact, only SGML's DTDs (and languages for co-constraints such as Schematron [Jel05], SchemaPath [MCV04] or DTD++[FGMV04]) can adequately describe such situations. For this reason we return to SGML syntax to express these patterns.

Figure 14: The Additive Context pattern

```
<!ELEMENT form %flow; +(input)>
<!ELEMENT input EMPTY>
```

Figure 15: The Subtractive Context pattern

```
<!ELEMENT a %inline; -(a)>
```

§ DTD--: Limiting possible structures to patterns only

While in the previous sections we have suggested a set of patterns limited enough to create well structured and unambiguous schemas, the temptation is strong to propose a schema language that allows a designer to only use exclusively these patterns. The goal of this language (which, for lack of fantasy, we call DTD--) is clear: the provision of a *tool* to help designers in applying the pattern-based approach with little effort, so that the patterns are not limited to be a cold set of guide-lines.

DTD-- is an *instance-based* schema language: a schema is a full-featured example of the document you would validate against the DTD and so, writing a DTD-- schema is as simple as writing an instance of the XML document you need to model. This feature is strictly connected with the minimality of the patterns: since there are few patterns, every document can be represented by a simple grammar where grammar rules can be directly inferred from the instance of the document, without any ambiguity.

Other *instance-based* languages were already proposed in the literature, like Examplotron [VAND03], but while the latter has many features for keeping DTD-like constructs, DTD-- limits possible syntax rules to patterns only. Thus, DTD-- is simpler and clearer, but at the same time it does not sacrifice the expressivity and the power of the schemas it is meant to create.

DTD-- brings us some other benefits:

- It is easy to learn: you only need to know XML (and few others structures introduced below) and no new special syntax.
- It is easy to use: several XML-based tools already exist, that can ease the creation and modification of DTD-- schemas.
- The creation of document is simplified, as one can simply *fill* a schema with some content as if it were a template.

- Syntax choices are limited: the patterns are the only needed constructs in order to build all the meaningful document structures.
- The creation of a grammar is simplified too. Rather than starting, as tradition demands [GLUMCG02], from the study of instances, an example-based approach skips the gap between the analysis of the instances and the actual creation of the corresponding schema.

Every abstract pattern has been mapped into a DTD-- construct, whose syntax and application will be described in the next subsections.

Atom, Record and Table

This schema shows some patterns as expressed in a fragment of a DTD-- schema:

Figure 16: Defining tables, records and atoms with the DTD-- syntax

```
<persons xmlns:dtd="http://vitali.web.cs.unibo.it/NS/dtdmm">
    <person>
        <name>Your name</name>
        <surname>Your surname</surname>
        <university>The university you come from</university>
    </person>
    <dtd:etc/>
</persons>
```

In the example in figure 16, the elements `name`, `surname` and `university` are interpreted as *atoms*, i.e., elements containing only simple text (`#PCDATA`); the element `person` is a *record*, specified by placing inside the element a set of non-repeated elements; the element `persons` (note the plural) is a *table* specified by placing the elements of the table followed by a special empty element `<dtd:etc/>`. This element expresses that we mean a table pattern (i.e., to have more of the previous elements) and not a record.

The same schema could be expressed in a DTD-like syntax as shown in figure 17:

Figure 17: Example of table, record and atoms in a DTD-like syntax

```
<!ELEMENT persons (person)+ >
<!ELEMENT person (name, surname, university) >
<!ELEMENT name (#PCDATA) >
<!ELEMENT surname (#PCDATA) >
<!ELEMENT university (#PCDATA) >
```

Note that DTD-- reserves itself a namespace (`http://vitali.web.cs.unibo.it/NS/dtdmm`, prefixed in all our examples by `dtd`) to mark-up DTD-- special elements and attributes and express information about the selected pattern.

Block, Marker and Subtractive Context

The schema in figure 18 shows the DTD-- syntax of a few more patterns.

