Extreme Markup Languages 2004

Montréal, Québec
August 2-6, 2004

DTD++ 2.0: Adding support for co-constraints

Davide Fiorello
University of Bologna

Nicola Gessa
University of Bologna

Paolo Marinelli
University of Bologna

Fabio Vitali
University of Bologna

Abstract

In this paper we present an evolution of the DTD++ schema language for XML documents. The original DTD++ language provided support for a large and significant subset of XML Schema while maintaining a syntax closely resembling DTDs: thus the expressive power of XML Schema and the readability of DTDs were both supported in a modular architecture that could rely on a number of validating engine for XML schema.

DTD++ did not provide solution for another big disadvantage of XML Schema and all other grammar-based validation languages: the inability (or limited ability) to express requirements on the co-existence or co-absence of elements and attributes, and other kind of reciprocal constraints cumulatively called co-constraints.

The new version of DTD++, uninspiredly called DTD++ 2.0 and presented in this paper, provides a solution for specifying and verifying co-constraints on XML documents: by relying on one additional construct (conditional type assignment) and one additional predefined type (#ERROR) it is possible to express a large number of co-constraints in DTD++. Additionally, we describe an implementation of a validation engine for DTD++ 2.0 that maintains the main advantage of the previous version, in that it is but a pre-processor to any of a number of XML Schema validating engine, each of which can be used interchangeably.

A Conference of IDEAlliance
Extreme Markup 2004 Languages®

DTD++ 2.0: Adding support for co-constraints

Table of Contents

A Conference of IDEAlliance
Extreme Markup Languages® 2004

DTD++ 2.0: Adding support for co-constraints

Davide Fiorello, Nicola Gessa, Paolo Marinelli, and Fabio Vitali

§ Introduction

Validation languages are used to specify rules by which to verify the correctness of individual XML instances. There exist more than 15 different languages for validating XML documents, the most famous of which are DTDs, XML Schema, RELAX NG and Schematron (see for instance [COVER] for a list of such languages).

But validation languages, and in particular DTDs, have been used in the past for another purpose: to provide an easy and readable specification document for a language, meant for humans to learn the language and be able to either create tools for the language (i.e., a browser or an editor) or to create documents in the language. In many cases, DTDs with a few sparse comments was all that was available (and, often, needed) to use an XML vocabulary.

XML-based validation languages have changed this situation for the worse. Without doubts, XML Schema and RELAX NG specifications are harder to read by the human eyes than DTDs. The XML specific grammar is an obstacle, and the verboseness of the language creates specifications that are two or three times longer and much harder to read. On the other hand an XML-based grammar makes it easier to produce tools for managing, transforming and using schema specifications, while DTDs had to rely on specific parsers that were hardly integrated with subsequent tools in the document manipulation processes.

Therefore a seemingly irreducible situation is born: if you use DTDs, you have to rely on DTD parsers and cannot employ sophisticated tools for additional applications of the schema; on the other hand, if you use any of the XML-based schema languages, you loose much readability of the schema itself, and have to rely on tools (for instance, XML Spy) to make sense of the specification.

In 2003 we proposed DTD++, a possible solution for this situation: an extension of DTDs that took most of the structures and concepts of XML Schema, and that could be converted to an equivalent XML Schema automatically by the use of a simple parser. The expressive power of XML Schema (much greater than that of DTDs) could be combined with the readability of DTDs.

But XML Schema (and, consequently, DTD++) has an additional disadvantage: the lack of support for co-constraints. Co-constraints are rules limiting and constraining the reciprocal presence of elements and attributes, and subjecting the presence, absence or valid values of an element or attributes to the presence, absence and actual values of another element or attribute. Co-constraints are present in some validation language (such as Schematron and xlinkit), and absent in most of the others.

§ Validation languages and co-costraints

The scenario of validation languages presents two main approaches to schema definition:

- *grammar-based languages* , by which document engineers create a whole context-free grammar according to top-down production rules in a specified formalism. XML Schema [TBMM01] and RELAX NG [CM01], as well as DTDs themselves, fall into this category.
- *rule-based languages* , by which document engineers list a sequence of validity predicates that the XML documents must satisfy, providing either an open specification (all that is not forbidden is allowed) or a closed specification (all that is not allowed is forbidden). Schematron [Jel02] and xlinkit [NCEF02] belong to this category.

In general, grammar-based languages are well-suited to express, through grammatical expressions, constraints on the logical structure of XML documents (structural constraints), and many of them allow to accurately describe the set of legal values of text nodes and attributes. However, grammar-based languages are limited in the definition of co-constraints, for which neither DTDs nor XML Schema have any support at all, and RELAX NG only limitedly.

© 2004 Davide Fiorello, Nicola Gessa, Paolo Marinelli & Fabio Vitali

On the other hand, rule-based languages provide mechanisms to simply express co-constraints on XML documents. However, they cannot enforce structural constraints as straightforwardly, and they are limited (see for instance Schematron), in the definition of constraints on data values.

Whatever the approach, validation languages can be evaluated according to several parameters; in [WC01] a summary reporting a comparison of XML Schema, DTD, RELAX NG and Schematron was issued. This comparison considered five major categories to analyze strengths and weaknesses of these languages with:

- *Content models and datatypes*: how sophisticated are the rules for expressing constraints on structures (the number and order of elements and attributes) and data (allowed values and defaults).
- *Modularity*: how easily can complex schemas be organized in independent modules, and how flexible it is to reuse these modules.
- *Namespaces*: what kind of namespace support is provided, and what kind of restrictions can be placed on qualified XML elements and attributes.
- *Linking*: what kind of explicit relations can be expressed between elements and attributes of a same document (e.g., the ID/IDREF relation in DTDs).
- *Co-constraints*: whether it is possible to express constraints on elements and attributes based on the presence or values of other attributes and elements.

Although Schematron ends up as a clear winner in most of these categories, dominating alone the category of co-constraints, XML Schema is praised for the richness of its data types and the sophistication of its data model.

A further description of the state of the art of schema language, including DTD and XML Schema, for XML documents is provided in [Jel01]: it considers eight different schema languages and highlights the main drawbacks of XML Schema: complexity results in fact as the main problem in managing XML Schemas, whereas DTDs stand out or their terseness. Similar results are provided also by [LC00], that stresses the limited expressive power of DTDs, but describes it as the easiest schema language to learn, especially compared to XML Schema.

