SchemaPath: Extending XML Schema for Co-Constraints

Paolo Marinelli    Claudio Sacerdoti Coen    Fabio Vitali

May 2004

Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7
40127 Bologna (Italy)
The University of Bologna Department of Computer Science Research Technical Reports, organized by year, are available in PDF and gzipped PostScript formats via anonymous FTP from the area ftp.cs.unibo.it:/pub/TR/UBLCS or via WWW at URL http://www.cs.unibo.it/. Plain-text abstracts organized by year are available in the directory ABSTRACTS. All local authors can be reached via e-mail at the address last-name[at]cs.unibo.it.

Recent Titles from the UBLCS Technical Report Series


2003-16  Robust Aggregation Protocols for Large-Scale Overlay Networks, Montresor, A., Jelasity, M., Babaoglu, O., December 2003.


SchemaPath: Extending XML Schema for Co-Constrains

Paolo Marinelli¹ Claudio Sacerdoti Coen¹ Fabio Vitali¹

Technical Report
May 2004

¹. Department of Computer Science, University of Bologna, Mura Anteo Zamboni 7, 40127 Bologna, Italy.
Abstract

In this paper we propose a new schema language for XML, SchemaPath. SchemaPath is a conservative extension to XML Schema for the support of co-constraints.

Validating is the process of verifying whether an XML document satisfies or not a set of structural and content rules. Validation brings many benefits, and it is necessary in several contexts and scenarios.

For instance, XML is used as interchange format between applications. In all likelihood each application has its own internal data format, and when it has to send data to another application, the information is converted into XML and then sent to the receiving application, which convert XML data in its internal format before processing it. It is important that XML data is validated before being converted and processed by the receiving application.

Often, validity rules for XML documents are collected and formally expressed within a schema. There are several languages to define schemas, and they are known as schema languages or validation languages. The usage of schema languages to declare XML validity rules brings numerous advantages. Indeed, they allow to define such rules in an explicit manner, using few and especially provided constructs, and they make the validation process platform-independent. Moreover, a schema allows to define validity rules in a formal and precise manner. Without a precise formalism, rules should be expressed in natural language, arising ambiguity and interpretation questions.

As aforementioned, validation languages are numerous, and each has its pros and cons. Roughly speaking, they can be seen as belonging to one of two categories: grammar-based languages and rule-based languages. The first ones describe the logical structure of XML documents through the definition of a tree grammar. An XML document is valid if it conforms to the defined grammar. On the other hand, rule-based languages define a set of validity predicates, expressed in a particular formalism. An XML document is valid if it satisfies all the predicates defined within the schema. Among grammar-based languages, the most important ones are: DTD, XML Schema, RELAX NG and DSD; while among rule-based languages we find Schematron and xlinkit.

DTDs are defined by the W3C and they represent the first XML schema language. Although they allow to compactly and simply define a tree grammar, being unable to impose complex constraints on attribute and text element values, they are not well suited in many situations. Furthermore, they are not able to define co-constraints (or co-occurrence constraints), i.e., those constraints describing the dependency among an element content and the value of other elements or attributes.

Due to the aforementioned limitations, other validation languages have been created, among which is the other W3C proposal, XML Schema, that is now recognized as the official XML schema language. XML Schema has several improvements over DTDs, such as a vast set of predefined types and a flexible mechanism to build new types upon existing ones. However, XML Schema does not provide any support for co-constraints definition, that, on the other hand, are necessary in several real situations. Actually, several domain-specific standards include laments that DTDs and XML Schema, the official schema languages for XML, do not allow to express co-constraints.

As aforementioned, the number of validation languages alternative to DTDs is vast. Among grammar-based ones, we find RELAX NG and DSD. Undoubtedly, these languages have some peculiarities making them interesting, such as a form of co-constraints support. However, we cannot claim they are better than XML Schema under all point of views, and least of all they represent the solution to all problems concerning the XML validation. For instance, RELAX NG does not provide any mechanism to specify default values, while DSD has no concept of predefined type. Furthermore, their expressive power is not sufficient to impose many co-constraints.

In general, we can claim grammar-based languages are not well suited to define co-constraints. On the other hand, rule-based languages allow to straightforwardly define this type of constraints. However, rule-based languages are not able to express structural constraints and constraints on attribute and text element values as so straightforwardly, for whose definition grammar-based languages are much more suited.

It has been proposed to embed Schematron rules within an XML Schema schema, obtaining a single specification able to express both constraints on the logical structure and text values (through XML Schema constructs) and co-constraints (through Schematron constructs). However, it surely not is an elegant solution, and, above all, it has problems arising from the low level of integration of the combined languages.

In this paper, we propose a new language, SchemaPath. SchemaPath is conservative extension to
XML Schema for the support of co-constraints. The extension is obtained adding conditional type assignment to elements and attributes, and adding one new built-in simple type, `xsd:error`, i.e., an unsatisfiable type. In SchemaPath, elements and attributes can be conditionally declared, i.e., they can be assigned one among a set of type definitions. The type definition to be assigned is chosen by evaluating on the instance document conditions specified by means of XPath expressions.

On a syntactic level, conditional type assignment is achieved adding just one new construct to plain and standard XML Schema element and attribute declarations: the `<xsd:alt>` element. This element specifies a type definition and the associated XPath condition.

For instance, the following conditional element declaration,

```xml
<xsd:element name="item">
  <xsd:alt cond="type='CL'" type="clothingItem"/>
  <xsd:alt cond="type='NC'" type="nonclothingItem"/>
</xsd:element>
```

assigns the `clothingItem` type to `<item>` elements when the child `<type>` value is "CL", and the type `nonclothingItem` when the child `<type>` value is "NC".

As aforementioned, SchemaPath is a conservative extension to XML Schema. It means that any of the XML Schema features is not absent in SchemaPath. It also means that any XML Schema specification is also a SchemaPath specification. In particular, the large set of built-in data type and the refined mechanism for their derivation are directly present in SchemaPath. Moreover, just as it is in XML Schema, the SchemaPath validation process generates a PSVI (Post Schema Validation Infoset), i.e., the set of information associated to the instance document once it has been validated.

Furthermore, conditional type assignments allow to compactly and elegantly express a large class of co-constraints, such as mutual exclusion among attributes, and many others. Thus, the main SchemaPath advantage is to be able to express both grammatical constraints (using standard XML Schema constructs), and co-constraints (using conditional declarations), and it allows to express them in a straightforward manner.

A peculiar feature of SchemaPath is in the easy of implementation of its validator. Indeed, the language syntax and semantics allow it to be implemented using a pair of XSLT stylesheets: given a SchemaPath schema $S$ and an XML document $X$, it is possible to obtain, through XSLT, a derived document $X'$ and a derived XML Schema schema $S'$, such that $X$ validates against $S$ in SchemaPath if and only if $X'$ validates against $S'$ in XML Schema. Thus, the validator does not depend on any programming language, and it can be adopted in any system supporting both XML Schema and XSLT.

The SchemaPath proposal has been already appreciated at international level. In [MSV04a] we provide a paper describing the main concepts explained here. [MSV04a] has been accepted by the Thirteenth International World Wide Conference, held at New York in May 2004.
Chapter 1

Introduction

1 The Origins of XML

Extensible Markup Language (XML) describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them. XML is a restricted form of SGML (Standard Generalized Markup Language), its immediate ancestor as a meta-markup language.


The main goal of XML is to overcome some weaknesses of HTML, keeping at the same time a low level of complexity. In XML, document authors can use tags of their choice, and thus they can represent information in a way that is completely independent of how the information will be processed. By the application of style sheets, a Web browser can choose the most appropriate way to display (or more generally, to present) an XML document to the user. Moreover, XML (re)introduces the concept of validity. Indeed, it is possible, within a schema, to declare a set of rules constraining the structure of an XML document. As we shall see in the following section, there are many scenarios where XML validation plays a crucial role, and where it is important to express declaratively a set of validity rules using a schema language.

2 The Role of Validation and Schema Languages

In XML there are two level of correctness: well-formedness and validity. An XML document is well-formed if it complies with a set of syntactical rules mainly ensuring that it has a rigorous tree logical-structure. On the other hand, a well-formed document is valid if it accords to a set of structural and content rules. XML validation is the process of verifying whether an XML document conforms to such a set of rules. It has several benefits, and it is necessary in different contexts and scenarios.

- In creating a language, syntax rules have to be defined, and they have to be respected by all terms of the language.
- Documents in an information system share a common structure, and are constrained by the same set of rules. It is important to assure that they actually conform to such rules and structure, because this aids in writing programs that process them, and in creating style sheets for their presentation. In fact, once documents have been validated, a lot of assumptions can be made on their structure and data values, and thus error-handling code may be avoided.
- XML can be used as an intermediate exchange format between applications. In all likelihood, each of them has its own internal format for data representation. In order to be transferred from an application to another, data is often converted from an application-specific
format to an intermediate XML format. The receiving application, once obtained the XML data, has to validate it before processing it and converting it into its internal format. Such a general framework still holds when XML data is exchanged between databases.

- All the applications requiring user input need to validate the inserted input before processing it. Sometimes, such input is collected as XML data, and thus validation has to be performed on it.

