SchemaPath, a Minimal Extension to XML Schema for Conditional Constraints

Paolo Marinelli
Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7, 40127
Bologna, ITALY
pmarinel@cs.unibo.it

Claudio Sacerdoti Coen
Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7, 40127
Bologna, ITALY
sacerdot@cs.unibo.it

Fabio Vitali
Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7, 40127
Bologna, ITALY
fabio@cs.unibo.it

ABSTRACT

In the past few years, a number of constraint languages for XML documents has been proposed. They are cumulatively called *schema languages* or validation languages and they comprise, among others, DTD, XML Schema, RELAX NG, Schematron, DSD, xlinkit.

One major point of discrimination among schema languages is the support of co-constraints, or co-occurrence constraints, e.g., requiring that attribute `A` is present if and only if attribute `B` is (or is not) present in the same element. Although there is no way in XML Schema to express these requirements, they are in fact frequently used in many XML document types, usually only expressed in plain human-readable text, and validated by means of special code modules by the relevant applications.

In this paper we propose SchemaPath, a light extension of XML Schema to handle conditional constraints on XML documents. Two new constructs have been added to XML Schema: conditions – based on XPath patterns – on type assignments for elements and attributes; and a new simple type, `xsd:error`, for the direct expression of negative constraints (e.g. it is prohibited for attribute `A` to be present if attribute `B` is also present).

A proof-of-concept implementation is provided. A Web interface is publicly accessible for experiments and assessments of the real expressiveness of the proposed extension.

Categories and Subject Descriptors

I.7.2 [**Document and Text Processing**]: Document Preparation—*markup languages*; D.3.2 [**Programming Languages**]: Language Classifications—*extensible languages*

General Terms

Languages

Keywords

XML, Co-constraints, Schema Languages, SchemaPath

1. INTRODUCTION

Validating a document is, in XML parlance, the process of verifying whether an XML document accords to a set of structural and content rules expressed in one of the many *schema languages* proposed by a number of different organizations and individuals.

The first and best-known schema language for XML is surely DTD (Document Type Definition), which was introduced and standardized within the XML recommendation itself [6], and which is a direct derivation and simplification of its counterpart in SGML, the immediate ancestor of XML as a meta-markup language.

Although DTDs proved fairly useful in the publishing domain, where most rules deal with the explicit structure of the document, rather than its content (e.g. requiring sections to have titles, or figures to have captions), new, unforeseen applications of XML clearly showed the limitations of DTDs. For instance, data exchange applications may want to make sure that the structures being exchanged are not only correctly labeled (structural constraints), but also that they contain correct data (for instance, a date is a date, an integer is an integer, a zip code is a zip code).

The number of schema languages created to overcome the limitations of DTDs is vast; in [2], a fairly authoritative source, 15 different schema languages for XML are listed besides DTDs, and at least one more, xlinkit [17], is missing. One of these languages, XML Schema [22, 5] is directly backed by the World Wide Web Consortium, and is being toted as the only and *official* schema language for XML documents, the natural substitution for DTDs. ISO, on the other hand, is active in DSDL [3], the international standardization of a couple of schema languages, RELAX NG [9] and Schematron [12], that are having a fair success despite being absolutely ignored by the W3C.

Roughly speaking, schema languages can be seen as belonging to one of two types:

- *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 and RELAX NG, as well as DTDs themselves, fall into this category.
- *rule-based languages*, by which document engineers list the rules that the XML document must satisfy, providing either an open specification (all that is not forbid-

den is allowed) or a closed specification (all that is not allowed is forbidden) [23]. Schematron and xlinkit belong to this category.

It is futile to decide which of these is the best schema language for XML documents. Each is tailored towards a different shade of validation requirements, and each provides a rich set of features often unmatched by the others: for instance, just limiting ourselves to the best-known candidates, DTDs supports character entities, XML Schema has a rich set of predefined data types and a sophisticated derivation mechanism, RELAX NG sports a simple and straightforward syntax, Schematron provides a powerful XPath-based rule mechanism.

At the XML 2001 conference, a panel of experts was summoned to test drive and compare these four schema languages and determine their strength and weaknesses. A final report was issued, in the form of a set of slides [23]. Strengths and weaknesses were collected in five major categories:

- *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, such as the mutual exclusion (only one of two different attributes can be present in an element).

At first glance, Schematron appears a clear winner: it supports most of the listed features, and practically alone dominates the co-constraints category, for which neither XML Schema nor DTDs offer any support at all, and RELAX NG appears clearly limited. Yet, XML Schema provides the best built-in datatypes and the most sophisticated mechanism for user-defined types, whereas Schematron has a limited number of data types and no way to specify default values.

Nonetheless, the problem of co-constraints (also known as *co-occurrence 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, [11]) that DTDs, XML Schema, etc., do not allow co-constraints: thus they provide these rules in natural language (with the obvious problems given by ambiguity and interpretation) and they recommend implementers to support the relevant rules directly in their software.

An example comes from FpML (Financial Products Markup Language), a markup language for financial derivatives trades. An XML Schema schema for FpML 4.0 exists, but is not able to capture many normative requirements (called *validation rules*), which are expressed in natural language.

Even a number of well-known W3C languages dictate normative co-constraints, expressing them in the plain text of the language description but not in the formal schema specifications. For instance, in XHTML the recursive presence of `<a>` elements within other `<a>` elements is prohibited[1], as specified in Appendix B of [19], yet it is expressed neither in the DTD nor in the XML Schema. XML Schema itself includes a number of co-constraints that cannot be expressed in the language, such as the mutual incompatibility of the `ref` and `name` attributes in an element or attribute definition.

Imposing constraints that cannot be expressed in the schema language of choice really **is** a serious shortcoming for interchange applications. The validation phase, in these applications, has the overall goal to ensure with minimum effort that the XML data does in fact conform to the prespecified rules. When not all rules can be expressed in the schema language, either some constraints will not be verified, or code will have to be written to implement the verification in the downstream application, forcing implementers to provide their own validation code, with repetition of efforts and no guarantee of correct and widespread implementations.

Yet, as mentioned, no single schema language provides all the necessary features for a rich and complex XML document type. Proposals have been made to mix two of them and take the best from both: for instance, it has been proposed [18, 1] to embed a rule-based specification in Schematron within a grammar-based XML Schema document, so that the cooperation of both validations yields the desired control onto the XML documents. However, this solution is complicated and not completely satisfying (see, for instance, the discussion in Sect. 3.5).