The need for co-costraints

The problem of co-constraints is important and it is heavily felt for in many user communities. Several domain-specific standard languages based on XML include lamentations (see for instance [FpMLReq]) that DTDs and XML Schema do not allow co-constraints: thus they provide these rules in natural language, and they recommend implementers to support the relevant rules directly in their software. Validation of incoming data is critical for e-business infrastructures, that require not only the use of a grammar and of a set of data types, but also the adoption of co-constraints, now implemented within application-specific modules [Ham02]. In [CJ02] it is highlighted that XML Schema provides no advance over DTD in managing attributes: there is no way to manage those complex constraints between attributes and elements, which are currently common in XML grammars.

Indeed, being unable to define any co-constraints, neither DTDs nor XML Schema can be used to impose simple and common requirements, such as mutual dependence (for instance, mutual exclusion among attributes, e.g., one of `A` and `B` attributes must be present, but not both), deep exclusions (as in SGML DTDs), structure-dependent structures (e.g., if the attribute `gratis` exists, the `<price>` element must be absent) and data-dependent structures (e.g., if `<city>` is ‘`Bologna`’ then `<country>` must be ‘`Italy`’).

Even some well-known W3C recommendations impose normative co-constraints expressing them in natural language throughout the plain text description of the language, but not in the formal schema specification.

For instance, in order to be correct, XML Schema documents must validate against the normative XML Schema for Schemas (in appendix A of [TBMM01]) and they also must satisfy a number of co-constraints that are only provided in natural language (throughout the official specification of the language in [TBMM01]) and that are not captured by the XML Schema for Schemas. For instance, it is required but not formally codified that within an element declaration only one of the `ref` and `name` attributes must be present, but not both (mutual exclusion).

Figure 1: An unsatisfactory DTD solution to the `zip` problem

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

Figure 2: An unsatisfactory DTD solution to the `salesTax` problem

```
<!ELEMENT orderForm (billToAddress, order, salesTax?, shippingInfo, paymentMethod) >
```

Analogously, strictly conforming XHTML documents must validate against the normative DTD specification provided in [ea00], that cannot specify the additional normative constraints described in natural language only in its appendix B. The most widely known constraint regards anchors, and states that `<a>` elements must not contain other `<a>` elements. Existing DTDs and XML Schemas for XHTML 1.0 can only enforce a weaker form of this prohibition: `<a>` cannot contain other `<a>` elements at the first level. Thus, in order to check the validity of an XHTML 1.0 document it is not sufficient to validate it against the normative DTD using one of the many available DTD processors, but additional code is needed to check whether any of the natural language constraints are violated.

Another, perhaps more meaningful example is provided in [Wal01]. In this document, which was given as input for the comparison between schema languages mentioned in the previous section, Norman Walsh describes three (purposefully contrived) test cases. These schemas represent three different cases that may occur designing XML documents. The third section of this document describes the "Order Form" document type. A readable set of requirements is provided for data type that "*uses address for both billing and shipping information*"; for instance, the zip element "*which must be either a five digit zip code or a nine digit "zip+4" code*". This definition cannot be supported by DTDs, as shown in fig. 1.

The same schema contains a `<billToAddress>` element: "*If the `<billToAddress>` is a US address and the state is not one of the following: AK, DE, HI, MT, NO, OR, OR WY, then the `<orderForm>` must also include a `<salesTax>` element immediately after the `<order>`*". This definition for the `<orderForm>` element requires the support for co-constraints, which are not expressable in the DTD's grammar (nor, for that matter, in XML Schema), as shown in fig. 2. DTD++ 2.0, the language discussed in this paper, presents a solution for both problems (and for all of the others that are presented in [Wal01]. We will show a DTD++ 2.0 solution for these problems in figg. 12 and 15. A complete solution of the "Order Form" problem using DTD++ 2.0 is provided in appendix to this paper.

Approaches to avoid coding in a standard programming language to perform additional co-constraint checks on XML documents, and to gain the advantages of both grammar-based and rule-based languages have been proposed in several works. Both [Rob02] and [Cos03] suggest to embed Schematron rules within XML Schema schemas ([Rob02] also proposes to embed Schematron rules with RELAX NG documents), obtaining a single specification that can impose both structural constraints (through XML Schema grammatical expressions) and co-constraints (through embedded Schematron rules). However such solution has several drawbacks caused by the low level of integration of the combined languages.

An alternative and interesting approach is proposed in [WRM03]. The concept is similar to the one explained in [Rob02] and [Cos03], i.e., enriching an XML Schema specification with validation rules. The difference is in the level of integration between the XML Schema specification and such rules. [WRM03] proposes in fact an extension of XML Schema, where both complex and simple type definitions can be enriched with one or more *rule definitions*. Basically, a rule definition is an XQuery [BS03] expression, extended with function calls. In order to be valid with respect to a type definition with validation rules, an element must satisfy both the validation rules and the normal constraints imposed by the type definition. Obviously, this proposal requires a validation engine that can recognize the new syntax and evaluate the validation rules.

Another interesting approach is represented by NRL (Namespace Routing Language) [Cla03], an XML schema language mainly designed to fit DSDL Part 4 (Selection of validation candidates). A NRL schema is basically a mapping from namespace URIs to schemas, which produces a partition of the instance document into sections. Each section is an XML tree where all elements belong to the same

Figure 3: a simple conditional element declaration

```
<xsd:element name="item">
  <xsd:alt cond="@itemType='CL'" type="ClothingItem"/>
  <xsd:alt cond="@itemType='NC'" type="NonClothingItem"/>
</xsd:element>
```

namespace, and is validated against its associated schema. The schemas used to validate sections are indentified by means of URIs, and they are not required to be written in the same validation language. Furthermore, NRL allows to associate more than one schema to the same namespace URI. For instance, a section could be validated against both an XML Schema specification and a Schematron schema. This feature can be used to verify grammatical constraints by means of an XML Schema schema, and co-constraints by means of a Schematron schema. Clearly, this solution requires the creation of two separated and completely independent schema documents, and has the same weaknesses, due to the low level of integration of the two schema languages, already raised for the solutions described in [Rob02] and [Cos03]. Moreover, since the validation against the XML Schema schema and the one against the Schematron schema are independent from each other, the Schematron schema cannot take advantage of the XML Schema validation process, in particular, it cannot take advantage of the generated PSVI.

§ SchemaPath

In [MSV04a] we proposed SchemaPath, a minimal and conservative extension to XML Schema to support co-constraints. SchemaPath extends XML Schema introducing conditional declarations, which assign types to elements and attributes based on the evaluation of XPath conditions. More precisely, a conditional element declaration lists a sequence of alternative type definitions, each associated with an XPath predicate and a priority. An element validates against a conditional declaration if it satisfies at least one XPath condition, and it is valid with respect to the type definition associated to the condition with the highest priority that is satisfied. An element of the instance document whose declaration is conditional is called a conditional element.