Often, syntax rules for XML are collected and formally expressed within a *schema*. There are languages allowing the definition of schemas using a specific formalism, and they are known as *schema languages*. There are several schema languages, using a non-proprietary format (usually XML). Using schema languages to declare explicit validity rules brings a number of advantages. In fact, they allow to express those rules using few and specially provided constructs, and make the validation process platform-independent. In their absence, one should turn to programming languages, which, although allowing to define virtually all needed syntactical constraints, introduce some well-known disadvantages: the need to write error prone processing code, difficulties with reuse and maintenance, programming language dependence and obfuscation of the validation constraints as they are spread throughout the codebase.

Moreover, as highlighted in [MM99], schemas can also be used in specific scenarios. For instance:

- In a typical scenario, a user community would agree on a common schema and on producing XML documents which are valid with respect to the specified schema.

- One important class of applications uses a schema definition to guide an author in the development of documents. The application can ensure that the author always knows whether to enter, for instance, a date or a part-number, and might even ensure that the data entered is valid.

- A query interface inspect schemas to guide a user in the formulation of queries. Any given database can emit a schema of itself to inform other systems what counts as legitimate and useful queries.

As aforementioned, there are many schema languages, and they are 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 [BPSMM00], and which is a direct derivation of its counterpart in SGML.

The principal advantage of DTDs is that they are supported by every validating XML 1.0 parser. Also, they have well understood and agreed upon semantics, and they are compact. Unfortunately, for many modern applications, their advantages are outweighed by their disadvantages. Indeed, although they provide simple constructs to declare structural requirements (the most important ones in the publishing domain), data exchange applications may want to make sure that attributes and text nodes have correct values, and DTDs provide little support for this kind of constraints (they define just a few datatypes and just for attributes).

In order to overcome the limitations of DTDs, numerous schema languages for XML have been developed. In [cov03], a fairly authoritative source, 15 different schema languages for XML are listed besides DTDs, and at least one more, xlinkit [NCEF02], is missing. Among these languages is XML Schema, which is directly backed by the W3C, and defined in the W3C recommendations [TBMM01, BM01]. XML Schema has been developed to be more expressive than DTDs, and to replace them as the de facto standard schema language for XML documents. ISO, on the hand is active in DSDL (*Document Schema Definition Languages*) [DSD], a project whose aims are to create a framework within which, using different schema languages, multiple validation tasks of different types can be applied to an XML document. Under this view, a couple of schema languages, RELAX NG [CM01a] and Schematron [Jel02], are standardized by ISO. Although being absolutely ignored by the W3C, both RELAX NG and Schematron are having a fair success.

Roughly speaking, schema languages can be seen as belonging to one of two types:
• **grammar-based languages**, by which document engineers create a whole tree grammar according to top-down production rules in a specific formalism. Commonly, expressions constraining elements content are called *types*, and they roughly match *non-terminals* of automata theory. 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 forbidden is allowed) or a closed specification (all that is not allowed is forbidden) [WC01]. 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 support character entities, XML Schema has a rich set of predefined datatypes 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 strengths and weaknesses. A final report was issued, in the form of a set of slides [WC01]. 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 the values of other attributes and elements, such as 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 datatypes and no way to specify default values.

Generally, rule-based languages allow to easily formalize co-constraints, but are not able to impose complex constraints on datavalues, and appear limited in structural constraints definition. On the other hand, grammar-based languages are well-suited to express structural constraints, and most of them provide rather powerful mechanisms to impose constraints on datavalues. However, co-constraints represent a weak point for such languages, which provide a limited support for their definition.

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 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.
For instance, FpML (Financial Products Markup Language) is a markup language for financial derivatives trades. Although an official XML Schema for FpML 4.0 documents is present, it is not able to capture a large range of constraints, among which a number of co-constraints. For this reason, such additional requirements are normatively 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 [ea00], 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. Appendix A of [TBMM01] presents the schema for schemas (an XML Schema schema specifying the structure and content of XML Schema documents) as a normative part of the specification. This means that, in order to be a correct XML Schema document, a schema has to validate against the schema for schemas. However, there is a number of normative additional constraints which are imposed and explained in natural language throughout the specification and that cannot be expressed by XML Schema. For instance, ref and name attributes are mutually incompatible in an element or attribute declaration; in an attribute declaration, if the default and use attributes are both present, use must have value "optional"; in an element declaration, attribute type and either <simpleType> or <complexType> child elements are mutually exclusive.

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 pre-specified 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 [Rob, com] 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, such a proposal requires the learning of two completely different languages, and presents problems concerning their interaction.

3  Aims and Paper Organization

In this paper our thesis is that it is possible to introduce, in grammar-based schema languages, conditional type assignments, i.e., elements and attributes are assigned one among a set of alternative types, according to values of the instance document.

In this way, a conservative extension of the grammar-based language is defined, which allows the specification of a sufficiently large class of co-constraints.

In particular, we state that it is possible to extend a specific schema language, XML Schema, introducing conditional declarations, i.e., declarations associating to attributes and elements one among a set of alternative type definitions, according to XPath predicates evaluated on the instance document. Such an extension is called SchemaPath [MSV04a], and we argue that it is a conservative extension to XML Schema (i.e., every XML Schema schema is also a SchemaPath schema) and it allows the definition of a sufficiently large class of co-constraints.

The choice of XML Schema as grammar-based language to extend is not taken by chance. XML Schema offers the most sophisticated mechanism to derive types (especially simple types), and it provides the most complete set of built-in datatypes. It is undoubtedly the best-known schema language for XML documents and the language for which the largest number of tools and

\(^1\) This is technically considered an exclusion, rather than a co-constraint, but there is only a very little difference.
experience exists, possibly only second to DTDs. XML Schema is directly backed by the W3C, and it is the only and official schema language for XML. A couple of W3C functional languages, XPath 2.0 \cite{BBC03} and XQuery \cite{BCF03}, directly relies on XML Schema for their type systems. Thus, extending XML Schema introducing dependent type assignments immediately arises interesting questions concerning the interaction between those functional languages and our extension type system.

Also the use of XPath as language to express conditions is motivated by deep reasons. Like XML Schema, it is a W3C recommendation \cite{CD99}, and its success and widespread use are undisputedly acknowledged. It actually allows to express rather complex paths on XML documents, through tests on element, attribute, namespace, and text nodes, equality and inequality operators between node-sets and datavalues, and a number of useful functions operating on strings, numbers, booleans and node-sets. Moreover, XPath syntax is not XML-based, and thus conditions on type assignments may be expressed in a compact and readable manner.

In order to prove our thesis, in the next chapter we provide a detailed description of the most relevant grammar-based and rule-based schema languages currently available. Such a description shows their strengths and weaknesses, and it especially highlights that grammar-based languages are better suited to express structural and content constraints than co-constraints, while for rule-based ones the contrary holds.

The proof goes on informally introducing SchemaPath syntax and semantics in Chapter 3. Moreover, Chapter 3 itself provides numerous real-world examples of constraints defined by SchemaPath specifications, showing the expressiveness, flexibility, and usefulness of the language.

The proof ends with Chapter 4, that shows a SchemaPath implementation, and thus demonstrates that the extension we propose is feasible. The implementation we describe has a value of its own. Indeed, it is based on two relatively simple XSLT stylesheets and a XML Schema processor. In a few words, in order to validate an XML instance document against a SchemaPath schema, two stylesheets are applied to them, obtaining a derived XML document and a derived XML Schema schema. The resulting XML document and the resulting XML Schema schema are constructed so that the former validates against the latter in XML Schema if and only if the original instance document validates against the original schema in SchemaPath.

Finally, Chapter 5 summarizes the statement and its proof presented in this paper, and draws development lines SchemaPath will follow for the future.
Chapter 2

Schema Languages for XML

In this chapter, 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. In particular, we examine DTDs, XML Schema, RELAX NG, DSD, Schematron and xlinkit.

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.

1 DTDs

DTDs have been originally introduced for validating SGML structures, and then ported to provide validation for XML documents. They represent the first schema language for XML, and are defined by the W3C within the XML 1.0 recommendation itself [BPSMM00]. XML DTDs are very similar to those for SGML. Indeed, except for some details, every XML DTD is also an SGML DTD, although the contrary does not hold (SGML DTDs have some features XML ones lack).

Essentially, a DTD is a sequence of element type declarations, (to constrain element contents), attribute-list declarations (to constrain the attributes which may appear within an element), and entity declarations (to define reusable characters sequences).

1.1 Element Type Declarations

Element type declarations represent the main construct in DTDs. They consist of a name (which must be unique among all the element type declarations) and a content specification constraining the element content. An element of the instance document is valid if there is an element type declaration with the same name, and whose content specification is matched by its content.

There are four kinds of content specifications. The first one, called content model, is used for elements which contain just child elements (no character other than whitespace is allowed). It is a simple grammar governing the allowed types of the child elements and the order in which they are allowed to appear. The grammar is an expression over element type names, using choice, sequence, and repetition standard operators. Each subexpression in the grammar is called content particle.

It is required that content models in element type declarations are deterministic, i.e., it is an error if an element in the document can match more than one occurrence of an element type in the content model.