In this paper we propose an alternative solution, called *SchemaPath*. SchemaPath is a conservative extension of XML Schema obtained adding conditional type attributions to elements and attributes. Conditional attributions can express co-constraints in a simple and compact syntax, and provide much more than co-constraints, as will be explained in the following. SchemaPath extends the XML Schema language with just one new construct and one new built-in type, so that understanding the differences and adopting the extensions is really trivial, especially comparing with the issues arising with the adoption of a completely different schema language such as Schematron. A proof-of-concept implementation of a validation engine for SchemaPath has been developed to help understanding the advantages of our approach. It can be tested on-line or downloaded for local use at `http://genesispc.cs.unibo.it:3333/schemapath.asp`.

The paper is structured as follows: in Sect. 2 we provide a brief recapitulation of some of the schema languages mentioned in the paper, with some attention to the issue of co-constraints. Sect. 3 describes the proposed syntax of SchemaPath, with some examples on using the new construct and the new type, and a brief proof of the correctness of our extension. Sect. 4 selects a few interesting co-constraints listed in the textual documentation of well-known W3C languages such as XHTML, XSLT, XML Schema, but not formalized in the corresponding schema specifications, and shows how they can somewhat elegantly be formalized in SchemaPath. Finally Sect. 5 describes our implementation.

[1] This is technically considered an *exclusion*, rather than a co-constraint, but there is only a very little difference.

2. SCHEMA LANGUAGES FOR XML

In this section, we examine the most relevant schema languages for XML documents, providing a few examples and paying particular care to the issues connected to co-constraints.

Besides DTDs [6] and XML Schema [22, 5], the best known schema languages include RELAX [16], TREX [8], RELAX NG [9], Schematron [12], DSD [13] and xlinkit [17].

As mentioned, these languages can be roughly divided in two categories, basing on their approach to validation: grammar-based languages and rule-based languages. As a first approximation, Schematron and xlinkit are rule-based languages, while the others are grammar-based.

2.1 DTDs

DTDs have been originally introduced for validating SGML structures, and then ported to provide validation for XML documents.

DTDs provide a sophisticated regular expression language for imposing constraints on elements and subelements (the so-called *content model*), but are very limited in the control of attributes and data elements. No data type exists to speak of, with the exception of strings, whitespace-free strings, and enumerations of strings. Furthermore, only attributes allow enumerations and default values, while text elements cannot even be checked for `null` values.

Since DTDs precede temporally the advent of namespaces, they provide no support for qualified elements, although by fixing the prefix associated to a namespace some support for validation can be obtained.

SGML DTDs are richer and more complex than XML DTDs, and in particular they have a feature that would be of wide interest in our discussion: *exclusions*. Exclusions specify that one or more elements cannot appear within an element or any of its children, providing a deep exception to the content model of an element. An example of exclusions is shown in Sect. 4.1. In a way, exclusions represent one kind of co-constraint, the only possible with DTDs (and only SGML DTDs, by the way!)

2.2 XML Schema

XML Schema is a W3C recommendation [22, 5] aimed at replacing DTDs as the official schema language for XML documents. It is by far the most widely supported schema language after DTDs, and provides a large number of improvements over them.

The first and most evident improvement is the switch to an XML-based syntax, which worsens the language in terms of readability and terseness, but highly improves it in terms of flexibility and automatic processability. Moreover XML Schema is completely namespace-aware.

Another major contribution of XML Schema is the Post Schema Validation Infoset (PSVI), i.e., the additional information that the validation adds to the nodes of the XML document so that downstream applications can make use of it for their own purposes. The most important advantage of PSVI is certainly the type, or the set of legal values that a node can have.

Types in XML Schema are either simple (strings with various constraints) or complex (markup substructures of the XML document including elements, attributes and text nodes). A large number of built-in simple types are provided, ranging from integers to dates, times, URIs, etc. Schema authors are encouraged to create new types by deriving existing ones, restricting their values via one of the many facets provided (such as the length of the string, the maximum or minimum values, etc.). Complex types are not predefined, but can be created providing an expression on the corresponding markup.

Types are either named, and referred to via their name, or anonymous, and inserted inline within the relevant elements and attributes. Elements are either global (defined directly within the `<schema>` element) or local (i.e., defined within a `<complexType>`). Local elements of different complex types can have the same name and different types without limitations.

The real strength of XML Schema lies in the rich collection of built-in simple types and the number of facets that can be applied to them. XML Schema improves over DTDs in complex types as well: it reintroduces unordered content models (although with some restrictions), and it allows controlled repetitions of elements (with the `minOccurs` and `maxOccurs` specifications).

XML Schema has no support at all for co-constraints or DTD-like exclusions, which is exactly the kind of limitations SchemaPath sets out to improve.

2.3 RELAX NG

RELAX NG [9] is a schema language for XML developed by an international working group, ISO/IEC JTC1/SC34/WG1. It is based on two preceding languages, TREX [8], designed by James Clark, and RELAX [16], designed by Murata Makoto.

Patterns are the central concept of RELAX NG. They extend the concept of content model: while in DTDs a content model is an expression over elements (and, very limitedly, text), in RELAX NG a pattern is an expression over elements, text nodes and attributes.

External definitions of datatypes can be used for constraining the set of values of text nodes and attributes. The most common datatype library is the one defined by XML Schema in [5].

Differently from DTDs and XML Schema, RELAX NG imposes no restriction to elements with unordered content and allows ambiguous definitions, i.e., elements with the same name and different content models, in the same context.

The symmetric treatment of elements, attributes and text nodes and the introduction of ambiguous definitions allow RELAX NG to specify a number of co-constraints on XML documents, such as mutual exclusion, inter-dependencies between elements and attributes, and others. To illustrate, consider the following example,

```
<element name="x">
  <choice>
    <attribute name="a"/>
    <attribute name="b"/>
  </choice>
</element>
```

that imposes that "*an* `<x>` *element must have either an* `a` *attribute or a* `b` *attribute, but not both*".

However such co-constraints must be defined as patterns, and there are co-constraints and context-dependent definitions that may produce extremely long patterns. Furthermore, RELAX NG cannot be used to define all types of constraints definable in other schema languages such as xlinkit, Schematron, and our own SchemaPath.

Finally, another limitation of RELAX NG is its inability to define default values for elements and attributes.

2.4 DSD

DSD [13] is a schema language co-developed by AT&T Labs and BRICS. At the time of writing, there are two version of DSD: DSD 1.0 and, recently, DSD 2.0 (DSD2)[2].