SchemaPath adds just one new construct to XML Schema (the `<xsd:alt>` element, for the expression of alternative type definitions) and one new built-in datatype, `xsd:error`, an unsatisfiable type used for the direct expression of negative rules, i.e., rules that need to be not satisfied for validity.

For instance, a simple conditional element declaration can be written to impose a co-constraint on an element (see fig. 3, a simplifiation of a constraints of the “Order Form” example): if the value of the `itemType` attribute is `'CL'`, then the `<item>` element must be of type `ClothingItem`; if the value of the `itemType` attribute is `'NC'`, then the `<item>` element must be of type `NonClothingItem`; otherwise, we have a validation error (i.e., the `itemType` attribute can only assume one of the two previous values).

SchemaPath is a conservative extension to XML Schema, so that every XML Schema specification is also a SchemaPath specification. This also means that, in order to obtain a rich SchemaPath, one can start writing a normal XML Schema and then just add those conditions that cannot be expressed in this language. For instance, it is an easy task to extend the (informative) XML Schema for XHTML 1.0 document [Mas02] to impose the prohibition on anchor elements previously mentioned (see fig. 4).

This simple conditional declaration assigns the type xsd:error to <a> elements whenever they contain other <a> elements, thus generating a validation error. In all other cases, the elements are assigned the same anonymous complex type, directly derived from the Inline type, that is used in the XML Schema for XHTML 1.0 [Mas02]. Note that the second alternative in the above conditional declaration has no explicit condition expressed. In this case, we assume that the condition is the XPath expression `true()`, which is always true and allows for the specification of default type assignments.

Similarly, it is possible to extend the XML Schema for Schemas [TBMM01] to impose co-constraints on element declarations (Figure 3).

The complex type "element" is, slightly simplified, the same complex type defined in the original XML Schema. The global conditional declaration for `<element>` applies to both global and local element declarations, and it is interpreted as follows:

Figure 4: prohibition on anchor elements

```
<xsd:element name=”a”>
  <xsd:alt cond=”.//a” type=”xsd:error”/>
  <xsd:alt>
    <xsd:complexType mixed="true">
      <xsd:extension base="Inline">
        <!-- attribute declarations -->
      </xsd:extension>
    </xsd:complexType>
  </xsd:alt>
</xsd:element>
```

Figure 5: imposing co-constraints on the declaration of elements

```
<xsd:complexType name="element">
  <xsd:sequence>
    <xsd:choice minOccurs="0">
      <xsd:element name="simpleType"  type="xsd:localSimpleType"/>
      <xsd:element name="complexType" type="xsd:localComplexType"/>
    </xsd:choice>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:NCName"/>
  <xsd:attribute name="ref" type="xsd:QName"/>
  <xsd:attribute name="type" type="xsd:QName"/>
</xsd:complexType>
<xsd:element name="element">
  <xsd:alt cond="@name and @ref"
        type="xsd:error" priority="4"/>
  <xsd:alt cond="(@type or @ref) and (xsd:simpleType or xsd:complexType)"
        type="xsd:error" priority="3"/>
  <xsd:alt cond="../xsd:schema and @ref"
        type="xsd:error" priority="2"/>
  <xsd:alt cond="not(@ref) and not(@name)"
        type="xsd:error" priority="1"/>
  <xsd:alt type="element" priority="0"/>
</xsd:element>
```

- The first alternative assigns the `xsd:error` type whenever both the `name` and `ref` attributes are present.
- The second alternative assigns the `xsd:error` type whenever both the `ref` attribute and a type definition (either named or anonymous) are specified.
- The third alternative assigns the `xsd:error` type whenever the element declaration is global and a `ref` attribute is present.
- The fourth alternative assigns the `xsd:error` type whenever the `ref` and the `name` attributes are neither present.
- The fifth and last alternative assigns the element type in all other cases (i.e., whenever no co-constraint is violated).

Each alternative of a conditional declaration has a priority. An error is generated if a conditional element matches more than one condition with the same priority. For this reason, the above declaration makes use of the priority attribute to assign different priorities to each alternative, thus avoiding possible conflicts. Note that the last alternative, used to express the default type assignment, has no explicit priority. This is because a default alternative (i.e., one whose condition is `true()`) is automatically assigned a priority lower than all other alternatives within the same declaration.

A last example on conditional declarations shows how to enforce a co-constraint on XSLT documents. The XSLT recommendation [Cla99] states that within a `<template>` element the match attribute is required unless the name attribute is present. While XML Schema is not able to formalize such a requirement, a reasonable SchemaPath element declaration for `<template>` is shown in Figure 4.

Figure 6: a Schema Path declaration for <template> elements

```
<xsd:element name="template">
  <xsd:alt cond="not(@match) and not(@name)" type="xsd:error"/>
  <xsd:alt type="xsl:templateType"/>
</xsd:element>

<xsd:complexType name="templateType">
  <xsd:sequence>
    <xsd:group ref="xsl:templateContent"/>
  </xsd:sequence>
  <xsd:attribute name="match" type="xsl:patternType"/>
  <xsd:attribute name="name" type="xsd:NCName"/>
</xsd:complexType>
```

The first alternative throws a validation error (i.e, it assigns the `xsd:error type`) when both `name` and `match` attributes are missing, and the second condition assigns the `templateType` type in all other cases.

As shown by the above examples, conditional type attributions can overcome one of the most evident limitations of XML Schema (and of grammar-based schema languages in general), the support for co-constraints. Indeed, a conditional declaration express a dependency between the types assigned to an element (and thus between its content) and the value or presence of elements and attributes involved in a XPath condition. At the same time, SchemaPath inherites all the features of XML Schema (such as the derivation mechanism for simple and complex types, and the Post Schema Validation Infoset). For more information on SchemaPath and its implementation see [MSV04b].

§ DTD++ 1.0

DSDL (ISO/IEC 19757, [DSDL]) is the on-going effort of ISO/IEC to create “a framework within which multiple validation tasks can be applied to an XML document in order to achieve more complete validation results than just the application of a single technology”. Part 9 of this effort is targeted at developing a “datatype- and namespace-aware DTD” extension. Today the development of this part is still empty. In 2003 ([VAG03]) we have introduced and discussed our proposal for an extension to DTDs that could fill in this part of the DSDL effort.