Another kind of content specification is the mixed content declaration, used to constrain mixed contents, i.e., contents where child elements are mixed with text. A mixed content declaration is a particular content model, where the \#PCDATA (Parsed Character Data) content particle (matched by text nodes) has to be used. However, all mixed content declarations have to comply with severe limitations. Indeed, they must consist of a repetition operator applied to a choice among element type names and \#PCDATA. Moreover, exactly one \#PCDATA must appear.
The last two kinds of content specifications serve to specify an empty content (through the \texttt{EMPTY} expression), and any content (through the \texttt{ANY} expression). In order to be valid against a type declaration whose content specification is \texttt{ANY}, an element must only contain child elements whose type has been declared within the DTD.

Since element type declarations are uniquely identified by their names, and each element in an instance document uniquely determines an element type declaration by its name, it is not possible to subject the content of an element to its context. For instance, it is not possible to require that the \texttt{<x>} element contains just text when its parent is \texttt{<p1>}, and contains a \texttt{<y>} child when its parent is \texttt{<p2>}.

To illustrate, we provide a brief DTD fragment showing element type declarations:

\begin{verbatim}
<!ELEMENT x (w | (y, z)) >
<!ELEMENT w (#PCDATA | y)* >
<!ELEMENT y EMPTY >
<!ELEMENT z (s+, t?) >
\end{verbatim}

It declares the \texttt{x} element type whose content model is a choice among the \texttt{w} type, and the sequence of the \texttt{y} and \texttt{z} types. The \texttt{w} type is matched by any sequence of \texttt{<y>} elements mixed with text, while the \texttt{y} type requires an empty content. Finally, the \texttt{z} type is satisfied by one or more child elements of type \texttt{s}, optionally followed by an element of type \texttt{t}. Both \texttt{s} and \texttt{t} types are defined elsewhere.

### 1.2 Attribute-List Declarations

Attribute-list declarations are used to define the set of attributes pertaining to a given element type, to establish type constraints for these attributes, and to provide them default values.

To illustrate the syntax and semantics of attribute-list declarations, an example is given:

\begin{verbatim}
<!ATTLIST x
  a (v1 | v2) "v1"
  b CDATA #REQUIRED
  c NMTOKEN #IMPLIED
  d CDATA #FIXED "v3" >
\end{verbatim}

It imposes that each element of type \texttt{x} must comply with the following constraints:

- An \texttt{a} attribute may optionally appear. If present, its value must be either "\texttt{v1}" or "\texttt{v2}"; otherwise it defaults to "\texttt{v1}".
- A \texttt{b} attribute must occur, and its value may be any string.
- A \texttt{c} attribute may optionally appear, and its value must be an alphanumeric string where no whitespace characters nor many punctuation marks may occur. No default value is provided.
- A \texttt{d} attribute may optionally appear. If present, its value must be "\texttt{v3}"; otherwise it defaults to "\texttt{v3}".
- No other attribute may occur.

Default values can be provided just for attributes, not also for element types.

### 1.3 Datatypes and Linking

In DTDs, it is not possible to impose constraints on text nodes appearing within mixed contents. Indeed, only the \texttt{#PCDATA} type is provided, which is matched by any string.

On the other hand, datatypes are provided for attributes. As previously seen, DTD authors may explicitly enumerate which strings are allowed for an attribute (the first attribute definition in the example), or they may use a predefined type (\texttt{CDATA}, \texttt{NMTOKEN}, and few others for a sum of 8 predefined types).
Among such predefined types, there are two ones used to establish links between attributes (eventually specified within different elements) of the instance document. Indeed, within an attribute-list declaration, an attribute can be defined to be of type \texttt{ID}. In an XML document there must not be more than one attribute of type \texttt{ID} with the same value. Moreover, an attribute can be defined to be of type \texttt{IDREF}, requiring the value of such an attribute to match that of another attribute whose type is \texttt{ID}. \texttt{ID} and \texttt{IDREF} types obviously share the same set of allowed strings. Finally, an attribute can be defined to be of type \texttt{IDREFS}, requiring its value to be a whitespace-separated list of values of type \texttt{IDREF} (i.e., each value must match that of an attribute of type \texttt{ID}).

### 1.4 Entity Declarations

Within an XML document, \textit{general entities} may be referenced. They associate a text to a name, and in order to be used they must be declared within the DTD of the document. An example of general entity declaration follows:

```xml
<!ENTITY dtd "Document Type Definition" >
```

General entities cannot be referenced within the DTD itself. However, for such a purpose \textit{parameter entities} are provided. They are very useful in writing large DTDs, where, for example, a number of content particles has to be defined in several element type declarations, or where many element types share a common set of attributes.

For instance, if many type declarations make use of the content particle \((y \mid w)\), it is useful to declare the following parameter entity:

```xml
<!ENTITY % yOrw "y \mid w" >
```

Then, it can be referenced as follows:

```xml
<!ELEMENT x (z \mid %yOrw;)* >
```

### 1.5 Namespaces

Since DTDs precede temporally the advent of namespaces, they provide no support for qualified elements and attributes, although by fixing the prefix associated to a namespace, as it is shown below:

```xml
<!ATTLIST p:x
  xmlns:x CDATA #FIXED "http://www.example.com">
```

### 1.6 Co-Constraints Support

DTDs provide no support for co-constraint definitions. Indeed, there is no way to subject content models (or other kinds of content specifications) to the presence or value of attributes or other elements.

However, SGML DTDs (richer and more complex than XML ones) have a feature that would be of wide interest in our discussion: \textit{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. In a way, exclusions represent one kind of co-constraint, the only possible with DTDs (and only SGML DTDs, by the way!).

### 2 XML Schema

XML Schema is a W3C recommendation [TBMM01, BM01] 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.
Roughly, an XML Schema schema consists of type definitions, element declarations, and attribute declarations. In brief, element and attribute declarations are associations of a name with a type definition. Among all schema languages described in this paper, XML Schema is the only one having a named approach to typing, while the others have a structural approach. As observed in [JS03], also DTDs have a named approach to typing, but they are so restricted that the structural and named approaches might be considered to coincide.

2.1 Type Definitions

Types in XML Schema are either simple or complex, and are assigned to elements and attributes to constrain their values. Types are either named (and referred to via their name) or anonymous (and inserted inline within the relevant element and attribute declarations).

A simple type is a set of string values (the so called value space), and can be assigned to elements whose content is just text, as well as to attributes. A large number of built-in simple types are provided, ranging from integers to dates, times, URIs, etc. New simple types may be defined by deriving existing ones.

Complex types are used to constrain elements containing child elements and/or attributes. A complex type is constructed by means of an expression over element declarations, similar to those in DTDs. Besides the operators provided by DTDs, XML Schema allows to explicitly control the number of repetition of content particles (through the minOccurs and maxOccurs attributes specification). Moreover, it reintroduces unordered content models (although with some limitations), a feature of SGML DTDs removed in XML ones.

Complex types can also be defined deriving existing simple or complex ones. There are two derivation methods: by restriction and by extension. When the base type is simple, the only allowed method is by extension, and is used to construct types whose content is simple, but where attributes are declared.

On the other hand, in defining a type as an extension of a base complex type, one declares a content expression and attributes. The new type is then treated as if it was defined applying a sequence operator to the base type content expression, and to the newly declared one. In other words, the newly declared content is “appended” to that declared within the base type. Furthermore, the new type is treated as it was defined declaring both attributes within the base type and those within it.

Finally, XML Schema allows to derive complex types by restriction. A restricted complex type allows less values than those allowed by the base type. The restricted type has to be entirely defined, heeding to not construct it so that it allows values not accepted by the base type.

The type system defined by an XML Schema schema is inspired by those of object-oriented programming languages. Indeed, each type (explicitly or implicitly) derives from a base one, thus establishing a type definition hierarchy.

2.2 Datatypes

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. Indeed, XML Schema provides 19 primitive built-in datatypes, and other 25 derived ones (also comprising all the datatypes of DTDs), all of which are defined in [BM01].

Although such types are directly usable in a vast spectrum of situations, each application needs to impose its own constraints on data values, which could not be precisely captured by any of the predefined types. For this reason, XML Schema allows to define new simple types, deriving existing ones. There are three derivation methods for simple types: by list, by union, and by restriction. The value space of a list-derived type is a whitespace-separated list of items, each belonging to the base type. The value space of a union-derived type is the union of the value spaces of two or more existing types. Finally, the value space of a simple type derived by restriction is

1. A more detailed discussion on simple types and their derivation methods can be found in the next subsection.
2. A type not explicitly derived by an existing one, is assumed to be derived by restriction from xsd:anyType, which is the root of the type definitions hierarchy of all schemas, and allows any (both complex and simple) value.
a subset of the base type one. The restriction is achieved through the application of one or more facets. For instance, a schema author can make explicit the allowed values by enumeration; he or she can identify a range of allowed values through the specification of minimum and/or maximum values (only for types where an order is defined); or even he or she can specify a regular expression (in a perl-like notation) constraining the value space of the derived type.