Figure 18: An example of block, markers and subtractive context in DTD--

```
<Article dtd:type="ArticleBody" Title="Title of the
article"> Write here your article. You can markup it with
emphasis <emph dtd:type="ArticleBody"/>, or with bold
fonts <bold dtd:type="ArticleBody"/> with citation
<cit dtd:type="ArticleBody"/> and notes <note
dtd:type="ArticleBody" dtd:exclude="note"/>. You
can put some bibliographic links with <bib
ref="Reference"/> </Article>
```

In figure 18 the element `Article` is an instance of the *block* pattern, which is expressed with the special attribute `dtd:type` and a mixed content of text and elements. The elements `emph`, `bold`, `cit` and `note` are inline elements while `bib` is a simple marker.

The element `note`, on the other hand, is a *subtractive context*, as expressed by the attribute `dtd:exclude` which contanins the list of the elements you want to forbid in the subtree of the element. In the example in figure 18, recursive containments of `note` elements within other `note` is not allowed.

Additive Context

An *additive context* can be expressed by using the special attribute `dtd:include`, which contains the list of the elements specially allowed inside the subtree. A designer can also impose the unicity of an element within the context by using the symbol "!" before the element name, i.e. `dtd:include="!Sig"`. The example in figure 19, encoding a contract that can contain a signature (only one) wherever in its body, shows the syntax of this pattern:

Figure 19: An example of additive context in DTD-- syntax

```
<Contract dtd:include="!Sig">
    ...
    <Sig>Your signature</Sig>
    ...
</Contract>
```

§ Processing DTD--

After introducing, in the previous sections, the patterns and the syntax of DTD--, we introduce here some tools useful to process DTD-- schemas. Two operations are covered: the creation of a schema and the validation process against it. As previously mentioned, DTD-- is an instance-based language, so that a schema can be derived from one or more XML document(s). We have developed a DTD-- processor based on a *monotonic process of refinement*: initially the processor classifies each fragment of the source document selecting one of the patterns; in a further pass, a more appropriate pattern could be chosen for that element. For instance, a fragment can be considered a marker at the end of the first iteration, a record after the second one and, finally, a table; analogously, a fragment recognized as block (because of its mixed content-model) cannot be subsequently recognized as a table. The minimality of the patterns make these steps simple and unambiguous.

The DTD-- processor may take in input several document and infer the DTD-- schema from the analysis of similarities and differences among all input documents, through a *join operation*, as shown in the following examples.

Figure 20: An XML instance to derive a DTD-- schema

```
<film>
    <title>Lord of the Rings</title>
    <date>
        <month>4</month>
        <day>23</day>
        <year>2005</year>
    </date>
    <characters>
        <character>Frodo</character>
    </characters>
</film>
```

Figure 21: A different XML instance to derive the same DTD-- schema

```
<film>
    <title>Lord of the Rings</title>
    <characters>
        <character>Gandalf</character>
        <character>Gollum</character>
        <character>Galadriel</character>
    </characters>
</film>
```

Figure 22: Deriving a DTD-- from the two previous instances:

```
<film>
    <title>Atom</title>
    <date>
        <month>Atom</month>
        <day>Atom</day>
        <year>Atom</year>
    </date>
    <characters>
        <character>Atom</character>
        <dtd:etc/>
    </characters>
</film>
```

In figure 23 the element `characters` is classified as a table: the first occurrence of `character` is a record, and the second one is a marker (which is considered as a reference of more `character` to come). Note that a table could be specified also by using the element `<dtd:etc/>`, but this form is considered equivalent.

Figure 23: Expressing a schema through an instance:

```
 <characters>
    <character>
        <name>Atom</name>
        <age>Atom</age>
    </character>
    <character/>
</characters>
```

The *join* operation plays a fundamental role in case of a *either/or* situations. Let us consider two XML documents validated against a DTD fragment such as `<!ELEMENT contact (email | phone)>`: in the first document, the element `email`, and in the second documento the element `phone` may appear without any `email` element. In such a case, the correct schema is obtained by merging both instances. Obviously the final result can be made more precise by increasing the number of the input documents.