The extension, called DTD++, is aimed at extending the syntax of DTD to cover most of the concepts and structures of XML Schema. In particular, during the development of the DTD++ language, we have considered the following design requirements:

- The language need to be a real superset of DTD so that every legal DTD is legal in DTD++
- The extension introduced have to be non-intrusive into the well-known DTD syntax
- We want to use as much as possible the existing constructs of DTD, and add the smallest possible set of new syntactical constructs to the language

Another proposal [WS03] also describes a compact text-based syntax (called XSCS) that claims to reuse the DTD syntactic constructs to simplify XML Schema creation and managements. Verbosity and complexity are pointed out here as well as criticisms in working with XML Schema.

The extended syntax of DTD++ relies on three clear extension models:

- Parameter entities in DTDs and named structures in XML Schema (simple types, complex types, attributes, etc.) perform the same task of providing a shared access to the same constructs, that are defined once and used many times. Parameter entities, of course, are a much less organized and

Figure 7: how parameter entities in DTDs and types in XML Schema are used for the same purpose

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

<xsd:complexType name="inline" mixed="true">
  <xsd:choice maxOccurs="unbounded">
    <xsd:element ref="b"/>
    <xsd:element ref="i"/>
    <xsd:element ref="u"/>
  </xsd:choice>
</xsd:complexType>

<xsd:element name="p" type="inline"/>
<xsd:element name="b" type="inline"/>
<xsd:element name="i" type="inline"/>
<xsd:element name="u" type="inline"/>
```

Figure 8: how content models in DTDs and anonymous types in XML Schema are used for the same purpose

```
<!ELEMENT html (head, body)>
<xsd:element name="html">
  <xsd:complexType>
    <xsd:sequence>
    <xsd:element ref="head"/>
    <xsd:element ref="body"/>
  </xsd:complexType>
</xsd:element>
```

controlled mechanism than named structures, but good form suggests their use in a similar fashion to named structures.

- Anonymous structures in XML Schema and content models in DTDs perform the same task of providing simple and immediate definition of structures that do not need to be reused.

- Content models in DTDs employ a syntax vaguely reminiscent of regular expression to control the structure and repetition of its elements.

So the syntax of DTD++ includes a number of structures typical of XML Schema, while at the same time inheriting the syntax philosophy of DTDs. In this sense we kept the DTD validation paradigm, while at the same time we enriched it by importing those XML Schema constructs that were compatible with it.

The first version of DTD++ provides support for namespaces, predefined simple datatypes, facets, occurrence constraints, user-defined simple and complex types, 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 that will be briefly reassumed in the following section. In the next subsections we review those extensions, that were already presented in [VAG03].

Namespace declaration

DTD++ provides a proper syntax to support namespace declarations as a means to specify a context for rules and constraints. Thus, DTD++ only defines namespaces as target namespaces, for which rules are expressed by the specification.

Target namespaces are specified using the new construct TARGETNS. 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 7 shows the definition of two namespaces and two elements.

Figure 9: 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)>
```

Figure 10: Simple type definition

```
<!-- anonymous Simple Type -->
<!ELEMENT position (#INTEGER[0,99999]) >

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

Simple types

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. On the contrary, DTDs have only one datatype for character elements (#PCDATA) and a handful of datatypes for attributes (CDATA, enumerations, IDs, IDREFs, and so on). To solve this gap, we have provided DTD++ with all the predefined types of XML Schema, with no distinction between attribute and elements. Predefined simple types are defined in the same vein as #PCDATA: they are called #STRING, #INTEGER, #DATE, #FLOAT, and so on.

Starting from predefined simple types, it is possible to create new types by deriving existing ones by restriction, union and list. The new types can be defined anonymously or with a name.

As mentioned, DTD++ introduces user-defined types by 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 datatypes are introduced with a new kind of parameter entity (using the # character instead of % to avoid syntactical confusions). Figure 8 shows the definitions of an anonymous and a named simple type.

Restriction, facets, list and union

XML Schema defines a number of constraints (called *facets*) that can be applied to a base simple type in order to create by restriction a new type. Among them there are for example `minInclusive`, `maxInclusive`, `length`.

DTD++ introduces facets to simple data types using a syntax that is similar to the regular expression syntax used in other parts of the syntax. List types are defined by specifying a “+” character after the base type name. Union types are defined by specifying the “|” character to separate the individual type names of the union. Some examples in using facets, list and union in DTD++ are show in Figure 9.

As a supplementary example, figure 10 shows the DTD++ solution of the zip problem if the "order form" document type.

Complex types

Complex types in XML Schema constrain both the content and the attributes of the element being defined. XML Schema therefore uses one construct (<xsd:complexType>) for constraints on both elements and attributes. DTD, on the other hand, uses two different constructs (`<!ELEMENT>` and `<!ATTLIST>`) for the same purpose.

DTD++ uses an intermediate approach: anonymous complex types in DTD++ are ELEMENT and ATTLIST constructs absolutely equivalent to the corresponding ones in DTDs. Named complex types, on the other hand, use yet another version of the parameter entities of DTD that is identified by the

Figure 11: Use of facets (with their XML Schema translations), lists and unions in DTD++ syntax

```
DTD++ syntax        Equivalent XML Schema syntax

#INTEGER[0,100]       <xsd:minInclusive value="0"/>
              <xsd:maxInclusive value="100"/>
#STRING{10,}        <xsd:minLength value="10"/>
(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}"/>

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

Figure 12: Definition of the zip element in DTD++

```
<!ELEMENT zip (#NMTOKEN/\d{5}|\d{5}-\d{4}/) >
```

Figure 13: Definition of a complex types in DTD++

```
<!ENTITY @ USAddress "(name,street,city)" "country (#NMTOKEN) 'US'">
<!ELEMENT PurchaseOrder (shipTo,billTo,comment[0,5],items)>
```

character "@" and that contains two separate values for the specification of element content and of attribute content. Constraints on cardinality of elements can be specified similarly to facets in simple types. See figure 10 for an example.

Similarly to XML Schema, DTD++ provides support for further extensions to complex types. Particularly:

- It improves the expressiveness of the mixed content model, allowing constraints and structures on the elements, syntactically allowing any structured content model near to a PCDATA specification.
- It reintroduces the ALL content model present in SGML DTDs (the "&" operator), that disappeared in XML DTDs but were reintroduced in XML Schema.
- It adds a way to specify constraints on occurrence and namespaces on the ANY content model, with a syntax similar to facets in simple types.

Figure 11 shows some definition for mixed, all and any content model.

§ DTD++ 2.0: supporting co-constraints

The main difference of DTD++ 2.0, with respect to version 1.0, is the support for co-constraints on elements and attributes. In order to support co-constraints we have modified the DTD++ syntax introducing two new constructs, that can be easily translated into the SchemaPath conditional type attribution and the predefined xsd:error simple type. Thus DTD++ 2.0 is designed to be semantically equivalent to a significant subset of SchemaPath, rather than of XML Schema, adding exactly those new constructs.