2.3 Element and Attribute Declarations

Element and attribute declarations are associations of a name with a type. A declaration is either global (at top-level) or local (within a complex type definition). Such a distinction allows schema authors to declare elements (attributes) with the same name but different types, provided that they appear in different contexts (i.e., they do not appear within the same complex type). Global declarations can also be referred to from complex type definitions. A schema snippet illustrating the main constructs described follows:

```xml
<xsd:element name="x">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="x" type="xsd:string"/>
      <xsd:choice maxOccurs="8">
        <xsd:element name="w" type="xsd:integer"/>
        <xsd:element name="y"/>
      </xsd:choice>
    </xsd:sequence>
    <xsd:attributeref="a" use="required"/>
  </xsd:complexType>
</xsd:element>

<xsd:attribute name="a">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="v1"/>
      <xsd:enumeration value="v2"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:attribute>
```

XML Schema also extends DTDs over default value specifications. Indeed, while DTDs allow default and fixed values just for attributes, in XML Schema a schema author can specify a default or fixed value also within element declarations. However, while default and fixed values apply to missing attributes (just as it is in DTDs), they apply to empty elements. Furthermore, the type assigned to the element is required to be simple.

2.4 Linking

DTDs provide a very weak linking mechanism. Indeed, an attribute of type ID is required to be unique with respect to the whole document. Furthermore, an attribute of type IDREF has to point to any other attribute of type ID. But there are many situations where attributes or elements have to be unique only among a set of other attributes and elements.

XML Schema provides a much finer linking mechanism. In order to specify uniqueness, the `<xsd:unique>` element is used within an element declaration. It first selects, by means of a limited XPath expression, a set of descendant elements, and then identifies the attribute or element “field” relative to each selected element that has to be unique within the scope of the set of selected elements.
Furhtermore, to require a field to be referenced, it has to be defined as a key using the \texttt{<xsd:key>} element. It acts as \texttt{<xsd:unique>}, but it also associates the key with a name. Then, a \texttt{<xsd:keyref>} element is used to specify (always through the scope and field selection) which field has to refer to the key.

2.5 Namespaces and Modularity

One of the major flaws of DTDs is the absence of any namespace support mechanism. Under this point, XML Schema highly improves over DTDs, providing a fine and flexible support. Each schema may optionally specify a target namespace. In this case, globally declared elements and attributes must be qualified and associated to the target namespace. On the other hand, each locally declared element or attribute must either be qualified (and associated to the target namespace) or unqualified.

When a document contains elements or attributes from multiple namespaces, more than one schema is required, each declaring all the elements and attributes associated to a given namespace. Then, if a schema needs, for instance, an external element declaration, it imports the schema containing the needed declaration which can be then normally referenced.

Indeed, an XML Schema schema may include and import one or more external ones, and even redefine part of them. The import mechanism, as aforementioned, is used to gain access to external definitions of schemas whose target namespace does not match that of the importing one. The include mechanism also allows to gain access to external definitions, but it requires that both the including and the included schemas have the same target namespace. Finally, the redefine mechanism is equivalent to the include one, but it allows included schema components to be redefined through the derivation methods previously described.

2.6 Post Schema Validation Infoset

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. Such augmentation makes explicit information which may have been implicit in the original document, such as normalized and/or default values for attributes and elements and the types of element and attribute nodes.

2.7 Co-Constraints Support

Although, as we have discussed in this section, XML Schema overcomes many limitations of DTDs, it does not add any support for co-constraints. Its grammar-based approach to validation makes it impossible to define any type of inter-dependencies among attributes and elements, and attributes and attributes.

However, as observed in the W3C Note [SM00], SGML exclusions may technically be formalized, but this would involve duplicating a large part of the specification, creating two subschemata (one with and one without the element to exclude) to be used outside and within the outermost excluded element. Of course, this rapidly leads to really unmanageable specifications, given their size and complexity.

3 RELAX NG

RELAX NG is a schema language for XML developed by an international working group, ISO/IEC JTC1/SC34/WG1, and it was published by ISO as an International Standard on 1st December 2003 [CM01b]. It is based on two preceding languages, TREX (Tree Regular Expressions for XML) [Cla01], designed by James Clark, and RELAX (Regular Language description for XML) [Mur00], designed by Murata Makoto.

A RELAX NG schema specifies a pattern for the structure and content of an XML document, thus identifying a class of XML documents consisting of those matching the pattern.

As discussed in [HM02], the main advantage of RELAX NG over XML Schema and DTDs is surely the inclusion of elements and attributes in a single regular expression (the so called
attribute-element constraints), which, as we shall see, also gives, even if in a limited way, some help on co-constraint definitions.

3.1 Simple Patterns
A pattern is a regular expression over elements, attributes and text nodes, and its syntax is XML-based. It uses standard operators to specify repetition and optionality of sub-patterns. A simple schema showing the basic features of RELAX NG follows:

```
<element name="x" xmlns="http://relaxng.org/ns/structure/1.0">
  <optional>
    <attribute name="a">
      <text/>
    </attribute>
  </optional>
  <element name="y">
    <attribute name="b">
      <choice>
        <value>v1</value>
        <value>v2</value>
      </choice>
    </attribute>
  </element>
  <element name="y">
    <oneOrMore>
      <choice>
        <element name="s">
          <empty/>
        </element>
        <element name="t">
          <text/>
        </element>
      </choice>
    </oneOrMore>
  </element>
</element>
```

It is matched by all documents whose root is `<x>`. Such a root may optionally have an `a` attribute whose value can be any string. The content of the root must consist of two `<y>` child elements. The former must have a `b` attribute whose value is either "v1" or "v2", and it also must be empty. The latter must contain a sequence of one or more `<s>` and `<t>` elements in any order. `<s>` must be empty, while `<t>` must just contain text.

As showed in the example above, RELAX NG allows ambiguous patterns, i.e., it allows patterns requiring the presence of two child elements with the same name and different patterns for their content.

Another advantage of RELAX NG over XML Schema is represented by the `<interleave>` operator, which serves to specify unordered content, and which can be used almost without limitations.

3.2 Named Patterns
For a non-trivial RELAX NG pattern, it is often convenient to be able to give names to parts of the pattern. For this purpose RELAX NG provides the `<grammar>` pattern, which is, as its name suggests, a grammar rather than a regular expression. A `<grammar>` element has a single `<start>` child element, and zero or more `<define>` child elements. The `<start>` and `<define>` elements contain patterns. These patterns can contain `<ref>` elements that refer to patterns defined
by any of the `<define>` elements. A `<grammar>` pattern is matched by a document matching the pattern contained in the `<start>` element. An example follows:

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <start>
    <ref name="X"/>
  </start>
  <define name="X">
    <element name="x">
      <ref name="Y"/>
    </element>
  </define>
  <define name="Y">
    <element name="y">
      <choice>
        <empty/>
        <ref name="X"/>
      </choice>
    </element>
  </define>
</grammar>
```

It is matched by all documents whose root is `<x>`. Such a root must contain a `<y>` child element which has to either be empty or contain a `<x>` child element, which must contain a `<y>` element, which has to either be empty or contain a `<x>` child element, and so on.

### 3.3 Datatypes

RELAX NG does not provide any datatype, however it allows patterns to reference externally-defined datatypes. The most commonly used ones are those defined by [BM01]. The library of datatypes being used is identified by the `datatypeLibrary` attribute, while the `<data>` pattern matches a string that represents a value of a named datatype.

`<data>` patterns may have parameters, and each one has its own set of applicable parameters, and in case of the XML Schema datatypes, they correspond to the set of facets for that datatype defined in [BM01].

Patterns can be constructed over `<data>`s, and thus schema authors are allowed to create their own datatypes. In particular, the `<choice>` operator may be used to define a union of datatypes, the `<list>` operator to define a pattern matched by whitespace-separated lists of tokens, and, through parameters, the value space of a datatypes can be restricted. Then, using the `<grammar>` pattern, it is possible to give names to user-defined datatypes. To illustrate, a schema snippet follows:

```
<define name="intOrStr" datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <choice>
    <data type="integer"/>
    <data type="string"/>
  </choice>
</define>
```

The `intOrStr` patterns is matched by any integer or string.

### 3.4 Namespaces

RELAX NG is namespace-aware, and it provides two mechanisms to handle qualified names. The first is represented by the `ns` attribute, which may appear within `<element>` and `<attribute>` elements. Its value is a URI and specifies the namespace which the matching element or attribute must be bound to. An empty value indicates a null or absent namespace URI.
The other mechanism is represented by the use of qualified names within the `name` attribute of `<element>` and `<attribute>` patterns. In such a case, an element matches the pattern if its namespace matches the one which the qualified name is associated with.

### 3.5 Modularity

RELAX NG allows schema authors to write modular schemas. There are two ways to refer to an external schema. The simplest one is represented by the `<externalRef>` pattern, which must have a `href` attribute pointing to an external RELAX NG schema. It is matched by what matches the pattern contained in the external file.

Another way to use external patterns is represented by the grammar merging mechanism, which is used to merge two or more `<grammar>` patterns. Indeed, within a `<grammar>` element, `<include>` elements may appear. They are used to include definitions stored in external grammars, thus allowing the including grammar to refer to them.

When two or more definitions have the same name, they have to be combined together. The combination of two definitions is similar to the derivation by extension of XML Schema. Indeed, in XML Schema the actual content defined by an extension-derived type is obtained applying a sequence operator to the content defined within the base type, and that within the new type. In RELAX NG two homonymous definitions can be combined (properly setting their `combine` attribute) either by `interleaving` or by `grouping`. In the first case, it is as a single definition was specified, and whose pattern was the `<interleave>` operator applied to those of the combined definitions. In the second case, it is as single definition was specified, and whose pattern was the sequence operator applied to those of the combined definitions.

Additionally, RELAX NG allows to redefine parts of an included grammar inserting `<define>` and `<start>` elements within the `<include>` element. The replacing grammar always uses the new version of a replaced definition.

### 3.6 Linking and Default Values

A controversial question of RELAX NG is the absence of any support for identity-constraints definition. Surely its introduction would have corrupted the straightforwardness and simplicity of the language, but on the other hand its lack requires the adoption of a separated language to impose this kind of constraints. However, if XML Schema datatypes are available, it is at least possible to use the ID/IDREF mechanism.

Another controversial design decision of RELAX NG concerns default values. Indeed, it is not possible to specify them either for attributes or elements. This choice comes under the more general policy of not modifying or augmenting the infoset of the instance document. There is also a more practical problem which does not allow a simple default values specification. Consider the following pattern:

```xml
<element name="x">
  <choice>
    <optional>
      <attribute name="y">
        <text/>
      </attribute>
    </optional>
    <element name="y">
      <empty/>
    </element>
  </choice>
</element>
```

Suppose that a default value is specified for the `y` attribute, and consider the instance document `<x/>`. In such a case, is the `y` attribute absent because the default value is intended to be used? Or is it absent because the `<y>` child element should be present, but it is erroneously missing? Thus, using typical mechanisms for default values specification seems to be unsuitable.
in some patterns, because it could lead to situations where it is not possible to decide whether a
pattern is satisfied or not.

3.7 Co-Constraints Support

The equivalent treatment of elements, attributes and text nodes allows RELAX NG to specify a
number of co-constraints on XML documents. For instance:

- **Mutual exclusion**: a constraint like “an `<x>` element must have either an `<a>` attribute or a `<b>` attribute, but not both” can be expressed in RELAX NG by the following pattern:

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

- **Inter-dependencies between elements and attributes**: A constraint like “an `<x>` element must have a `<y>` child if an `<a>` attribute is specified and its value is not "v1", otherwise it must be empty” can be expressed in RELAX NG by the following pattern:

```xml
<element name="x">
  <choice>
    <group>
      <attribute name="a">
        <data type="string">
          <except>
            <value>v1</value>
          </except>
        </data>
      </attribute>
      <element name="y"><text/></element>
    </group>
    <optional>
      <attribute name="a">
        <value>v1</value>
      </attribute>
    </optional>
  </choice>
</element>
```

However, co-constraints must be defined as patterns, i.e., through regular expressions, and
often, as it can be deduced observing the example above, a co-constraint is better formalized
through something like a logic formula, rather than an equivalent regular expression. Further-
more, there are co-constraints and context-dependent definitions that may produce extremely
long patterns. For instance, SGML-like exclusions can be formalized using patterns, but it could
require, as previously observed for XML Schema, to duplicate a large part of the specification.

Finally, there are several kinds of co-constraints which cannot be formalized at all in RELAX
NG. For instance, it is not possible to require sibling elements to have different values for a given
attribute, as well as it is not possible to require two attributes to be different if both are specified.

4 DSD

DSD (Document Structure Description) [KMS00] 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
DSD has an XML-based syntax, and takes a grammar-approach to validation. However, it provides some constructs to impose conditional constraints and context-dependent definitions, besides those for specifying default values, and identity-constraints (points-to requirements in DSD parlance). A major flaw of DSD is its inability to support namespaces. However, such a limitation is not present in DSD2.

A DSD document mainly consists of element definitions, constraint definitions, and default specifications. An element definition is used to associate one or more constraint definitions to an element. Each constraint either specifies which content is allowed for that element, which attributes, or even a boolean formula which the element must comply with. Constraints are defined either locally within element definitions (and in this case they are called descriptions), or globally, at top-level. Globally defined constraints have a name which allows them to be referred to from element definitions.

Defaults are specified separately from element and constraint definitions.

4.1 Element Definitions
An element definition is used to associate one or more constraint definitions to an element. Each element definition is uniquely identified by an ID, which has to be made explicit by the ID attribute. Element definitions also provide a name indicating the name of the element which the element definition is assigned to. Indeed, during the DSD validation process, each element of the instance document is assigned an ID identifying an element definition. An element is valid if it complies with all the constraints within the element definition, and its name matches that specified by the element definition. Each DSD schema must specify which element definition the root element has to be assigned to.

4.2 Attribute Declarations and Content Expressions
An important constraint is represented by the attribute declarations, consisting of a name and a string regular expression (see the subsection below), describing the set of allowed values for that attribute.

The content of an element is constrained by a content expression, which is a regular expression over element definitions making use of standard operators like those within RELAX NG patterns.

A simple element definition making use of an attribute declaration and a content expression follows:

```xml
<ElementDef ID="X" Name="x">
  <AttributeDecl Name="a" Optional="Yes">
    <Union>
      <String Value="v1"/>
      <String Value="v2"/>
    </Union>
  </AttributeDecl>
  <Optional>
    <Element IDRef="X"/>
  </Optional>
</ElementDef>
```

Such an element definition is satisfied by all `<x>` elements optionally having an `a` attribute whose value is either "v1" or "v2", and whose content is either empty or consisting just of an element satisfying this element definition itself.

The IDs of element definitions are reminiscent of nonterminals in context-free grammars, and they allow several versions of an element to coexist.

---

3. 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.
4.3 Datatypes
In DSD, datatypes are called string types and are string regular expressions over Unicode characters, making use of common and standard operators, like sequence, union, repetition, intersection and complement. A string type can be assigned to attributes as well as to elements. There is no built-in string type.

Since string types can be globally defined, a string type can be constructed referencing existing ones. Under this view, rather than speaking of string regular expressions, we should speak of string grammars.

However, string types have to be constructed using a heavy XML-based syntax, which, in complex cases, requires very verbose expressions to be written.

4.4 Linking
DSD provides a finer linking mechanism than that in DTDs. Indeed, each attribute may be declared to be either of type ID (thus uniquely identifying the element it appears within) or of type IDRef (and thus pointing to an element having an attribute of type ID).

Additionally, an attribute of type IDRef may be constrained by a so called points-to requirement, specifying a pattern on the context of the element the attribute appears within. Such a pattern identifies which element the attribute of type IDRef points to.

4.5 Default Specifications
DSD provides a very flexible and fine default specification mechanism. Indeed, default values may be provided both for elements and attributes, and are specified at top-level, i.e., they are specified independently of element definitions and attribute declarations. A default specification is associated to a boolean expression (see 4.7), and is applied to those elements and attributes of the instance document satisfying the expression.

4.6 Modularity
DSD allows to include an external DSD document through the <?include?> processing instruction. An including DSD may reference all the included definitions. Moreover, a definition can be redefined by another one by means of the RenewID attribute. A peculiar feature of DSD is that the IDRef attribute makes reference to the final version of a definition (or redefinition), while the CurrIDRef attribute makes reference to the second last definition (or redefinition).

4.7 Co-Constraints Support
DSD provides support for co-constraints definition through two kinds of constructs: boolean expressions and conditional constraints. A boolean expression is a boolean formula constructed over attribute descriptions and context patterns using common boolean connectors. An attribute description is used to check the presence and value of an attribute, while a context pattern is used to impose constraints on the context of an element, i.e., the sequence of its ancestors. Thus, in order to impose the mutual exclusion among two attributes, a boolean expression may be used, as in the following element definition:

```xml
<ElementDef ID="X" Name="x">
  <AttributeDecl Name="a" Optional="Yes">
    <StringType IDRef="S"/>
  </AttributeDecl>
  <AttributeDecl Name="b" Optional="Yes">
    <StringType IDRef="S"/>
  </AttributeDecl>
  <And>
    <Not>
      <And>
        <Attribute Name="a"/>
      </And>
    </Not>
  </And>
</ElementDef>
```
An `<x>` element satisfies the above element definition if it has either an `a` attribute or a `b` one, but not both.

A conditional constraint is an “if-then-else” construct whose guard is a boolean expression. An example is shown in the following element definition:

```
<ElementDef ID="X" Name="x">
  <If>
    <Context>
      <Element Name="p">
        <Attribute Name="a" Value="int"/>
      </Element>
      <Element Name="x"/>
    </Context>
    <Then>
      <StringType IDRef="Integer"/>
    </Then>
    <Else>
      <StringType IDRef="Float"/>
    </Else>
  </If>
</ElementDef>
```

requiring that, if the `<p>` parent has an `a` attribute whose value is "int" then the content of `<x>` must be a string matching the `Integer` string type, otherwise it must be a string matching the `Float` string type.

However, boolean expressions (and consequently conditional constraints) have two problems. The first one concerns the syntax. Indeed, boolean expressions use an XML-based syntax, which makes heavy writing complex boolean formulae. The second problem concerns their expressiveness. Indeed they allow to express constraints just on ancestor elements and their attributes. Thus, it is not possible to express conditions on sibling or descendant elements. Furthermore, it is not possible to compare values, and thus, for instance, an attribute cannot be required to be greater than another attribute.

## 5 Schematron

Schematron [Jel02] 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. Schematron provides an open specification, i.e., all that is not forbidden is allowed.

A Schematron document defines a sequence of `<rule>`s, logically grouped in `<pattern>` elements. Each rule has a `context` attribute, which is an XSLT pattern determining which elements in the instance document the rule applies to. Within a rule, a sequence of `<report>` and `<assert>` elements is specified, both 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. The assertion
within a <report> is output when its test succeeds, while that within an <assert> is output when its test fails. Thus, the <report> element is used to tag negative assertions about the instance document, while the <assert> element is used to tag positive ones.

5.1 Specifying Element Content and Attributes

As aforementioned, Schematron is a rule-based language. Thus there is no explicit construct to declare element contents or attributes: all has to be formalized through XPath expressions. Obviously, XPath has not been designed to impose grammatical-like constraints, but rather to express paths on XML documents. As a consequence it is really not straightforward to formalize such a kind of constraints. For instance, given the following DTD element type declaration

    <!ELEMENT x (y?, (w | z)+) >

requiring all <x> elements to have an optional <y> child element followed by a sequence of one or more <w> or <z> elements, an equivalent Schematron rule is:

    <sch:rule context="x">
        <sch:report test="normalize-space(text())!=''" >Text is not allowed.</sch:report>
        <sch:assert test="count(*)=count(y|w|z)" >A not allowed child.</sch:assert>
        <sch:report test="y and not(*[1][self::y])" >y element only in first position.</sch:report>
        <sch:report test="count(y) > 1" >More than one y child.</sch:report>
        <sch:assert test="w or z" >At least a w or z must be present.</sch:assert>
    </sch:rule>

The first <report> is used to impose that no character other than whitespace is allowed, while the following <assert> imposes that only <y>, <w>, and <z> child elements are allowed. Then, the rule imposes that if a <y> element is present, then it must be in first position, and that no more than one <y> element may be present. Finally, the last <assert> assures that at least one among <w> and <z> is present.

On the other hand, requiring the presence of attributes is easier, as shown by the following rule:

    <sch:rule context="x">
        <sch:assert test="count(@*)=count(@a)" >A not allowed attribute.</sch:assert>
        <sch:assert test="@a" >@a is required.</sch:assert>
    </sch:rule>

requiring all <x> elements to have an a attribute.

Moreover, the rule-based approach of Schematron does not allow it to support default values. Indeed, XPath expressions used in the various Schematron constructs are always evaluated on the instance document without any preceding validation process occurs.

5.2 Datatypes

In Schematron, the only built-in datatypes are those provided by XPath: booleans, strings, and numbers. Schema authors have to impose constraints on datavalues using XPath expressions, which are not well-suited for these purposes. XML Schema simple types and DSD string types seem much more straightforward and natural to use than XPath expressions.

5.3 Namespaces

Schematron is namespace-aware, and thus qualified names can be used within XPath expressions. Prefixes of qualified names have to be declared using <ns> elements within the <schema> root element. An example follows:
5.4 Linking

Schematron allows schema authors to require an element or attribute to have a unique value among other elements and attributes, and it also allows them to require an attribute to point to another element or attribute. An example follows:

```xml
<sch:rule context="x">
    <sch:assert test="count(//x[@id=current()]/@id)=1">
        @id not unique.
    </sch:assert>
    <sch:assert test="//x[@id=current()]/@idref"]
        @idref points to nothing.
    </sch:assert>
</sch:rule>
```

The first assert of such a rule requires all `<x>` elements to have an `id` attribute uniquely identifying them among all other `<x>` elements. The second one requires all `<x>`s to have an `idref` attribute pointing to any `<x>` element.

Furthermore, Schematron allows `<key>`s within rules so that the XSLT key mechanism can be used. An example follows:

```xml
<sch:rule context="x">
    <sch:key name="key" path="@id"/>
    <sch:assert test="key('key', @idref]
        @idref points to nothing.
    </sch:assert>
</sch:rule>
```

5.5 Modularity

Schematron provides a simple macro mechanism on rules. A `<rule>` element can have one or more `<extends>` elements, which have a `rule` attribute referencing an abstract rule. In this way, the assertions of the abstract rule are brought into the current one. A rule is declared to be abstract setting the `abstract` attribute to "true". Moreover, Schematron provides the so called phase mechanism. A Schematron schema may specify a sequence of `<phase>` elements, each containing `<active>` elements pointing to `<pattern>`s. Thus, a phase identifies a sequence of active patterns. A Schematron implementation may give the user the opportunity of choosing which phases have to be used for a particular validation.

However, Schematron does not allow to reference external patterns.

5.6 A Common Implementation of Schematron

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, obtaining the output of the validation process (i.e, a sequence of assertions).
5.7 Co-Constraints Support

The strength of Schematron is in the facility of the co-constraints definition. Indeed, while its approach to validation is not well-suited to express classical constraints usually formalized through grammars, it makes the definition of co-constraints straightforward.

To illustrate the expressiveness of Schematron, we show a rule imposing some co-constraints that cannot be checked by the languages previously described.

```xml
<sch:rule context="x">
  <sch:assert test="@ny=count(y)" >Number of ys must be @ny.</sch:assert>
  <sch:assert test="count(y)>count(z)" >number of ys less or equal to that of zs.</sch:assert>
  <sch:report test="@a and @b and @a!=@b" >@a and @b must have the same value.</sch:report>
</sch:rule>
```

Such a rule requires all `<x>` elements:

- to have the same number of `<y>`s as that indicated by the `ny` attribute,
- to have a number of `<y>`s strictly greater than that of `<z>`s, and
- to provide the same value for the `a` and `b` attributes if both are specified.

6 xlinkit

xlinkit [NCEF02] 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. All of its authors have a software engineering background, and thus it is not surprising that the main goal of xlinkit is to provide support in the area of consistency checking of distributed specifications. However, it has been recognized also as a powerful tool in checking complex requirements in XML documents which cannot be imposed by grammar-based languages.

An xlinkit application defines, using an XML-based syntax, a set of documents (called document set) and a set of consistency rules (called rule set), and proceeds to verify the rules on every file in the document set. The rule is expressed in the constraint language CLIX (Constraint Language in XML), which is a first-order language using predicates, quantifiers and variables. In order to select the nodes associated to a quantifier, XPath expressions are used.

A document set is an XML document making reference to one or more XML input documents, while a rule set is an XML document referencing one or more rule files, where each rule file is an XML document specifying a set of consistency rules. When consistency rules are checked, the produced output is not a boolean value, but rather a set of XLink hyperlinks [DMO01], binding together elements satisfying the rules and/or those violating them. Such a particular output is specifically thought to meet needs of consistency checking.

6.1 Specifying Element Content and Attributes

As observed for Schematron, xlinkit is not well-suited to express classical constraints on elements content, that are better captured by grammar-based languages. For instance, a consistency rule equivalent to the DTD element type declaration

```xml
<!ELEMENT x (y | z)+ >
```

is the following one:

```xml
<consistencyrule id="r1">
  <forall var="x" in="//x">
    <and>
      <not>
        <exists var="u" in="$/x/*[name()!='y' and name()!='z']"/>
      </not>
    </and>
  </forall>
</consistencyrule>
```
which requires all \(<x>\) elements of the document set to not contain an element other than \(<y>\) or \(<z>\), and to have at least one among \(<y>\) and \(<z>\).

In order to check the presence of an attribute, the \(<\text{exists}>\) quantifier can be used.

### 6.2 Datatypes and Default Values

Like Schematron, xlinkit does not provide any datatype, except those used within XPath expressions. Thus, the \(<\text{equal}>\) operator, used to compare two values, works on numbers, strings and booleans. Obviously, simple requirements on datavalues can be met, but imposing complex constraints like those formalized by means of string regular expressions is a problematic issue.

Like Schematron, xlinkit is not able to specify default values for attributes or elements.

### 6.3 Namespaces

xlinkit is namespaces-aware, thus qualified names can be used in XPath expressions. In order to associate the prefix of a qualified name with a namespace URI, a namespace declaration has to be provided within the root element of a rule file.

### 6.4 Linking

Identity constraints may be formalized using consistency rules. For instance, the consistency rule

\[
<\text{consistencyrule id="r1">}
<\forall var="x" in="//x">
<\forall var="y" in="//x">
<\text{implies}>
<\text{equal op1="}$x/@id$" op2="}$y/@id$/"}>
<\text{same op1="}$x$" op2="}$y$/"}>
</\text{implies}>
</\forall>
</\forall>
</\text{consistencyrule}>
\]

requires that if there are two \(<x>\) element nodes with the same value for the \(id\) attribute, then they must be the same element node.

### 6.5 Modularity

xlinkit has been designed to be highly modular. Indeed, as aforementioned, the set of consistency rules is constructed selecting single rule files, and similarly the document set is constructed selecting single XML documents. Moreover, a rule set document can select also specific rules of a rule file. In fact, each reference to a rule file can be associated with an XPath pattern evaluated on the rule file document. In order to simplify this task, each consistency rule has an \(id\) attribute uniquely identifying it within a rule file.

Furthermore, xlinkit provides a macro mechanism that allows to reuse a formula. Indeed, a rule file may include a \textit{macro definition file}, an XML document defining one or more \textit{macros}. Each macro has zero or more parameters, which allow it to be used in different consistency rules.

xlinkit also allows authors to write their own \textit{operators}, which are parameterized plug-in predicates to be used in consistency rules. An \textit{operator set} is an XML document serving as an interface
Combining Different Approaches

It specifies their name, parameters, and where their implementation file is located. At the moment, the only language supported by xlinkit to implement operators is ECMAScript.

Finally, a document set may reference a document whose format is not XML. In such a case, it has to specify a specific fetcher for that document. Such a fetcher has to be implemented and registered with xlinkit before it can be referred to in a document set.

6.6 Co-Constraints Support

It is possible to express rather complex co-constraints in xlinkit. Indeed, the conjunction of XPath expressions and boolean connectors leads xlinkit to have a great expressiveness. For instance, the mutual exclusion between two attributes can be expressed by means of a conceptually simple consistency rule, as well as the dependency of an element on an attribute value. The rule