Although in principle logical inconsistencies could occur, in particular when the instances are based on bad-designed schemas, the system is smart enough to derive everytime a the most general schema that validates all documents provided. Since we are working within a descriptive environment, we are not interested in deducing the original, *real* DTD/Schema of these documents: so on the whole, this inference is always possible, and make sense as much as the input documents follow the DTD-- philosophy of patterns.

The validation process against a DTD-- schema has been implemented through an intermediary conversion into XML-Schema. Rather than creating a new validation engine from scratch, we prefer to convert a DTD-- schema into the corresponding XML-Schema and validating it with existing and well-tested validators. The conversion of the patterns is quite simple, since in general context models are simpler in DTD-- than in XML-Schema.

For instance, a *record* can contain only unordered and not reapeatable elements, so it will be converted by the mean of the `xsd:all` construct, with the correct values for `minOccurs` and `maxOccurs` attributes. An example is shown in figure 24. It worth to remark that in records the order of the elements is not relevant, since we are working in a descriptive environment.

Figure 24: A DTD-- record to be converted into an XML Schema definition

```
<Person>
    <Name>Your name</Name>
    <Surname>Your surname</Surname>
</Person>
```

Figure 25: Deriving an XML Schema definition from the previous DTD-- record:

```
<xsd:element name="Person">
    <xsd:complexType>
        <xsd:all>
            <xsd:element ref="Name" minOccurs="0" />
            <xsd:element ref="Surname" minOccurs="0" />
        </xsd:all>
    </xsd:complexType>
</xsd:element>
```

While the conversion of basic patterns such as atoms, blocks and tables is straightforward in XML-Schema, additive and subtractive contexts need more complex management, as they cannot be expressed directly in the XML-Schema syntax (but note that it was possibile in SGML DTDs). To handle such situations, the actual DTD-- processor is based non on XML-Schema, but rather on SchemaPath [MCV04], an extension of XML Schema that can handle co-constraints and conditional expressions in type assignments by exploiting XPath expressions. SchemaPath can perform tasks similar to Schematron [Jel05] (but without requiring users to learn a whole new language) and it allows designers to express a very large number of co-constraints on XML documents.

To handle context patterns, the conversion is actually performed from DTD-- into SchemaPath and, in turn, a SchemaPath validation is activated on the output. It should be noted, on the other hand, that a SchemaPath without conditional expressions is actually a plain XML-Schema document, and therefore DTD-- schemas that do not contain contexts are directly translated into plain XML-Schema

Consider an element `table` that cannot contain another `table` element, as required by the SGML DTD for Extreme Markup Conference. DTD-- allows users to express such constraint through the pattern `<table dtd:exclude="table">`. The following example shows the same constraint in SchemaPath syntax; see [MCV04] for more details.

Figure 26: Expressing a subtractive context with SchemaPath:

```
<xsd:element name="table">
    <xsd:alt cond=".//table" type="xsd:error"/>
    <xsd:alt                 type="typeOf_table"/>
</xsd:element>
```

An additive context adds some complexity to the conversion: since it cannot be directly converted into a single SchemaPath fragment the whole schema has to modified by spreading the additive element into the context: the contexual element has to be actually blended inside the content model of any element in the context (including the nested levels therein) and then some SchemaPath conditions have to be added in order to verify their correctness.

A more complex example can explain the issue: consider `contracts` as a table of `contract` records containing several elements including a `title`, which is an atom. Furthermore a signature (`sig`) can be contained in any descendant of the `contract` element, as shown in the DTD-- fragment shown in figure 27:

Figure 27: Expressing an additive context with SchemaPath:

```
 <contracts>
    <contract dtd:include='Sig'>
        <title>Title of the contract</title>
        ...
        <Sig>Your signature</Sig>
    </contract>
    <dtd:etc/>
</contracts>
```

Figure 28 shows the converted SchemaPath. The element `title` is transformed from an atom into a block; note also how SchemaPath allows us to express the unicity of the `title` within the `contract` element

since the latter is a record and, above all, the fact that `title` cannot contain `sig` if it is out of `contract`, according to the semantics of additive contexts. We could also express the unicity of the element `sig` in the context (in DTD-- this is specified with the attribute `dtd:include="!sig"`) by adding the following SchemaPath condition `<xsd:alt cond ="count(.//sig)>1" type="xsd:error/>"` in the definition of the `contract` type.