In designing the new DTD++ constructs we followed the same philosophy outlined in the introduction on DTD++: the new syntax have to be as less intrusive as possible, making full use of the existing DTD constructs. In our mind the introduction of a new element, or the use of special characters, would have

Figure 14: Definition for mixed, ALL and ANY content model

```
<!--  a classical mixed content model -->
<!ELEMENT salutation (#PCDATA | name | address)*>

<!--  an extended mixed content model -->
<!ELEMENT letterBody (#PCDATA (salutation?,quantity,productName,shipDate[0,1]))>

<!--  an ALL content model -->
<!ENTITY @ PurchaseOrderType  "(shipTo & billTo & comment?)">

<!--  a ANY content model -->
<!ELEMENT chapter ANY[1,]{http://www.bar.it}>
```

Figure 15: DTD++ solution to the salesTax issue in the "Order Form" example

```
<!ELEMENT orderForm "billToAddress[(country='US' or not(country)) and
                           (state!='AK' and
                            state!='DE' and
                            state!='HI' and
                            state!='MT' and
                            state!='NO' and
                            state!='OR' and
                            state!='WY')] and not(salesTax)" (#ERROR) >
<!ELEMENT orderForm "" (@OrderForm;) >
```

worsened the readability of the document, so we preferred to fiddle with the `<!ELEMENT>` construct to support conditional type definitions.

Thus, the definition of an element with a conditional type assignment is repeated for any condition imposed on the element, each with its proper type assigned (both anonymous - content model - or named - parameter entity).

Therefore the presence, within an element declaration, of an XPath expression, identifies the element as a conditional element. An element declaration without an XPath expression is a non-conditional element, while an empty string as the XPath expression is equivalent to the default condition (a `true()` condition). Moreover it is possible to explicitly set a priority on the condition by suffixing it with a colon and the numerical value of the condition. The predefined type `xsd:error` in SchemaPath has a corresponding `#ERROR` type in DTD++ 2.0. In the following section we will show some examples to clarify the use of conditional types in DTD++.

Some Examples in DTD++ 2.0

In this section we provide the corresponding DTD++ 2.0 equivalent structures to express the co-constraints described in Section 3, thus showing the expressiveness and usefulness of our proposal. Such solutions are simple translations of the SchemaPath conditional declarations given in Section 3, and thus they also show the advantages of the DTD-like syntax in terms of readability.

No salesTax in some USA states in the "Order Form" document type

If the state of the billToAddress is one of "Ak, DE, HI, MT, NO, OR, WY" then a sales tax must be applied. In this case we raise an error if there is no salesTax element whenever a US address has one of the listed values for the state element.

No nesting of <a> elements in XHTML 1.0

In order to enforce the prohibition of XHTML 1.0 `<a>` elements within other `<a>` elements, we write a type declaration, tempType, equivalent to the anonymous type defined within the corresponding SchemaPath solution (see figure 12). We then use two `<!ELEMENT>` constructs to express the two alternative type assignments specified within the SchemaPath conditional declaration. The first one assigns the `#ERROR` type when at least one `<a>` element is present within the declared `<a>` element, while the latter assigns the declared tempType type in all other cases.

Figure 16: DTD++ solution to avoid nesting of <a> elements in XHTML 1.0

```
<!ENTITY @ tempType "@inline;" "%attributes;">
<!ELEMENT a ".//a" (#ERROR)>
<!ELEMENT a "" (@tempType)>
```

Figure 17: imposing uncovered co-constraints on element declaration with DTD++

```
<!ELEMENT simpleType (@localSimpleType;)>
<!ELEMENT complexType (@localComplexType;)>
<!ENTITY @ element "(simpleType|complexType)"
      "name (#NCName;)
       ref  (#QName;)
       type (#QName;)">
<!ELEMENT element "@name and @ref":4 (#ERROR)>
<!ELEMENT element "(@type or @ref) and (xsd:simpleType or xsd:complexType)":3 (#ERROR)>
<!ELEMENT element "../xsd:schema and @ref":2 (#ERROR)>
<!ELEMENT element "not(@ref) and not(@name)":1 (#ERROR)>
<!ELEMENT element "":0 (@element;)>
```

Figure 18: DTD++ declaration for <template> element

```
<!ELEMENT template "not(@match) and not(@name)" (#ERROR) >
<!ELEMENT template "" (@templateType;) >
<!ENTITY @ templateType  "%templateContent;"
      "match (#patternType;) name(#NCName;)">
```

Co-constraints on XML Schema element declarations

The DTD++ 2.0 solution to express the co-constraints on XML Schema `<element>` elements enforced by the SchemaPath conditional declaration provided in Section 2 is shown in figure 13:

This example clearly shows the improvement of DTD++ 2.0 over SchemaPath in terms of readability. Indeed, while the SchemaPath solution consists of twenty-six lines of code, the corresponding DTD++ 2.0 solution consists of just twelve.

Named template in XSLT

The SchemaPath conditional declaration enforcing the presence of at least one among the `match` and `name` attributes within an XSLT `<template>` element is translated into the equivalent DTD++ 2.0 schema snippet shown in figure 14:

The two `<!ELEMENT>` constructs have the same meaning of the alternatives within the SchemaPath declaration, while the `<!ENTITY>` construct define the complex type for the `<template>` elements, and it is equivalent to the type definition provided in the corresponding SchemaPath solution.

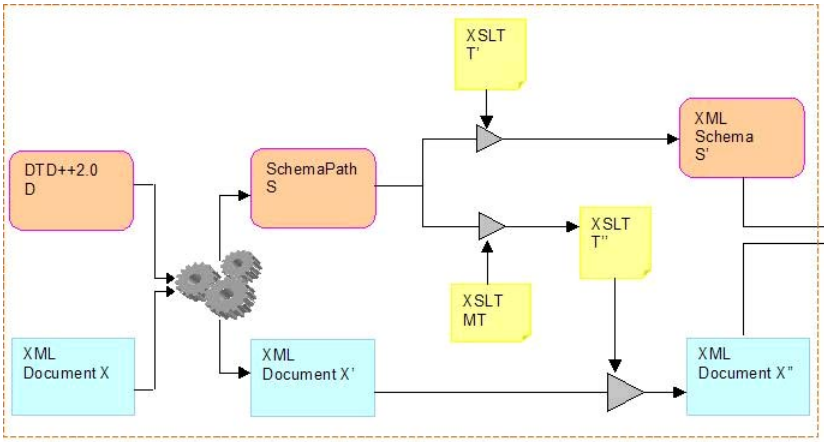
Implementing a validating engine for DTD++ 2.0 schemas

Adding support for the definition of co-constraints in DTD++ by adopting the same constructs defined in SchemaPath for conditional declaration has a double advantage. First of all, we can rely on the SchemaPath proposal to demonstrate the usefulness, in a grammar-based language, of conditional type assignments. The usage of XPath predicates allows us to *conservatively* extend the grammar-based language providing the support for a large class of co-constraints.