Constraints are the central concept in DSD. A constraint is used to specify the content of an element, its attributes and its context (the sequence of nodes from the root to the element). An *element definition* specifies a pair consisting of an element name and a constraint.

The element content is constrained by a *content expression*, that is, a regular expression over element definitions.

Context patterns are used to impose constraints on the context of an element. For instance, the context pattern

```
<Context>
  <Element Name="x">
    <Attribute Name="a" Value="v"/>
  </Element>
  <SomeElements/>
  <Element name="y"/>
</Context>
```

is matched by all `<y>` elements within an `<x>` element, having an `a` attribute, whose value is `"v"`.

A particular constraint expression is the *conditional constraint expression*, an "if-then-else" construct whose boolean expression is built with XML elements describing boolean connectors. A few examples will be shown in Sect. 4.

A problem of DSD is that the boolean expressions are able to test conditions just on ancestor elements and their attributes, and not on sibling and descendant nodes too. Moreover, they do not provide any comparison operator between values, and thus they appear limited in some cases. Finally, being built on XML elements, boolean expressions can easily become very verbose.

2.5 Schematron

Schematron [12] is a rule-based schema language created by Rick Jelliffe at the Academia Sinica Computing Center (ASCC). It has great expressive power, and is mainly used to check co-constraints in XML instance documents.

A Schematron document defines a sequence of `<rule>`s, logically grouped in `<pattern>` elements. Each rule has a `context` attribute, which is an XPath pattern determining which elements in the instance document the rule applies to. Within a rule, a sequence of `<report>` and `<assert>` elements is specified, having a `test` attribute, which is an XPath expression evaluated to a boolean value for each node in the context. The content of both `<report>` and `<assert>` is an *assertion*, which is a declarative sentence in natural language. When the test of a `<report>` succeeds, its content is output. Contrarily, when the test of an `<assert>` fails, its content is output. Thus, the `<report>` element is used to tag negative assertions about the instance document, while the `<assert>` element is used to tag positive ones. Therefore, the output of the Schematron validation process is a list of assertions.

A Schematron schema can be easily transformed into an equivalent XSLT document. In particular, a `<rule>` element can be transformed into a `<template>`, whose `match` attribute is set to the context of the rule. Both `<assert>` and `<report>` elements can be mapped into conditional XSLT elements (e.g., `<choose>` and `<if>`).

In fact, Schematron is commonly implemented as a meta-stylesheet, called *skeleton*. This skeleton is applied to the Schematron schema and the resulting XSLT is in turn applied to the XML instance document.

The strength of Schematron is in the facility of the co-constraints definition. For instance, the sentence "*if an `<x>` element has an `a` attribute, then it must also have a `y` element, otherwise it must be empty*" can be formalized by the following rule:

```
<rule context="x">
  <report test="@a and not(y)">Error: no y element.</report>
  <report test="not(@a) and *">Error: x must be empty.</report>
</rule>
```

However, the rule-based approach of Schematron does not allow to easily express usual constraints on the content of an element that are naturally captured by grammatical expressions. Furthermore, Schematron is not able to impose complex restrictions on text values.

2.6 xlinkit

xlinkit [17] is more than a schema language: it is an application service that provides rule-based link generation and checks the consistency of distributed web content.

An xlinkit application defines a set of documents and a set of *consistency rules*, and proceeds to verify the rules on every file in the document set. The rule is expressed in the constraint language CLIX, which is a first-order language using predicates, quantifiers and variables. For instance, the rule "*all `<x>` elements must have an `a` attribute*" is expressed as follows:

```
<consistencyrule id="r1">
  <forall var="x" in="//x">
    <exists var="y" in="$x/@a">
  </forall>
</consistencyrule>
```

The XPath expression specified in the `in` attribute of the `<forall>` element is applied to the documents that belong to the document set. Then, a union of the node sets returned by the evaluation of the expression is computed. The sub-formula `<exists>` is then evaluated for each node in the union.

When consistency rules are checked, the produced output is not a boolean value, but rather a set of XLink [10].

It is possible to express rather complex co-constraints in xlinkit, such as the one stating that "*all `<x>` elements must have a `n` attribute whose value (treated as a number) defines the exact number of its children*". Nevertheless, being rule-based, xlinkit is more suited for expressing co-constraints than for expressing the classical constraints of DTDs and XML Schema.

3. SCHEMAPATH

In this section we illustrate the SchemaPath syntax, and show a few fragments of specifications that demonstrate its flexibility. SchemaPath is a conservative extension to XML Schema. This means that any correct XML Schema is also a correct SchemaPath. This also means that in order to obtain a rich SchemaPath specification, one can start writing a normal XML Schema specification, and then just add those conditions that cannot be expressed in XML Schema.

[2]Here we address DSD 1.0, because only a prototype Java processor for DSD2 has been implemented. In the following, we use the term DSD for DSD1.0.

SchemaPath adds one new construct, the `<xsd:alt>` element, for expressing alternative type attributions for elements and attributes, and one new built-in datatype, `xsd:error`, for the direct expression of negative rules, i.e., rules that need to be *not* satisfied for validity.

3.1 The `<xsd:alt>` element

Within an element or attribute definition, it is possible to subject the type attribution to alternative conditions expressed with a number of `<xsd:alt>` elements containing an XPath pattern, an optional explicit priority of choice (much like in XSLT templates), and the type to be applied to the element or attribute if the condition holds.

The simplest example is subjecting the type of an element to the value of another element. For instance: "*the `<quantity>` of an `<invoiceLine>` is of type integer if the value of `<unit>` is `items`, and of type decimal if the value of `<unit>` is `meters`*".