Figure 28: A complete example of conversion in SchemaPath:

```
<xsd:schema xmlns:xsd="http://www.cs.unibo.it/SchemaPath/1.0">
  <xsd:element name="contracts" />
    <xsd:complexType name="typeOf_contracts">
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element ref="contract"/>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="contract">
    <xsd:alt cond="count(title)>1" type="xsd:error"/>
    <xsd:alt                       type="typeOf_contract"/>
  </xsd:element>
  <xsd:complexType name="typeOf_contract">
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element ref="title"/>
      <xsd:element ref="Sig"/>
      ...
    </xsd:choice>
  </xsd:complexType>
  <xsd:element name="Sig" type="typeOf_Sig"/>
  <xsd:complexType mixed="true" name="typeOf_Sig"/>
  <xsd:element name="title">
    <xsd:alt cond="count(Sig)>0 and count(ancestor::contract)=0" type="xsd:error"/>
    <xsd:alt                                                    type="typeOf_title"/>
  </xsd:element>
  <xsd:complexType mixed="true" name="typeOf_title">
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element ref="Sig"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:schema>
```

§ Discussion and merits of DTD--

Markup languages are traditionally distinguished in *procedural* and *declarative* [CRD87]. Different proposals were brought forth to improve on such characterization and identify with more precision the features of all markup language subclasses; the main objective is to study the actual role and the most suitable context for each particular language, in order to improve and specialize its design. In [REN00], Allen Renear describes two dimensions in markup languages: *domain* and *mood*. The mood indicates wheter or not the language instructions impose new constraints on a document, and it can be classified as either "indicative" or "imperative". In an indicative context an element says that the tagged text fragment is a specific "object", intrinsically and independently from its mark-up; in an imperative context it says that the same fragment has to be modeled as that object. The "mood" dimension is independent and complementary to the domain characterization: either a language designed to describe the actual content of manuscripts (logical domain) or its final formatting (renditional domain) could be imperative or indicative (although, as Renear remarks, a "indicative renditional" language seems to make little sense).

Moving off Renear's ideas, Wendell Piez proposes a detailed map of markup languages from different perspectives [PIE01]. A language can be classified in terms of *time processing*, i.e., whether it looks backward (*restrospective languages*) or forward (*prospective languages*): a retrospective markup language is one that seeks to represent something that already exists, while a "prospective" language seeks to identify the document’s constituent parts as a preliminary step to further processing. Renear's and Piez's classifications partially overlap, so that, as Piez himself notes, the 'prospective' property corresponds to Renear’s imperative mood, and 'retrospective' to indicative. Piez further introduces a new category, the "explorative/mimetic" markup. This class includes all the languages that can handle structurally invalid markup and that are flexible in the modeling of legacy documents. This markup details those features of the text that are relevant to the encoder, without requiring to adapt the content structure to the schema language, nor having the schema language completely predict the document evolution.

Piez also discusses the level of validation appropriate to each class of languages. As expected, a *prescriptive* language requires strict validation because several and strong constraints have to be verified in order to be sure that the document will be processed exactly as it has been designed for; analogously, an *indicative* language is straightforwardly associated to a loose validation, since it is unnecessary to

express constraints on the future use and verification of documents and, above all, no strong assumption on its structure is really required. Markup languages placed into the extremes of the spectrum are quite rare to find: most of them assume a medium level of validation that can strictly verify some constraints but relax others and, more importantly, can look simultaneously backward and forward in time. The most common compromise (adopted by TEI, DocBooc, W3C specifications and so on) is using a prescriptive markup and anticipating the role and the future processing of the document.