```xml
<consistencyrule id="r1">
  <forall var="x" in="/r/x">
    <forall var="y" in="/r/y">
      <iff>
        <equal op1="x/@a" op2="'v1'"/>
        <exists var="z" in="y/z"/>
      </iff>
    </forall>
  </forall>
</consistencyrule>
```

enforces all `<y>` elements to have a `<z>` child if and only if its sibling `<x>` has an `a` attribute whose value is "v1".

As another example, consider the rule

```xml
<consistencyrule id="r1">
  <forall var="x" in="/r/x">
    <forall var="y" in="/r/y">
      <not>
        <equal op1="x/@a" op2="y/@a"/>
      </not>
    </forall>
  </forall>
</consistencyrule>
```

It requires `<x>` and `<y>` elements (which are sibling) to have differing values for their `a` attributes.

7 Combining Different Approaches

From the description of the six schema languages above, it can be deduced that none of them provides all the necessary features for a rich and complex XML document type. In general, what can be straightforwardly formalized in a language could require a much more involved expression in another language, and viceversa. In particular, as previously highlighted, a given co-constraint is naturally and easily expressed using a xlinkit consistency rule or a Schematron rule, but requires a long and convoluted RELAX NG pattern, and it cannot be formalized at all in XML Schema or DTD. On the other hand, a simple and clear DTD content model could require a long and obscure consistency rule. Moreover, almost all grammar-based languages provide some kind of support to specify default values, which undisputedly is a common and appreciated feature. On the other hand, neither xlinkit nor Schematron provide such kind of support. Finally, many grammar-based languages allow to define rather complex constraints on datavalues, while both Schematron and xlinkit clearly appear limited on this issue.

Thus a schema language alone could not be sufficient to check whether an instance document meets all the syntactic requirements that an application needs to impose on it. When DTDs
and XML Schema were the only schema languages available, the only possible solution was to write schemas as strict as possible, and then checking the unspecified requirements through specific validation code written in one of the several programming languages. As observed in [com, NE03], such an approach has the advantage that the full power of a programming language can be exploited, but implementers are forced to provide their own validation code, with repetition of efforts and no guarantee of correct and widespread implementations. Furthermore, programming language-dependence is introduced.

Then, with the advent of other schema languages (and in particular, with the advent of rule-based ones), other approaches were proposed. Indeed, in order to obtain a rich and complete validation, an instance document can pass through a grammar-based validation, and then through a rule-based one. The former assures that the instance document complies with structural requirements (by means of grammatical expressions), and that datavalues belong to specific set of values (by means of datatypes). The latter checks whether the instance document complies with additional constraints. Under this view, both [Rob] and [com] propose to embed a Schematron specification within an XML Schema schema, thus creating a single schema able to specify both “classical” constraints (using XML Schema constructs) and co-constraints (using Schematron rules). The idea is to put Schematron <pattern> elements into XML Schema <appinfo> elements (which may optionally appear within almost all elements defined by XML Schema, and which are intended to provide information to applications). In particular, a Schematron <pattern> can be embedded within the <appinfo> element of an element declaration, specifying that the declared element has to satisfy the type defined by the declaration, and the embedded Schematron rule. From such an extended XML Schema schema, an XSLT stylesheet may extract a Schematron schema comprising of the embedded rules, and then a plain and standard XML Schema validation may be performed on the instance document, which may be also validated against the extracted Schematron schema.

Although such a framework could appear as a simple and practical solution, it presents some disadvantages. As also highlighted in [com], a schema author is forced to learn two different schema languages. Furthermore, such a solution surely is not much elegant, and seems to be a bit contrived. In the general case, it is not in the least so easy to associate a Schematron rule with an XML Schema element declaration. For instance, consider the following schema snippet:

```xml
<xsd:element name="x">
  <xsd:annotation>
    <xsd:appinfo>
      <sch:pattern name="Check y greater than z">
        <sch:rule context="x">
          <sch:assert test="y > z">
            y should be greater than z.</sch:assert>
        </sch:rule>
      </sch:pattern></xsd:appinfo>
    </xsd:annotation>
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="y" type="xsd:integer"/>
        <xsd:element name="z" type="xsd:integer"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
```

Such an element declaration is intended to constrain the content of <x> elements to have two child elements, <y> and <z>. Both are declared to be of type integer. Additionally, the embedded Schematron rule is used to check whether <y> is greater than <z>. To be more precise, the embedded rule asserts that all <x> elements of the instance document has a <y> child greater than a <z> one. Suppose that the above declaration is provided within a given complex type,
and that the schema also declares (of course within another complex type) an \(<x>\) element to be of type string. Since the embedded rule shown above applies to all \(<x>\) elements, it also applies to all \(<x>\)s declared to be of type string. Obviously, although being valid, such elements do not satisfy the rule.

Writing a rule which just applies to elements declared with the element declaration which it is embedded within, is a hard (if not impossible) task, because such elements have to be recognized through XPath expressions.

Furthermore, consider the following statement: "\(<x>\) must contain a \(<y>\) child element if its parent has not an \(a\) attribute whose value is \("v1\)" otherwise its content must be simple and constrained by the \(S\) simple type". Obviously, such a constraint cannot be formalized by a simple XML Schema schema. Thus, a Schematron rule could be embedded within the \(<x>\) element declaration, as shown in the following:

```xml
<xsd:element name="x">
  <xsd:annotation>
    <xsd:appinfo>
      <sch:pattern name="Not precise">
        <sch:rule context="x">
          <sch:report test="../@a!='v1' and 
          (not(y) or normalize-space()!='')">
            >If ../@a!='v1' y must be present and only whitespace is allowed.</sch:report>
          <sch:report test="../@a='v1' and y">
            >A y element cannot be present if ../@a='v1'.</sch:report>
        </sch:rule>
      </sch:pattern>
    </xsd:appinfo>
  </xsd:annotation>
  <xsd:complexType mixed="true">
    <xsd:sequence>
      <xsd:element name="y" type="yT" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Such an element declaration precisely formalizes just the first part of the statement: whenever the \(<x>\)’s parent has not an \(a\) attribute whose value is \("v1\), \(<x>\) must contain a \(<y>\) child, and characters other than whitespaces are not allowed. The second part is not well expressed. In fact, it is just assured that if the \(<x>\)’s parent has an \(a\) attribute whose value is \("v1\) then \(<y>\) cannot be present and text may appear, but it is not in the least assured that such text must satisfy the \(S\) simple type. In order to impose also this constraint, further Schematron assertions are required, but, as previously discussed, Schematron is not well-suited to constrain datavalues.

Moreover, checking co-constraints through embedded Schematron rules also makes difficult the analysis of the overall validation process output. Indeed, the XML Schema validation process output is the PSVI, while the Schematron one is a list of assertions (eventually in XML format). As aforementioned, when XML Schema (as all the other schema languages) is not able to impose a constraint, for instance on an element, the type of that element has to be laxly defined, i.e., it has to accept all correct values, even if wrong ones could be accepted too. Thus, the validity of the element can be established only analysing the output of both XML Schema and Schematron validation processes. As a consequence, the PSVI may lose part of its importance, especially when the lax type is really lax (for instance, that in the example above).
8 Formal Analyses of Schema Languages

As the importance of XML validation becomes evident, formalizations of schema languages, and mathematical analysis in the XML research assume a relevant role. Several motivations have driven many researchers to use formal methods in this environment in the last years. Many efforts have come from the database community. For instance, [PV00] addresses the problem of, given a source DTD and a view definition, automatically inferring a tight DTD for the view. In particular a formal framework is provided, where XML documents are modeled as ordered trees with labeled nodes, while DTDs are modeled as ltds (labeled tree definitions). In order to be able to express certain inferred types, [PV00] extends DTD types from regular expressions to context-free grammars. A query language for ltds is formally defined.

[AMN01] investigates the static typing problem for XML queries, i.e., verifying at compile time that every XML document which is the result of a specified query applied to a valid input document, satisfies the output DTD. In the formal framework defined there, XML documents are abstracted as data trees, i.e., finite ordered labeled trees with data values attached to nodes, while DTDs are abstracted as extended context-free grammars, i.e., a context-free grammars where right-hand sides of productions are regular expressions over terminals and non-terminals. A query language allowing comparisons on data values is formally defined.

Work on formal analysis of XML documents and XML type systems is also made in [HP03], which proposes XDue, a functional programming language that takes XML documents as primitive values. Types are essentially regular expressions over elements and type names. Functions can be defined over XML data, and XDue performs static typing for these functions, verifying that their output will always be of the claimed type. [HP03] formally defines the core XDue language using inference rules.