In this case, we create a conditional type attribution for the element `<quantity>`, with two alternative types, `xsd:integer` and `xsd:decimal` according to the relative condition expressed as XPath templates.

```
<xsd:element name="invoiceLine">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="unit" type="unitType"/>
      <xsd:element name="quantity">
         <xsd:alt cond="../unit='items'"  type="xsd:integer"/>
         <xsd:alt cond="../unit='meters'" type="xsd:decimal"/>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

The SchemaPath engine makes no assumption about the validity of the XPath pattern: any pattern can be expressed, even an impossible one according to the data type of the element; if this is the case, the pattern will simply be never satisfied by the document, and the alternative never chosen. For instance, there is no control in the previous example that a `<unit>` element is actually defined as a sibling of the `<quantity>` element: if this is the case, then the pattern may be satisfied, otherwise it will be always ignored. On the other hand, failure to satisfy at least one alternative condition will yield a validation error.

3.2 Priorities

It is of course possible for an element or an attribute to match more than one condition at the same time. For instance, in the following declaration,

```
<xsd:element name="quantity">
  <xsd:alt cond="../unit='items'" type="xsd:integer"/>
  <xsd:alt cond="../unit"         type="xsd:decimal"/>
</xsd:element>
```

the first condition checks that the `<unit>` element contains the string `"items"`, while the second just verifies that a `<unit>` element is present; of course, a situation satisfying the first condition also satisfies the second one.

In these situations, alternatives should be given different priorities through the optional `priority` attribute, which is a positive or negative real number. If not explicitly specified, `priority` defaults to `0.5`. Thus, the example above could be rewritten as follows:

```
<xsd:element name="quantity">
  <xsd:alt cond="../unit='items'" type="xsd:integer"/>
  <xsd:alt cond="../unit"         type="xsd:decimal"
           priority="0"/>
</xsd:element>
```

assuring that the second alternative is chosen only when the first one does not hold.

It is an error if during the validation process an element or attribute simultaneously matches two or more conditions with the same priority. In such a case, a SchemaPath processor may signal the error, otherwise it must recover by choosing the alternative occuring last in lexical order.

3.3 The `xsd:error` simple type

The `xsd:error` built-in simple type is an unsatisfiable type, i.e., its *value space* [5] is empty. Thus, assigning an `xsd:error` to an element (or, more likely, to an alternative condition in an element definition) will yield a validation error.

The `xsd:error` can be used to directly express a negative condition, i.e., a condition that we do not want to hold in our XML documents. It is used for roughly the same purposes as the `<report>` statement in Schematron.

The simplest example of use of the `xsd:error` is mutual exclusion, e.g. to prevent the presence of an attribute in an element when another attribute is already present. For instance: "*the `<description>` element of the `<invoiceLine>` can have either a `print` attribute, with the internal code for the type of print, or a `color` attribute, with the Pantone code of the color of the dye. It is incorrect for the element to have both attributes*".

In this case, we provide a direct type to one of the two attributes, and a conditional attribution to the other, selecting the `xsd:error` type if the first attribute is already present.