In this middle gray area among the purely 'indicative' or 'imperative' languages, Piez proposes a further classification, the distinction between *proleptic* and *metaleptic* markup languages. A proleptic markup is any markup where the meaning of the tagging is intimately connected with the expectations for processing it. On the contrary, a metaleptic markup is a retrospective tagging for prospective purposes: thus, it works by saying something about the past, but in order to create new meaning out of it.

The classifications described so far help us to explain the nature of DTD-- with more details. DTD-- is not a specific XML dialect, but a validation language: it is not a single block in the Piez taxonomy, but an area covering different blocks which address different domains (according to Renear's classification). All the languages in that area are based on the same design principles. We could label DTD-- (or, better, any language derived from DTD--) as retrospective and metaleptic: retrospective because a DTD-- schema models existing data from a general perspective without imposing strong constraints; metaleptic because the simplified usage of patterns makes efficient and reliable the future management of the same data. It also worth to investigate whether or not DTD-- generates exploratory/mimetic languages: even if we cannot define DTD-- 'exploratory' as Piez meant (DTD-- is not adaptable to the exceptions and irregularities as required), we could consider it 'mimetic' because a DTD-- schema blurs completely with the instance of the document.

Another perspective can be useful: in [WIL02] Wilmott identifies two main categories of markup languages, whether they have to be interpreted by humans or automatically processed by machines,*human-based* and *machine-based* languages, and emphasizes their similarities and differences concluding that any language has to be designed bearing in mind what it will be really used for. Again, even according to Wilmott's classification DTD-- cannot be placed at the extreme of the spectrum, and we considered it 'human-based' because of the idea of noise minimization, readability, minimization of constructs that may appeal to human readers, but also 'machine-based' because of its ease of processing by future applications.

The nature of DTD-- justifies some of its design principles, and in particular the minimalistic approch in providing constructs and the use of nested wrappers. Several works in the literature anticipated such approach: in [USD02], Usdin claims that designers are interested in flexible semantics and not in flexible syntax, observing that, if different people might produce different, but correct, documents to express the same meaning, the risk of misinterpretation is increasing. DTD-- severly limits the choices in structures and composition of elements, while maintaining full descriptive in the definition of elements and attributes. Thus it agrees with Usdin's point about limiting the flexibility of syntax. What DTD-- proposes is not 'syntactic sugar', but rather a limited, well-defined and understandable set of meaningful choices: errors and misunderstandings are minimized by minimizing the choices.

In [USD02] Usdin also faces another interesting issue, i.e., the semantics of metamarkup languages such as SGML and XML: can the users infer something that authors had not implied? What a document says is always what the author really would say? In [RTW96], [COV98] and [SMQHR00] the authors remark that an XML document (but this also applies to SGML) need some extra information to be interpreted by humans, in particular names carefully selected by domain experts. Thus, of itself, XML is only partially suitable to interchange information among machines: while humans have a common ontology (the word "title" indicates something that is a "title"), machines do need a common and unambiguous semantics of the same tags. More recently, in [RDSMQH02] the authors discuss the importance of such a clear semantics describing the BECHAMEL Markup Semantics Project, a system for expressing semantic rules and meanings for markup languages based on PROLOG inferences and deductions ([DSRH03] and [DUB03]).The relation between BECHAMEL and the Semantic Web [THL01] is evident (after all, both of them want to transform information in something completely interpretable and processable by machines): but while the latter looks at these issues from a more general perspective, BECHAMEL focuses the attention on a specific domain.

While BECHAMEL and related works build a metastructure that can infer rules and semantics from the language, DTD-- has a different goal: proposing a restricted set of structures and substructures that already have intrinsic and unambiguous semantics. Yet, we are not concerned with the semantics of names and objects, but rather with the semantics of structures and the relations and dependencies among the elements.

The main idea is that any DTD-- document has a meaningful and structured nesting of elements (through patterns and wrappers), so that inferring their connections is both simple and unambiguous.