Second, it allows us to implemented our DTD++ 2.0 validating engine as a pre-processor (*PreValidator for DTD++ 2.0*) to a SchemaPath validating engine. The SchemaPath validator, described in [MSV04a],

is in itself a simple pre-processor based on XSLT to a standard XML Schema validation. Thus, the overall DTD++ 2.0 validation process architecture is depicted in Fig. 15.

Figure 19: Overall DTD++ 2.0 Validation process



It is composed of two separate steps: the first is a DTD++ 2.0 parser that creates a SchemaPath document S out of the DTD++ 2.0 document D, and resolves all the document entities that are present in the XML document X by creating a new document X' (in the subsequent steps there cannot be entities in the XML document). The SchemaPath document, by construction of the language, validates the new XML document if and only if a corresponding XML Schema document validates a properly transformed new XML document X". Thus the SchemaPath validator generates through the XSLT stylesheet T' an XML Schema S', that will be used for the final validation. At the same time, another XSLT stylesheet T" is used to generate the final XML document X". Only, the T" stylesheet is not general, but depends on the actual SchemaPath being used. Thus T" is created automatically by an XSLT meta-stylesheet MT. At the end, the XML Schema S' and the XML document X" are sent to a standard XML Schema validating engine and the result is sent back to the initial application. Since all steps are semantically equivalent, we can rest assured that the XML document X is valid according to the DTD++ 2.0 D **if and only if** the XML document X' is valid according to the XML Schema S'.

Such an implementation is highly portable, and does not require any adjustment in external applications. The application transforming the DTD++ 2.0 instance D and the XML instance document X has been wholly written in Java, modifying a preceding version of PreValidator for DTD++ discussed in [VAG03].The SchemaPath validation process and the implementation of the SchemaPath processor are discussed in [MSV04b].

It is important to stress that any XML Schema validating engine can be used to perform the final validation process on the resulting XML Schema and XML documents produced by the SchemaPath pre-processor. The result is equivalent, minus the conversion of the actual error messages, which are different in each implementation. We have tested our implementation with both the Xerces XML parser and with the MSXML4.0 parser. In order to test the usefulness and expressiveness of DTD++ 2.0, a web interface is publicly accessible at the URL **http://tesi.fabio.web.cs.unibo.it/dpp**.

§ Conclusions

This paper stresses the fact that the are two main purposes in providing a schema for an XML vocabulary: on the one hand, verification of the correctness of individual XML instances according to the required grammar (the validation process).

On the other hand, a schema is a useful tool for learning the XML vocabulary itself and deducing the format of the XML document to be produced. For years authors and programmers have received scantily commented DTDs for both SGML and XML, and have used them to learn the language and to create

tools and deliver valid documents. Experience proves that DTDs can be read and understood with the aid of no additional browsing tool.

On the other hand, the limited expressive power of DTDs, and the need for a special processor to read and parse them, has led developers to create a number of different schema languages, each different and each much less readable and understandable than DTDs.

The original reason to create DTD++, as mentioned in [VAG03], was to create a schema language that could benefit from both the expressive power of XML Schema and the readability of DTDs. Yet, the expressive power of XML Schema, as noted by many authors, is not adequate for many important validating tasks, the most important of which are the verification of co-constraints.

SchemaPath [MSV04a] has been created as a solution to extend the existing and powerful XML Schema language to handle conditional type assignments, which are a simple and easily-learnt mechanism to express co-constraints.

DTD++ 2.0, the language described in this paper, is thus our proposal for joining the two experiences of SchemaPath and DTD++, allowing us to create a validating language that is suspiciously similar to DTDs, is semantically equivalent to a large and significant subset of XML Schema, **and** provides support for co-constraints.

Finally, the modular architecture of our validating engine, and the reliance on any of a number of existing XML Schema parsers, guarantees the easy deployment of any DTD++ solution, and the possibility of caching the generated documents at any step of the process guarantees a controllable increase of efficiency in any real life usage of the language. We think that DTD++ 2.0 could be a reasonable and interesting proposal to fill the currently empty part 9 of the DSDL ISO/IEC effort.

As previously mentioned, our DTD++ 2.0 implementation can be downloaded and tested online, by accessing the URL **http://tesi.fabio.web.cs.unibo.it/dpp** .

§ Appendix

In this appendix we provide first a solution for the "Order Form" problem [Wal01] using a plain DTD, and then using the full syntax of our own DTD++ 2.0 language.

Figure 20: A standard DTD for order form documents.

```
<!ENTITY % Address "(street+, city, (state | stateOrProvince?), (zip | postalCode?),
country?)" >
<!ELEMENT street (#PCDATA) >
<!ELEMENT city (#PCDATA) >
<!ELEMENT state (#PCDATA) >
<!ELEMENT stateOrProvince (#PCDATA) >
<!ELEMENT zip (#PCDATA) >
<!ELEMENT postalCode (#PCDATA) >
<!ELEMENT country (#PCDATA) >
<!ELEMENT orderForm (billToAddress, order, salesTax?, shippingInfo, paymentMethod) >
<!ATTLIST orderForm    xmlns CDATA #FIXED "urn:x=xmlns:example:orderForm" >
<!ELEMENT billToAddress %Address; >
<!ELEMENT order (item)+ >
<!ELEMENT item (itemNumber, description, quantity?, unitPrice, (size, color,
alternateColor, monogram?)?) >
<!ELEMENT itemNumber (#PCDATA) >
<!ELEMENT description ANY >
<!ELEMENT quantity (#PCDATA) >
<!ELEMENT unitPrice (#PCDATA) >
<!ELEMENT size (#PCDATA) >
<!ELEMENT color (#PCDATA) >
<!ELEMENT alternateColor (#PCDATA) >
<!ELEMENT monogram (#PCDATA) >
<!ELEMENT shippingInfo (shipToAddress, shipBy) >
<!ELEMENT shipToAddress %Address; >
<!ELEMENT shipBy (#PCDATA) >
<!ATTLIST shipBy   shippingCost NMTOKEN #REQUIRED   rush (none | 3day | 2day |
overnight) "none" >
<!ELEMENT paymentMethod (creditCard | checkOrMoneyOrder) >
<!ATTLIST paymentMethod   amount NMTOKEN #REQUIRED >
<!ELEMENT creditCard (type?, number, expiration) >
<!ATTLIST creditCard   type (Amex | Visa | MasterCard) #IMPLIED >
<!ELEMENT type (#PCDATA)>
<!ELEMENT number (#PCDATA)>
<!ELEMENT expiration (#PCDATA)>
```

```
<!ELEMENT checkOrMoneyOrder EMPTY >
<!ELEMENT salesTax (#PCDATA) >
```

This DTD cannot formalize a large number of the requirements described in natural language. These requirements concern the support for namespaces, the logical structure of order form documents, the set of legal values of text nodes, and the dependence of some elements on the presence or the value of other elements or attributes.