```
<xsd:element name="description">
  <xsd:complexType>
     <xsd:attribute name="print" type="PrintCodeType"/>
     <xsd:attribute name="color">
        <xsd:alt priority="0"     type="PantoneCodeType"/>
        <xsd:alt cond="../@print" type="xsd:error" />
     </xsd:attribute>
  </xsd:complexType>
</xsd:element>
```

The first `<xsd:alt>` has no condition expressed. This is a shorthand for expressing an always true condition, and allows for the specification of a default type assignment (i.e., for all those situations where no explicit condition holds). An alternative that uses the default condition should always have a priority lower than those of all the other alternatives, as shown in the example. Therefore, the `color` attribute will be of type `xsd:error` if the condition holds, and of type `PantoneCodeType` in all other cases.

3.4 Namespaces and qualified conditions

SchemaPath defines the namespace `http://www.cs.unibo.it/SchemaPath/1.0`, but it also accepts schemas belonging to the plain XML Schema namespace. Either one can be used, provided it is used consistently.

SchemaPath supports namespaces and qualified elements and attributes in much the same way as XML Schema does. Yet, SchemaPath makes one restricted assumption on patterns on fully qualified elements, borrowed from XSLT. A well-known constraint of XSLT (which on the other hand is being stigmatized and scheduled for removal in the next version, as specified in [15]) is that patterns on fully qualified elements need to have a non-null prefix to work. SchemaPath makes the same constraint and plans to remain aligned on this issue to XSLT, removing it only when the corresponding constraint will be removed from XSLT.

This means that it is not possible to use a default namespace (i.e., without a prefix) as the target namespace in SchemaPath, because the XPath patterns would not work correctly.

3.5 PSVI

A peculiar feature of XML Schema is the PSVI. This is the information associated to the memory representation of all the nodes of an XML document after having been parsed and validated. This information can then be used by a downstream application for performing specific computations on nodes and node values.

SchemaPath does not modify the content of the PSVI for valid documents. In fact, the `<xsd:alt>` structure added by SchemaPath does not survive the validation phase, since it is only used to determine the actual type to be associated to the element. In a way, it is equivalent to a specific type attribution via an `xsi:type` attribute in the XML document itself.

The only difference in PSVI is for invalid documents: since SchemaPath adds another built-in type, namely `xsd:error`, an invalid element may have been assigned the `xsd:error` type and (obviously) have failed the validation.

The precision of the PSVI generation is an important difference between SchemaPath and the proposal [18, 1] of embedding Schematron-like rules within a XML Schema specification. Indeed, the PSVI obtained validating a document against a SchemaPath schema is strict, since it describes the precise type assigned to each element after the evaluation of the guards of the conditional type assignements. On the contrary, in the alternative approach, to express a co-constraint we are often forced to declare lax XML Schema types, that also accept wrong values. The Schematron validator is responsible of rejecting invalid values, but that does not affect the generated PSVI. Thus the lax types reach the PSVI, that becomes less informative. Therefore, in general, the PSVI obtained with SchemaPath is more precise that with XML Schema and Schematron together, which represents a major advantage of our proposal.

3.6 A Formalization of SchemaPath

Since we are proposing an extension of XML Schema, we need to grant that all the interesting properties of XML Schema still hold for SchemaPath. Thus we have formally described the SchemaPath semantics, adapting the formalization of XML Schema presented in [20]. Due to lack of space, the formalization will not be presented in this paper. The interested reader can find it in [14].

The formalization allows us to prove several important results. The first one is that the *validation theorem* holds for SchemaPath, too.

The validation theorem for XML Schema proves that an untyped XML document validates against a given schema yielding a typed tree if and only if the typed tree matches the type given in the schema and yields the original document when types are removed.

Intuitively, the validation theorem asserts that the PSVI built during the validation phase is a faithful representation of both the original XML document (when the types are not considered) and the type derivation that proves that the document is well-typed according to the schema. The above mentioned property of PSVI holds when the schema is expressed in SchemaPath, too.

The second important result is that the roundtripping and reverse-roundtripping properties holds for SchemaPath under the same set of conditions required for XML Schema.

The roundtripping property states that serializing into XML a PSVI and deserializing it again yields the original PSVI. In other words, using the XML format to communicate the Post Schema Validation Infoset to another application is not a lossy operation.

Reverse-roundtripping is the property that assures that validating an XML document and then serializing the obtained Post Schema Validation Infoset yields exactly the original XML document. In other words, the deserialization and serialization cycle is idempotent: a document can be parsed and saved back as many times as we want without loosing or changing the information it conveys.

The two properties are a direct consequence of the validation theorem, that grants a perfect correspondence between the PSVI and the pair formed by the original XML document and its schema, and they hold for SchemaPath schemas just as they do for XML Schema[3].

To summarize, the SchemaPath conservative extension of XML Schema satisfies all the good theoretical properties identified so far for XML Schema. In particular, we proved the validation theorem for SchemaPath, that implies a fundamental practical property of a schema language: for a large class of documents, the PSVI does not change when a document is serialized (saved) and deserialized (loaded).

4. CO-CONSTRAINTS IN SCHEMAPATH

In this section we provide a comparison between SchemaPath and other schema languages already described in previous sections. The comparison is by examples on co-constraints. We have chosen a number of "famous" examples, taken by important W3C specifications.

4.1 No nesting of `<a>` elements in XHTML

In Appendix B of the XHTML recommendation [19], some element prohibitions are listed. These prohibitions are specified in natural language, since neither DTD nor XML Schema can be used to specify them.

The first (and most widely known) element prohibition is the exclusion of elements `<a>` within an element `<a>`. This means that hypertext anchors cannot nest regardless of their level.

Existing schemas for XHTML only provide a subformulation of the exclusion: they cannot prevent the nesting of `<a>` elements within `<a>` elements at all levels, but just at the first level.

Actually it is technically possible to enforce the rule in XML Schema, but this would involve duplicating a large part of the specification, creating two subschemata (one with and one without `<a>` as an allowable element) to be used outside and within the outermost `<a>` element [21]. Of course

[3]Unfortunately, due to a bad design choice of XML Schema, the two properties hold only for schemas satisfying certain conditions. SchemaPath, being a conservative extension of XML Schema, suffers from the same limitation, without augmenting its severity: we designed SchemaPath so that no new conditions restricting the set of instances that satisfy the roundtripping and reverse roundtripping properties are introduced.

Table 1: Example: No nesting of <a> elements in XHTML

Schematron
`<rule context="x:a">`
`  <report`
`    test="./x:a"`
`  >a must not contain other`
`   a elements</report>`
`</rule>`
	DSD
	`<ElementDef ID="a">`
	` <Not>`
	`  <Context>`
	`   <Element Name="a"/>`
	`   <SomeElements/>`
	`   <Element Name="a"/>`
	`  </Context>`
	` </Not>`
	` <Content IDRef="Inline"/>`
	`</ElementDef>`
		xlinkit
		`<consistencyrule>`
		`  <forall var="x" in="//x:a">`
		`    <not>`
		`      <exists var="y" in="$x//x:a"/>`
		`    </not>`
		`  </forall>`
		`</consistencyrule>`

SGML DTD
`<!ENTITY % inline`
`   "(#PCDATA | a | b | ... )">`
`<!ELEMENT a - - (%inline;) -(a) >`
	SchemaPath
	`<xsd:element name="a">`
	` <xsd:alt cond=".//x:a" type="xsd:error"/>`
	` <xsd:alt priority="0"  type="x:a.type"/>`
	`</xsd:element>`	

Table 2: Example: Named templates in XSLT

xlinkit
`<consistencyrule id="r">`
` <forall var="x" in="//xsl:template">`
`  <implies>`
`   <not>`
`    <exists var="y" in="$x/@name"/>`
`   </not>`
`   <exists var="z" in="$x/@match"/>`
`  </implies>`
` </forall>`
`</consistencyrule>`
	Schematron
	`<rule context="xsl:template">`
	` <report test="not(@name) and not(@match)"`
	` >Error</report>`
	`</rule>`
		RELAX NG
		`<element name="template">`
		` <choice>`
		`  <group>`
		`   <attribute name="name">`
		`    <data type="NCName"/>`
		`   </attribute>`
		`   <attribute name="match">`
		`    <ref name="Pattern"/>`
		`   </attribute>`
		`  </group>`
		`  <attribute name="name">`
		`   <data type="NCName"/>`
		`  </attribute>`
		`  <attribute name="match">`
		`   <ref name="Pattern"/>`
		`  </attribute>`
		` </choice>`
		` <ref name="templateContent"/>`
		`</element>`

DSD
`<ElementDef ID="template">`
` <AttributeDecl Name="match" Optional="yes">`
`  <StringType IDRef="Pattern"/>`
` </AttributeDecl>`
` <AttributeDecl Name="name" Optional="yes">`
`  <StringType IDRef="NCName"/>`
` </AttributeDecl>`
` <If><Not><Attribute Name="name"/></Not>`
`  <Then><Attribute Name="match"/></Then>`
` </If>`
` <Content IDRef="templateContent"/>`
`</ElementDef>`
	SchemaPath
	`<xsd:element name="template">`
	` <xsd:alt cond="not(@match) and not(@name)" type="xsd:error" />`
	` <xsd:alt priority="0"                        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>`	

this rapidly leads to unmanageable specifications, given their size and complexity. RELAX NG has the same problem, and must use a similar solution with similar limitations.

SGML DTDs had the exact construct needed, exclusions, but it was later removed from XML DTDs. On the other hand, SchemaPath, xlinkit, Schematron and DSD all allow reasonable definitions of the constraint by means of consistency rules on the nesting of elements. Possible solutions using these languages are proposed in Table 1.

4.2 Named templates in XSLT

An `<xsl:template>` element may contain both a `match` and a `name` attribute. The XSLT recommendation [7] describes the relation between `match` and `name` attributes as follows:

> [Section 5.3] The `match` attribute is required unless the `xsl:template` element has a `name` attribute.

> [Section 6] If an `xsl:template` element has a `name` attribute, it may, but need not, also have a `match` attribute.

The above sentences can be restated as "*the absence of the* `name` *attribute implies the presence of the* `match` *attribute*".

xlinkit and DSD directly formalize this new restatement of the constraint. Schematron and SchemaPath can be used to report an error when both `name` and `match` are missing (which is logically equivalent to the restatement). RELAX NG has no conditional statements, so a verbose grammatical expression is required. Finally, DTDs and XML Schema provide no way to define this constraint.

Table 2 compares the different solutions.

4.3 Elements in XML Schema

There are some syntactic requirements imposed on a XML Schema document that cannot be described by XML Schema itself. For instance, section 3.3.2 of [22] states that:

Table 3: Example: XML Schema specification

DSD