The identification of functional dependencies is a key step in the intepretation of data, regardless of the nature of these data. Traditionally two different perspective are identified,*document-analysis* and *data-analysis*: although they come from different disciplines and use different techniques, in [GLUMCG02] Glusko argues that they have a lot of unexpected common aspects and can be managed with a unified approach. In particular, he outlines a parallelism among the process of database normalization [DAT81] and the use of (sub-)structures in the documents to express dependencies and nesting. DTD-- share the same approach: relational databases use normalized tables to express hierarchies, as well as DTD-- use the wrappers to give structure and depth to the documents.

A DTD-- document is created by composition and nesting of a limited set of elements. In [ST01] Tompa defines such approach as 'Document assembly', i.e. the process of constructing a new document instance from fragments or components. By reducing the number of available items and composition rules, the whole process of documents' creation and management comes out simple and efficient. After all, the benefits of patterns in software engineering are well-known [GHJV94], and they are universally recognized as efficient, flexible and manageable solutions. But are these patterns sufficient to capture the entire information of some data structures? The point is that we do not need to capture the whole information, but rather the *relevant* one. In [GLUMCG02] Glusko notes that any document can be divided into three distinct components: *content*, *structure*, *presentation*. The content is the actual information carried by the document (what the document says?), the structure captures the organization of content (how is it placed in the document? what containers are used?) and the presentation says everything about formatting and rendering rules. All documents are thus heterogeneous regarding content, but share a common set of structures. The goal of DTD-- is just to capture only these structures in order to reduce the noise of unused and ambiguous constructs and help designers into providing simple, flexible but equally powerful schemas.

§ Conclusions

Designing a markup language requires, first of all, a deep study of what the language will be used for. While the goal of a perspective language is tagging documents to indicate how they will be processed in the future, the real goal of a descriptive one is capturing the meaning and the structure of the elements. In that perspective, a language has to be designed to gather the actual semantics of documents even if by relaxing constraints (remind that the goal is not imposing restrictions but describing things).

The basic justification for introducing the patterns is our feeling that validation languages offer much more choices of structures than necessary in these situations. Alternatives, for instance, represent a relevant structure only when we need to enforce a choice, and not when we are simply capturing and describing previously existing documents. Analogously, repeatable subgroups, when captured within wrappers, can arguably provide a clearer hierarchical and semantical structure than subgroups without loosing generality and widespread applicability.

In this paper we have discussed a set of patterns sufficient (with some exceptions and extensions) to create simple and well-structured schemas within a descriptive environment. Two design principles converge on our proposal: the minimality of the patterns and their ability of expressing relations and functional dependencies among the elements without any ambiguity. DTD-- is an instance-based validation language that help users to apply these patters in the designing documents. The possibility of deriving schemas directly from the instances, using existing XML tools and having few patterns make it easy to use and learn DTD--. Some tools were presented, that handle these schemas/documents by converting them into XML-Schema (or better SchemaPath). In the future, we plan to follow two complementary directions: from a theoretical perspective, we will study other possible applications and variants of patterns; from a practical one, we will test and improve these tools by implementing specialized editors and converters and, probably, a native DTD-- validator.

Bibliography

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

[COV98] Robin Cover, Cover Pages XML and Semantic Transparency. October 23, 1998. Revised November 24, 1998. **http://www.oasis-open.org/cover/xmlAndSemantics.html**.

[CRD87] Coombs, James H., Allen H. Renear, and Steven J. DeRose. (1987) 'Markup systems and the future of scholarly text processing.' *Communications of the ACM*, 30(11):933-947.

[DAT81] C. J. Date: *An Introduction to Database Systems*, 3rd Edition Addison-Wesley, 1981

[DSRH03] Dubin, D., Sperberg-McQueen, C. M., Renear, A., and Huitfeldt, C. A logic programming environment for document semantics and inference. *Literary and Linguistic Computing 18*, 2 (2003), 225–233. (This is a corrected version of an article that appeared in 18:1 pp. 39–47).

[DUB03] D. Dubin. Object mapping for markup semantics. In B. T Usdin, editor, *Proceedings of Extreme Markup Languages 2003*, Montreal, Quebec, August 2003.