For instance, the <description> element should contain elements from any namespace other than the order form namespace, while our DTD allows it to contain elements from any namespace, including the order form namespace.

Furthermore, there are elements, such as <state>, <zip>, <itemNumber>, <quantity>, and <uniPrice>, whose content is defined simply as #PCDATA, and thus are allowed to contain any text string, while the requirement indicates much more restricted sets of data values.

Or even, some requirements are not even formalized in the DTD. For instance, the presence of the <salesTax> element, as discussed in the main text of the paper, or the fact that the content of the <item> element depends on the value of its child <itemNumber>. All of those constraints are simply ignored in the DTD.

Finally, DTDs are not namespace-aware, and thus the DTD for order form documents has to resort to the expedient of fixing the prefix associated to the order form namespace.

On the other hand, DTD++ 2.0 provides a number of constructs that are exactly useful in these situations. In fig. 21 we provide a DTD++ 2.0 schema for order form documents, which satisfies all the requirements of [Wal01], even those not discussed in the previous sentences.

Figure 21

```
<!TARGETNS "urn:x-xmlns:example:orderForm" >
<!ENTITY # USState "(#NMTOKEN(AK|AL|AZ|AR|CA|CO|CT|DC|DE|FL|GA|ID|HI|IL|IN|IA|KS|KY|
LA|ME|MD|MA|MI|MN|MS|MT|MO|NE|NO|NV|NH|NJ|NM|NY|NC|ND|OH|OK|OR|
PA|RI|SC|SD|TN|TX|UT|VT|VA|WA|WV|WI|WY))" >
<!ENTITY # Price "(#DECIMAL/\d*\.\d{2}/)" >
<!ENTITY # CreditCardType "(#NMTOKEN(Amex|Visa|Mastercard))" >
<!ENTITY # AmexNumber "(#NMTOKEN/\d{15}/)" >
<!ENTITY # VisaNumber "(#NMTOKEN/\d{13}|\d{16}/)" >
<!ENTITY # MastercardNumber "(#NMTOKEN/\d{16}/)" >
<!ENTITY # ShipByContent "(#NMTOKEN(USPS|FedEx|UPS|DHL))" >
<!ENTITY # Rush "(#NMTOKEN(none|3day|2day|overnight))">
<!ENTITY @ USAddress "(street+, city, state, zip, country?)">
<!ENTITY @ InternationalAddress "(street+, city, stateOrProvince?, postalCode?, country)" >
<!ENTITY @ OrderForm "(billToAddress, order, salesTax?, shippingInfo, paymentMethod)" >
<!ENTITY @ NonClothingItem "(itemNumber, description, quantity?, unitPrice)" >
<!ENTITY @ ClothingItem "@NonClothingItem;,(size, color, alternateColor, monogram?)" >
<!ENTITY @ ShipBy "#ShipByContent;" "shippingCost (#Price;) #REQUIRED
rush (#Rush;) 'none'">
<!ENTITY @ CreditCard "(type?, number, expiration)" "type (#CreditCardType;)">
<!ELEMENT street (#STRING) >
<!ELEMENT city (#STRING) >
<!ELEMENT state (#USState;) >
<!ELEMENT stateOrProvince (#STRING) >
<!ELEMENT zip (#NMTOKEN/\d{5}|\d{5}-\d{4}/) >
<!ELEMENT postalCode (#NMTOKEN) >
<!ELEMENT country (#STRING) >
<!ELEMENT orderForm "billToAddress[(country='US' or not(country)) and (state!='AK' and
state!='DE' and state!='HI' and state!='MT' and state!='NO' and state!='OR' and
state!='WY')] and not(salesTax)" (#ERROR) >
<!ELEMENT orderForm "" (@OrderForm;) >
<!ELEMENT billToAddress "country='US' or not(country)" (@USAddress;) >
<!ELEMENT billToAddress "" (@InternationalAddress;) >
<!ELEMENT order (item)+ >
<!ELEMENT item "starts-with(itemNumber, 'NC')" (@NonClothingItem;) >
<!ELEMENT item "starts-with(itemNumber, 'CL')" (@ClothingItem;) >
<!ELEMENT itemNumber (#NMTOKEN/(CL|NC)-\d{4}/) >
<!ELEMENT description (ANY{##other}) >
<!ELEMENT quantity (#POSINTEGER) >
<!ELEMENT unitPrice (#Price;) >
<!ELEMENT size (#NMTOKEN(S|M|L|XL|LT|XSLT)) >
```

```
<!ELEMENT color (#NMTOKEN) >
<!ELEMENT alternateColor "text()=../color/text()" (#ERROR) >
<!ELEMENT alternateColor "" (#NMTOKEN) >
<!ELEMENT monogram (#NMTOKEN/[A-Z]{1,3}/) >
<!ELEMENT shippingInfo (shipToAddress, shipBy) >
<!ELEMENT shipToAddress "country='US' or not(country)" (@USAddress;) >
<!ELEMENT shipToAddress "" (@InternationalAddress;) >
<!ELEMENT shipBy "shipToAddress[country and country!='US'] and
(@rush='overnight')" (#ERROR) >
<!ELEMENT shipBy "" (@ShipBy;) >
<!ELEMENT paymentMethod (creditCard | checkOrMoneyOrder) >
<!ATTLIST paymentMethod   amount (#Price;) #REQUIRED >
<!ELEMENT creditCard "(@type and type) or (not(@type) and not(type))" (#ERROR) >
<!ELEMENT creditCard "" (@CreditCard;) >
<!ELEMENT type (#CrediCardType;)>
<!ELEMENT number "(not(../type) and ../@type='Amex') or (../type='Amex' and
not(../@type))" (#AmexNumber;) >
<!ELEMENT number "(not(../type) and ../@type='Visa') or (../type='Visa' and
not(../@type))" (#VisaNumber;) >
<!ELEMENT number "(not(../type) and ../@type='Mastercard') or (../type='Mastercard'
and not(../@type))" (#MastercardNumber;) >
<!ELEMENT expiration (#GYEARMONTH) >
<!ELEMENT checkOrMoneyOrder EMPTY>
<!ELEMENT salesTax (#Price;) >
```

DTD++ 2.0 is namespace-aware, and thus the trick of fixing the namespace prefix used in the standard DTD is abandoned in favour of the more robust TARGETNS construct.