```
<ContentDef ID="elementContent">
 <Optional>
  <Union>
   <Element IDRef="simpleType"/>
   <Element IDRef="complexType"/>
  </Union>
 </Optional>
</ContentDef>
<ElementDef ID="globalElement"
 Name="element">
 <AttributeDecl Name="name">
  <StringType IDRef="NCName"/>
 </AttributeDecl>
 <AttributeDecl Name="type"
  Optional="yes">
  <StringType IDRef="QName"/>
 </AttributeDecl>
 <If><Attribute Name="type"/>
  <Then><Empty/></Then></If>
 <Content IDRef="elementContent"/>
</ElementDef>
<ElementDef ID="localElement"
 Name="element">
 <AttributeDecl Name="name"
  Optional="yes">
  <StringType IDRef="NCName"/>
 </AttributeDecl>
 <AttributeDecl Name="type"
  Optional="yes">
  <StringType IDRef="QName"/>
 </AttributeDecl>
 <AttributeDecl Name="ref"
  Optional="yes">
  <StringType IDRef="QName"/>
 </AttributeDecl>
 <If><Attribute Name="ref"/>
  <Then>
   <Empty/>
   <Not>
    <Or>
     <Attribute Name="type"/>
     <Attribute Name="name"/>
    </Or>
   </Not>
  </Then>
  <Else>
   <If><Attribute Name="type"/>
    <Then><Empty/></Then></If>
  </Else></If>
 <Content IDRef="elementContent"/>
</ElementDef>
```

xlinkit

```
<consistencyrule id="local-ref">
 <forall var="x"
  in="//xsd:element[@ref]">
  <and>
   <not>
    <exists var="y"
     in="$x/parent::xsd:schema"/>
   </not>
   <and>
    <not>
     <exists var="y"
      in="$x/@name or $x/@type"/>
    </not>
    <not>
     <exists var="y"
      in="$x/xsd:simpleType | $x/xsd:complexType"/>
    </not>
   </and>
  </and>
 </forall>
</consistencyrule>
<consistencyrule id="no-ref">
 <forall var="x"
  in="//xsd:element[not(@ref)]">
  <and>
   <exists var="y" in="$x/@name"/>
   <not>
    <and>
     <exists var="y" in="$x/@type"/>
     <exists var="y"
      in="$x/xsd:complexType | $x/xsd:simpleType"/>
    </and>
   </not>
  </and>
 </forall>
</consistencyrule>
```

RELAX NG

```
<define name="globalElement">
 <element name="element">
  <attribute name="name">
   <data type="NCName"/>
  </attribute>
  <ref name="namedOrAnon"/>
 </element>
</define>
<define name="localElement">
 <element name="element">
  <choice>
   <attribute name="ref">
    <data type="QName"/>
   </attribute>
   <group>
    <attribute name="name">
     <data type="NCName"/>
    </attribute>
    <ref name="namedOrAnon"/>
   </group>
  </choice>
 </element>
</define>
<define name="namedOrAnon">
 <choice>
  <attribute name="type">
   <data type="QName"/>
  </attribute>
  <optional>
   <choice>
    <ref name="complexType"/>
    <ref name="simpleType"/>
   </choice>
  </optional>
 </choice>
</define>
```

Schematron

```
<pattern name="elementDecl">
 <rule context="xsd:element[@ref]">
  <assert test="not(parent::xsd:schema)"
  >@ref is not allowed.</assert>
  <assert test="not(@type) and not(@name)"
  >@type and @ref are not allowed.</assert>
  <assert test="not(xsd:simpleType) and
              not(xsd:complexType)"
  >anonymous type is not allowed.</assert>
 </rule>
 <rule context="xsd:element[not(@ref)]">
  <assert test="@name"
  >@name is required</assert>
  <report test="@type and
             (xsd:complexType or xsd:simpleType)"
  >error: @type and an anonymous type.</report>
 </rule>
</pattern>
```

SchemaPath