[FGMV04] Fiorello D., N. Gessa, P. Marinelli, F. Vitali, "DTD++ 2.0: Adding support for co-constraints", *Proceedings of the Extreme Markup Languages 2004* **http://www.mulberrytech.com/Extreme/Proceedings/xslfo-pdf/2004/Vitali01/EML2004Vitali01.pdf**

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994*

[GLUMCG02] Glushko R.J., McGrath T. "Document Engineering for e-Business"*DocEng '02: Proceedings of the 2002 ACM Symposium on Document Engineering* McLean, Virginia, USA, ACM Press, 2002, 42-48. **http://doi.acm.org/10.1145/585058.585067**

[Jel05] Rick Jelliffe, Schematron 1.5**http://xml.ascc.net/schematron/**

[MCV04] Marinelli, P., Coen, C. S., Vitali, F. "SchemaPath, a Minimal Extension to XML Schema for Conditional Constraints". In *Proceedings International WWW Conference*, New York, USA, 2004. **http://www.www2004.org/proceedings/docs/1p164.pdf**

[PIE01] Piez W,"Beyond the 'descriptive vs. procedural' distinction", *Extreme Markup Languages 2001*, Montréal, August 2001.

[RDSMQH02] Allen Renear, D. Dubin and C.M. Sperberg-McQueen. Towards a Semantics for XML Markup. In *Proceedings of the 2002 ACM Symposium on Document Engineering*, 119-126, 2002. ACM Press.

[REN00] Renear, A. "The Descriptive/Procedural Distinction is Flawed". *Markup Languages: Theory and Practice 2*, 4 (2001), 411-420. Earlier version presented at Extreme Markup Languages 2000, Montréal, August 2000.

[RTW96] Raymond, D. R., Tompa, F. W., and Wood, D. >From data representation to data model—Meta-semantic issues in the evolution of SGML. *Computer Standards and Interfaces 18*, 1 (January 1996), 25–36.

[SMQHR00] Sperberg-McQueen, C. M., Huitfeldt, C., and Renear, A. Meaning and interpretation of markup. Markup Languages: *Theory and Practice 2*, 3 (2000), 215–234.

[ST01] Salminin A., Tompa F. "Requirements for XML document database systems". In EV Munson (Ed.), *Proceedings of the ACM Symposium on Document Engineering*, DocEng '01, pp. 85-94.

[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/**.

[THL01] Tim Berners-Lee, James Hendler, Ora Lassila, The Semantic Web, Scientific American, May 2001.

[USD02] Usdin B.T. "When ``It Doesn't Matter'' Means ``It Matters".",*Proceedings of Extreme Markup Language 2002* 2002 **http://www.mulberrytech.com/Extreme/Proceedings/xslfo-pdf/2002/Usdin01/EML2002Usdin01.pdf**

[VAND03] Eric van der Vlist, Examplotron **http://examplotron.org/**

[WIL02] Wilmott S. "The Dichotomy of Markup Languages."*Proceedings of Extreme Markup Language 2002* 2002 **http://www.mulberrytech.com/Extreme/Proceedings/xslfo-pdf/2002/Wilmott01/EML2002Wilmott01.pdf**

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.

Angelo Di Iorio
University of Bologna, Department of Computer Science
Mura A. Zamboni, 7
Bologna
Italy
diiorio@cs.unibo.it

Angelo Di Iorio holds a Laurea degree in Computer Science from the University of Bologna and has been a Ph.D. student since January 2004. His research interests include content management systems, web technologies and data formats

Daniele Gubellini
University of Bologna, Department of Computer Science
Mura A. Zamboni, 7
Bologna
Italy
gubellin@cs.unibo.it

Daniele Gubellini holds a Laurea degree in Computer Science from the University of Bologna.

Extreme Markup Languages 2005®
Montréal, Québec, August 1-5, 2005
This paper was formatted from XML source via XSL
by Mulberry Technologies, Inc.