Elements such as <state>, <zip>, <itemNumber>, <quantity>, and <uniPrice> are assigned simple types that precisely restrict their sets of legal data values, according to the specification [Wal01].

Moreover, this DTD++ 2.0 schema imposes those co-constraints that were completely left out by the first DTD. For instance, the mutual exclusion among the <type> element and the `type` attribute is forced through a conditional declaration for the <creditCard> element. The <number> element, representing a credit card number, is assigned a different type based on the credit card type (Amex, Visa, or Mastercard). Also, overnight shipping to international addresses is forbidden.

All of these improvements are obtained at a low price. Indeed, the standard DTD for order form documents consists of 53 lines of code, while the DTD++ 2.0 schema has 72.

Bibliography

[BS03] Boan, Scott, et alt. *XQuery 1.0: An XML Query Language*. W3C Working Draft. November 2003. **http://www.w3.org/TR/xquery/**.

[CJ02] Clark, James, Jelliffe, Rick. James Clark and Rick Jelliffe on RELAX NG and W3C XML Schema. June 2002. **http://xml.coverpages.org/Clark- Jelliffe-Schemas20020604.html**.

[Cla03] Clark, James. *Namespace Routing Language (NRL)*. June 2003. (2003-06-13). **http://www.thaiopensource.com/relaxng/nrl.html**.

[Cla99] Clark, James. *XSL Transformation (XSLT) Version 1.0*. W3C Recommendation. November 1999. **http://www.w3.org/TR/xslt**.

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

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

[COVER] The OASIS Cover Pages: The Online Resource for Markup Language Technologies. **http://www.oasis-open.org/cover/schemas.html**. June 2003.

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

[ea00] Pemberton, Steven, et alt. *XHTML 1.0 The Extensible HyperText Markup Language (Second Edition)*. W3C Recommendation. January 2000. **http://www.w3.org/TR/xhtml1**.

[FpMLReq] FpML Architecture Working Group. FpML Validation Language Requirements. FpML Technical Report. March 2003. **http://www.fpml.org/documents/working-papers/technical_notes/ValidationRequirements2003-03-04.pdf**

[Ham02] Hamscher, Walter. *XBRL and its relationship to XML Web Services, ebXML and other infrastructures for e-Business*. May 2002. **http://home.earthlink.net/~hamscher/Docs/XBRL-WS-ebXML-etc-2002-05-25.pdf**.

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

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

[LC00] Lee, Dongwon, and Chu, Wesley W. Comparative Analysis of Six XML Schema Languages. In *ACM SIGMOD Record, Vol. 29, No. 3.* Sep 2000. **http://nike.psu.edu/publications/sigmod-record-00.pdf**.

[Mas02] Masayasu, Ishikawa. *XHTML 1.0 in XML Schema*. W3C Note. September 2002. **http://www.w3.org/TR/2002/NOTE-xhtml1-schema-20020902/Overview.html**.

[MSV04a] Marinelli, Paolo, Sacerdoti Coen, Claudio, and Vitali, Fabio. SchemaPath, a Minimal Extension to XML Schema for Conditional Constraints. In *Proceedings of the 13th International World Wide Web Conference*, pages 164-174. New York, NY, USA. May 2004. ISBN:1-58113-844-X.

[MSV04b] Marinelli, Paolo, Sacerdoti Coen, Claudio, and Vitali, Fabio. *A Formal Semantics for SchemaPath*. Technical Report. To be published.

[NCEF02] Nentwich, Christian, Capra, Licia, Emmerich, Wolfgang, and Finkelstein, Anthony. xlinkit: A Consistency Checking and Smart Link Generation Service. In *ACM Transaction on Internet Technology*. May 2002.

[Rob02] Robertsson, Eddie. *Combining Schematron with other XML Schema languages*. June 2002. **http://www.topologi.com/public/Schtrn_XSD/Paper.html**.

[TBMM01] Thompson, Henry S., Beech, David, Maloney, Murray, and Mendelsohn, Noah. *XML Schema Part 1: Structures*. W3C Recommendation. May 2001. **http://www.w3.org/TR/xmlschema-1/**.

[VAG03] Vitali, Fabio, Amorosi, Nicola, and Gessa, Nicola. Datatype- and namespace-aware DTDs. In *Proceedings of Extreme Markup 2003.* August 2003. Montreal, Canada. **http://www.mulberrytech.com/Extreme/Proceedings/html/2003/Gessa01/EML2003Gessa01.html**.

[Wal01] Walsh, Norman. *Three Schemas. Schema Test Cases for the Schema Comparison Panel*. October 2001.**http://nwalsh.com/xml2001/schematownhall/schema.html**.

[WC01] Walsh, Norman, and Cowan, John. *Schema Language Comparison*. December 2001.**http://nwalsh.com/xml2001/schematownhall/slides/**.

[WRM03] Warren, Paul, Reakes, Gareth, and Massari, Alberto. Business Rules Validation - the Standard the W3C Forgot. In *Proceedings of the XML Europe 2003 Conference*. May 2003. **http://www.idealliance.org/papers/dx_xmle03/papers/03-02-03/03-02-03.pdf**.

[WS03] Wilde, Erik, and StillHard, Kilian. A Compact XML Schema Syntax. In *Proceedings of XML Europe 2003*. London. May 2003. **http://dret.net/netdret/docs/wilde-xmleurope2003.html**.

The Authors

Davide Fiorello
University of Bologna, Department of Computer Science
Mura A. Zamboni, 7
Bologna
Italy
fiorello@cs.unibo.it

Davide Fiorello 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 is a Ph.D. student since January 2003.

Paolo Marinelli
University of Bologna, Department of Computer Science
Mura A. Zamboni, 7
Bologna
Italy
pmarinel@cs.unibo.it

Paolo Marinelli holds a Laurea degree in Computer Science at the University of Bologna. The topic of his Master Thesis regards SchemaPath, the conservative extension of XML Schema for expressing conditional content models and co-constraints, partially described in this paper.

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 hypertext functionalities, the World Wide Web, and XML.

Extreme Markup Languages 2004
Montréal, Québec, August 2-6, 2004
This paper was formatted from XML source via XSL
by Mulberry Technologies, Inc.