```
<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="@type and (xsd:simpleType or xsd:complexType)"
         type="xsd:error" priority="2.5"/>
 <xsd:alt cond="parent::xsd:schema and not(@name)"
         type="xsd:error" priority="2"/>
 <xsd:alt cond="parent::xsd:schema and @ref"
         type="xsd:error" priority="1.5"/>
 <xsd:alt cond="not(parent::xsd:schema) and
              ((@ref and @name) or (not(@ref) and not(@name)))"
         type="xsd:error" priority="1"/>
 <xsd:alt cond="not(parent::xsd:schema) and @ref and
               (@type or xsd:complexType or xsd:simpleType)"
         type="xsd:error"/>
 <xsd:alt type="xsd:element" priority="0"/>
</xsd:element>
```

`<element>`s within `<schema>` produce global element declarations; `<element>`s within `<group>` or `<complexType>` produce either particles which contain global element declarations (if there's a `ref` attribute) or local declarations (otherwise). For complete declarations, top-level or local, the `type` attribute is used when the declaration can use a built-in or pre-declared type definition. Otherwise an anonymous `<simpleType>` or `<complexType>` is provided inline.

The statement can be rephrased as follows: "*an `<element>` must contain a `name` attribute, and may contain either one of `<complexType>` or `<simpleType>`, or a `type` attribute pointing to either a built-in or a pre-declared type definition. A local `<element>` might also have a `ref` attribute, in which case it has no `name` attribute, no `type` attribute and no `<complexType>` or `<simpleType>` content*".

Again, XML Schema and DTDs cannot be used to specify such constraints. Both xlinkit and Schematron declare two rules: one for `<element>`s with a `ref` attribute, and one for `<element>`s without it. However, the XML-based syntax of xlinkit formulae, and the requirements that their boolean operators must be applied to no more than two subformulae, make the xlinkit solution more verbose than with Schematron. RELAX NG defines two patterns and DSD defines two element definitions: one for global elements and the other for local elements. While DSD uses boolean expressions to enforce co-constraints, RELAX NG must resort to purely grammatical expressions, providing a rather involved solution. Finally, in SchemaPath the solution is to write a single type definition for both global and local declarations, where `name`, `type`, and `ref` attributes are optional, and optional also is the anonymous type definition. Then, a condition assigns `xsd:error` to the element whenever a co-constraint is violated, and the correct type otherwise. Our solution is based on the types specified in the official XML Schema for XML Schemas [22].

These solutions are shown in Table 3.

5. IMPLEMENTATION

SchemaPath draws important design decisions from XSLT: for instance, SchemaPath conditions use the same XPath expressions that XSLT accepts as predicates in template patterns, and alternatives of conditional declarations are assigned a priority, just as it is for XSLT templates.

These decisions were not taken by chance: these designs are well known, well understood and highly reasonable, and they greatly simplified the task of choosing the right syntax for our language. But there is one more reason for these decisions, connected to the ease of implementation of a SchemaPath validator.

Implementing from scratch a full-featured SchemaPath validator is a task well beyond the possibilities of our small academic team. This is due not so much on the syntax particularities introduced specifically by SchemaPath, but rather on the complexity of the XML Schema itself, which SchemaPath extends: XML Schema validators are several hundred of thousands lines of code, their implementation involves subtle figuring out of the actual meaning of the W3C standard, and they have been already implemented several times.

Hacking an existing XML Schema validator is also a non

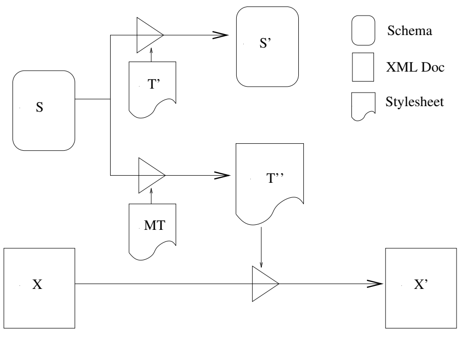

Figure 1: Implementation.

trivial task; although a smaller job than a full implementation, it still requires a deep knowledge of the internals of the existing engine, so that the changes for introducing the support for SchemaPath extensions harmonize with the rest of the code. Furthermore, this would inevitably involve freezing the code supporting XML Schema, and not taking advantage of the new versions of the hacked validator.

Rather, we found out (and, in minimal part, actually designed SchemaPath so that this would hold) that the language allows an easy implementation of its validator as a pre-processor to a plain and standard XML Schema validator.

Just like a Schematron specification really is an XSLT transformation in disguise, our SchemaPath preprocessor is actually based on a couple of XSLT stylesheets, that create a derived XML Schema and a derived XML document that are the ones being used for the actual XML Schema validation.

More precisely, given an XML document X, and a SchemaPath S, we apply two XSLT stylesheets, T' and T'', respectively to S (obtaining a new schema S') and to X (obtaining a new XML document X'); T' and T'' have the property that S validates X in SchemaPath if and only if S' validates X' in XML Schema.

Whereas the stylesheet T' can be applied uniformly to any SchemaPath schema, we need a different stylesheet T'' for each document X. Therefore, T'' is generated on the fly by means of the application of a meta-stylesheet MT to S. Thus the actual architecture of our pre-processor is the one shown in Fig. 1.

Although this implementation can be hardly considered efficient, it works and can be used to test the expressiveness of the SchemaPath language. Furthermore, the implementation is independent of the actual XML Schema validator, and thus can be used in any software architecture that supports both XSLT and XML Schema. The overall procedural part is a couple of dozens line long[4], and can be ported to any programming language in just a few minutes.

Our implementation can be tested on-line at the URL `http://genesispc.cs.unibo.it:3333/schemapath.asp`, and can be downloaded for local tests from the same address. The downloadable package consists of a zip file containing an ASP script, the T' and MT stylesheets, and an XML document and a SchemaPath specification that can be used for testing.

In the next section we give further details on our implementation, explaining the operations performed by the

[4]Excluding the back conversion of the validation errors.

stylesheet T' and the meta-stylesheet MT. Our implementation has some well-known limits: they are explained in Sect. 5.2.

5.1 Transforming the source and the schema

The most important step in the conversion implemented by our SchemaPath processor is the insertion of each conditional element within a new wrapper element manifesting the condition that holds with the highest priority. The name of the wrapper element is obtained combining the name of the element being wrapped, the condition with the highest priority among the conditions the element satisfies, and such highest priority. The wrapper element is in turn inserted within a meta-wrapper, whose name depends only on the conditional element name.

Such a conversion is performed by T'', which is basically an identity stylesheet — that is, a transformation that verbatim copies its input to the output—, but it adds the necessary templates to insert the appropriate wrappers and meta-wrappers around the conditional elements. In particular, there is a template for each alternative of each conditional declaration. The template matches the elements that satisfy the condition specified in the corresponding alternative. The priority of each alternative is copied within the corresponding template.

The transformation T' on the SchemaPath documents is also an identity stylesheet except for a few rules that create the declaration of the meta-wrapper and wrapper elements. Indeed, each conditional element declaration is mapped into a meta-wrapper declaration, whose type is anonymously defined and consists of a choice among wrapper elements (one for each alternative). The type of each wrapper is defined to be a sequence of just one element, whose name is that of the conditional element, and whose type is the one specified in the corresponding alternative.

Since each template in T'' matches a conditional element according to the XSLT priority order, we are sure that the wrapper being inserted is the one corresponding to the actual satisfied condition. Since each wrapper limits the type of the contained element to the one being specified in the corresponding alternative condition, we are sure that only correct pairs of wrappers and element types will be accepted by the XML Schema processor. Thus the XML Schema engine will validate all and only those converted documents that are valid according to the SchemaPath specification. The meta-wrapper is added to handle complex situations arising within `<all>` constructs.

5.2 Implementation limits

Our current implementation has a number of limitations, which are not intrinsic to SchemaPath, but which are consequences of our approach based on XSLT.

The first and most trivial of the limitations concerns the management of namespaces in SchemaPaths whose target namespace is that of SchemaPath itself. Indeed, given the SchemaPath snippet

```
<xsd:element name="alt" type="xsd:alt.type"/>
```

and assuming that the `xsd` prefix is associated with the SchemaPath namespace, T' maps it into an identical element, but declares the `xsd` as being associated with the XML Schema namespace. As a consequence, the `type` attribute references a type definition not present in S' (the target namespace defined in S' remains that of SchemaPath).

However, such a limitation is not severe as the author can always rewrite the schema using two different prefixes for the SchemaPath namespace.

A more severe limitation regards the interactions between local conditional elements with the same name. In theory, homonymous local elements have independent lives, and their conditions should be independent of each others. Unfortunately, our implementation applies global XSLT templates, regardless of the complex types in which the local elements are being defined. As a consequence, conflicting template rules could be generated in T''.

For instance, let us assume that we have two local elements with the same name and different conditions:

```
<xsd:complexType name="aType">
 <xsd:sequence>
  <xsd:element name="quantity">
   <xsd:alt cond="../unit='items'"  type="xsd:integer"/>
   <xsd:alt cond="../unit='meters'" type="xsd:decimal"/>
  </xsd:element>
 </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="anotherType">
 <xsd:sequence>
  <xsd:element name="quantity">
   <xsd:alt cond="../unit"  type="xsd:string"/>
  </xsd:element>
 </xsd:sequence>
</xsd:complexType>
```

The condition of the second `<quantity>` is more general than the conditions of the first `<quantity>`, i.e., each element satisfying one of the alternatives of the first declaration also satisfies that of the second declaration. Since the wrapping templates are global (i.e., they are the same for all the `<quantity>` elements), and since the templates connected to all the conditions in both the first and the second `<quantity>` elements have the same priority, such templates are conflicting, and thus the underlying XSLT processor could signal the error.

Our implementation is able to automatically detect those schemas that could be handled incorrectly due to this limitation, and it notifies the user with a warning message. However, a workaround exists for this limitation, even if it cannot be just as easily implemented, and it does not apply to every situations.

In fact, wherever conditions on other local elements with the same name conflict with the local conditions, new conditions matching the other ones can be inserted locally, repeating the correct type. These new conditions must be identical character by character to the old ones, and not just semantically equivalent XPaths. Moreover, they must have the same priority.

For instance, to have our implementation process correctly the previous example, the definition of the complex type `anotherType` needs to change to:

```
<xsd:complexType name="anotherType">
 <xsd:sequence>
  <xsd:element name="quantity">
   <xsd:alt cond="../unit"          type="xsd:string"
            priority="0"/>
   <xsd:alt cond="../unit='items'"  type="xsd:string"/>
   <xsd:alt cond="../unit='meters'" type="xsd:string"/>
  </xsd:element>
 </xsd:sequence>
</xsd:complexType>
```

With this trick, all the conditions that are local to `anotherType` match those of `aType`, and the correct wrappers are inserted.

6. CONCLUSIONS AND FUTURE WORK

Extending XML Schema has several advantages over inventing a new language, or extending Schematron, or extending any of the other schema languages. First of all, we can exploit the abundant feature set of XML Schema, especially the built-in data types, without needing to reinvent the wheel; secondly, XML Schema is undoubtedly the best-known schema language for XML documents and the language for which the largest number of tools and experience exist, possibly only second to DTDs[5]. Thirdly, since nodes are assigned precise types determined by successful conditions, the SchemaPath validation produces strict PSVIs, a feature not achievable simply embedding Schematron-like assertions within a XML Schema specification. Finally, the implemented validation engine, designed as a pre-processor to an XML Schema validator, shows that implementing a SchemaPath processor is really easy and straightforward. A formal proof of correctness of our implementation (up to the well-known limitations described in Sect 5.2) can be found in in [14].

A challenging future work consists of extending SchemaPath to make conditional attributions survive the validation phase and enter the PSVI, without breaking the good properties inherited from XML Schema (for instance, decidability of sub-typing).

The extension would be particularly interesting, since it would pave the way to experimenting the usage of SchemaPath to type XSLT expressions, directly leading to a dependently-typed typing discipline[6]. A dependent type allows a fine control over the content model returned by a function, whereas only a less precise disjunctive type can be assigned when XML Schema types are the only types available.

7. REFERENCES

[1] XML Schemas: Best Practices. http://www.xfront.com/BestPracticesHomepage.html.

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

[3] The DSDL project. http://www.dsdl.org/.

[4] N. Amorosi, N. Gessa, and F. Vitali. Datatype- and namespace-aware DTDs: a minimal extension. In *Proceedings of the Extreme Markup Conference*, 2003.

[5] P. V. Biron and A. Malhotra. *XML Schema Part 2: Datatypes.* http://www.w3.org/TR/xmlschema-2. W3C Recommendation, May 2001.

[6] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. *Extensible Markup Language (XML) 1.0 (Second Edition).* http://www.w3.org/TR/REC-xml. W3C Recommendation, October 2000.

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

[8] J. Clark. TREX - Tree Regular Expressions for XML. http://www.thaiopensource.com/trex, 2001.

[9] J. Clark and M. Murata. RELAX NG. http://relaxng.org, 2001.

[10] S. DeRose, E. Maler, and D. Orchard. XML Linking Language (XLink) Version 1.0. http://www.w3.org/TR/XLink. W3C Recommendation, June 2001.

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

[12] R. Jelliffe. Schematron. http://www.ascc.net/xml/resource/schematron/, October 2002.

[13] N. Klarlund, A. Møller, and M. I. Schwartzbach. DSD: A schema language for XML. In *Proceedings of the third workshop on Formal methods in software practice*, Portland, 2000.

[14] P. Marinelli, C. Sacerdoti Coen, and F. Vitali. A Formal Semantics for SchemaPath. Technical report, University of Bologna, 2004. To be published.

[15] S. Muench and M. Scardina. *XSLT Requirements Version 2.0.* http://www.w3.org/TR/xslt20req, 2001.

[16] M. Murata. RELAX (REgular LAnguage description for XML). http://www.xml.gr.jp/relax, 2000.

[17] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: A Consistency Checking and Smart Link Generation Service. In *ACM Transaction on Internet Technology*, May 2002.

[18] E. Robertsson. Combining Schematron with other XML Schema Languages. http://www.topologi.com/public/Schtrn_XSD/Paper.html.

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

[20] J. Siméon and P. Wadler. The Essence of XML. In *Proceedings of the 30th ACML SIGPLAN Symposium on Principles of Programming Languages*, New Orleans, January 2003.

[21] C. M. Sperberg-McQueen. Context-sensitive rules in XML Schema. Not published, 2000.

[22] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema Part 1: Structures.* http://www.w3.org/TR/xmlschema-1/. W3C Recommendation, May 2001.

[23] N. Walsh and J. Cowan. Schema Language Comparison. http://nwalsh.com/xml2001/schematownhall/slides/, December 2001.

[24] H. Xi and F. Pfenning. Dependent types in practical programming. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 214–227, New York, NY, 1999.

[5]In [4] we also propose an extension to DTDs to make them as expressive as XML Schema.

[6]In a dependently-typed programming language, the output type of a function depends on the actual value of the input. For instance, the function `f(x:int) { if x = 0 then 0 else true }` is assigned the admissible type `if x = 0 then int else bool`. See for instance [24] for a practical dependently-typed functional programming language that allows strict and statically checked control over program invariants.