[MLM01] proposes to use tree grammar theory as a general formal framework to analyse in abstract mathematical terms the several schema language proposals. Four subclasses of tree grammars are formally defined, each characterized by its own expressive power. Based on this framework, a number of existing grammar-based languages is then analysed and compared, stating whether a schema language is more powerful than another schema language. However, the described framework is not able to capture all the features of schema languages, e.g., identity constraints in XML Schema, or conditional constraints and boolean expressions in DSD.

There are also validation languages with an official formal specification. In this way, the language semantics is rigorously and compactly defined, so that users can deeply understand it, and developers’ task is made easier. Also the designers of the language itself are in somehow helped. For instance, they can better evaluate what repercussions the introduction of new features may cause. Moreover, when a language is accompanied with a formal specification, designing other specifications and tools that build on that language becomes easier. Among the schema languages with a formal semantics are XLink and RELAX NG. [NCEF02] formally defines the language used by consistency rules, thus rigorously showing how their evaluation generates XLink hyperlinks, while RELAX NG specification uses formal methods to define the semantics of its patterns.

Although [TBMM01] does not provide a formal semantics for XML Schema, two works subsequently published cover such a lack: [BFRW01] and [JS03]. [BFRW01] proposes MSL (Model Schema Language), an attempt to formalize some of the core idea in XML Schema, taking a purely structural approach. Initially, it proved helpful in the design of XML Query. However, both XQuery and XPath working groups then abandoned MSL in favour of the idea described in [JS03], which provides, on the other hand, a purely named approach to typing. Indeed, XQuery has both a specification in prose and a formal semantics, which is precisely based on the idea showed in [JS03].
Chapter 3

SchemaPath

1 Introduction

In this chapter we informally illustrate the SchemaPath syntax and semantics. We also provide numerous real-world examples of constraints defined by SchemaPath specifications, showing the expressiveness, flexibility, and usefulness of the language. Finally, in Sect. 14 we give an hint to a SchemaPath formalization.

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.

SchemaPath extends XML Schema introducing the concepts of conditional declaration, conditional element and conditional attribute. A conditional declaration lists a sequence of alternative type definitions, each associated with an XPath predicate and a priority. A conditional element (attribute) is an element (attribute) node of the instance document whose declaration is conditional. To validate a conditional element (attribute), the XPath predicates are evaluated. The conditional element (attribute) is valid if its type is the one associated to the successful XPath predicate with the highest priority.

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 must not be satisfied.

2 Namespace

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.

Unless otherwise stated, in the rest of this chapter, we will use the xsd prefix, assuming it is bound either to the SchemaPath namespace or to the XML Schema one.

3 Conditional Declarations

In SchemaPath, just as it is in XML Schema, a declaration is an association of a name with a type definition. Given an element (attribute) node of the instance document and an element (attribute) declaration, the element (attribute) validates against the declaration if and only if its name matches the one specified in the declaration and its content satisfies the type.

A conditional declaration lists a sequence of alternative type definitions, each associated with an XPath predicate and a priority. To validate a conditional element (attribute), the XPath predicates are evaluated. The conditional element (attribute) is valid if its type is the one associated to the successful XPath predicate with the highest priority.
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 conditions expressed as XPath templates.

```xml
<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>
```

Briefly, we could express this definition as follows: the <invoiceLine> element must have a <unit> element whose type is unitType and that is followed by a <quantity> element whose type is:

- xsd:integer, when the string value of <unit> is "items",
- xsd:decimal, when the string value of <unit> is "meters".

If neither the first nor the second condition is satisfied, a validation error occurs.

SchemaPath makes no assumption on the validity of the XPath expression: any correct statement can be expressed, even an impossible one according to the data type of the element; if this is the case, the expression will simply be never satisfied by the document, and the alternative never chosen. For instance, SchemaPath does not 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 expression may be satisfied, otherwise it will be always ignored.

### 3.1 Syntax

Here, we present the syntax of conditional declarations making use of the representation used in [TBMM01].

The syntactic structure of a conditional element declaration is:

```xml
<element
  block = (#all | List of (extension | restriction))
  form = (qualified | unqualified)
  id = ID
  maxOccurs = (nonNegativeInteger | unbounded) : 1
  minOccurs = nonNegativeInteger : 1
  name = NCName
  {any attributes with non-schema namespace...}>
  Content: (annotation?, (alt+, (unique | key | keyref)*))
</element>
```

With respect to a plain XML Schema element declaration, a conditional element declaration skips some attributes (abstract, final, fixed, default, nillable, substitutionGroup, ref and type), and allows no anonymous type definition.

The syntactic structure of a conditional attribute declaration is:
3 Conditional Declarations

<attribute
    form = (qualified | unqualified)
    id = ID
    name = NCName
    use = (optional | prohibited | required) : optional
    {any attributes with non-schema namespace . . .}>
    Content: (annotation?, (alt+))
</attribute>

Again, with respect to a plain XML Schema attribute declaration, a conditional attribute declaration lacks some attributes (default, fixed, ref and type), and allows no anonymous type definition.

Within a conditional element or attribute declaration a list of one or more <alt> elements is expected. The <alt> element syntax is as follows:

<alt
    cond = an XPath expression : true()
    default = string
    fixed = string
    nillable = boolean : false
    priority = Number
    type = QName
    {any attributes with non-schema namespace . . .}>
    Content: (annotation?, (simpleType | complexType)?)
</alt>

All of the attributes but the cond and priority are present in plain declarations and are equivalent semantically to those of non-conditional element and attribute declarations.

In the representation of conditional element and attribute declarations we have intentionally omitted the ref attribute, because conditional declarations allow no references.

3.2 The cond Attribute

Conditions are specified in form of XPath expressions by the cond attribute. More precisely, the XPath expressions are those used in the predicates of XSLT patterns (see [Cla99]). This means that there are two important restrictions: neither the XSLT current() function, nor variable references can be used in a condition.

There is another restriction that concerns fully qualified names and that is also borrowed from XSLT. A well-known limitation of XSLT (which on the other hand is being stigmatized and scheduled for removal in the next version, as specified in [MS01]) 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.

The cond attribute may not explicitly appear within an <xsd:alt> element. In this case, it defaults to true(), i.e., 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).

For instance, the element declaration

<xsd:element name="x">
    <xsd:alt cond="../@a='v1'" type="xsd:decimal"/>
    <xsd:alt type="xsd:integer"/>
</xsd:element>

enforces the <x> element to be of type xsd:decimal whenever the value of the a attribute of
the containing element is "v1" and to be of type xsd:integer in all other cases.

3.3 Priorities and the priority Attribute

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

```xml
<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 `<unit>` element is present; of course, a situation satisfying the first condition also satisfies the second one.

In these situations, each alternative of a conditional declaration should be explicitly assigned a priority, which is a positive or negative real number. It is set through the optional priority attribute of the `<xsd:alt>` element.

When the priority attribute is not present, SchemaPath computes a default priority as function of the XPath condition. The rules used to compute the default priority are similar to those used to compute the default priority of XSLT templates [Cla99], and they are as follows:

- If the condition has the form of a QName preceded by a ChildOrAttributeAxisSpecifier or has the form processing-instruction(Literal) preceded by a ChildOrAttributeAxisSpecifier, then the priority is 0.
- If the condition has the form NCName:* preceded by a ChildOrAttributeAxisSpecifier, then the priority is -0.25.
- Otherwise, if the condition consists of just a NodeTest preceded by a ChildOrAttributeAxisSpecifier, then the priority is -0.5.
- Otherwise, if the condition is true(), then the priority is the greatest integer strictly lower than the lowest priority of the other alternative within the same conditional declaration.
- Otherwise, the priority is 0.5.

When a conditional element or attribute matches more than one alternative with the same priority, SchemaPath adopts the same behaviour as that of XSLT for conflicting template rules. Indeed, a SchemaPath processor may signal the error; otherwise, it must recover by choosing, from amongst the matching alternatives, the one that occurs last in lexical order.

According to the aforementioned rules, the default priority of both alternatives of the above element declaration is 0.5. Thus, in order to avoid possible conflicts, we can rewrite that conditional declaration as follows,

```xml
<xsd:element name="quantity">
  <xsd:alt cond="./unit='items'" type="xsd:integer" priority="1"/>
  <xsd:alt cond="./unit" type="xsd:decimal"/>
</xsd:element>
```

assuring that the first condition is checked first.

On the other hand, the following element declaration

```xml
<xsd:element name="quantity">
  <xsd:alt cond="@unit type="myInteger"/>
  <xsd:alt cond="@unit='meters'" type="myDecimal"/>
  <xsd:alt type="xs:string"/>
</xsd:element>
```
cannot generate conflicts, because the three alternatives have priority respectively 0, 0.5, and -1.

Note that the definition of the default priority for an alternative whose XPath predicate is true() implies that any other alternative of the same conditional declaration cannot have true() as XPath predicate and no explicit priority set (the priority attribute is absent). Indeed, in such a case it would not be possible to compute the default priority for those alternative whose predicate is true().

4 The xsd:error Simple Type

SchemaPath introduces a new built-in simple type: xsd:error. It is an unsatisfiable type, i.e., its value space is empty. Thus, assigning xsd:error to an element will inevitably 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.

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.

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

Since the cond attribute defaults to true(), the color attribute will be of type xsd:error if the <description> element has a print attribute, and of type PantoneCodeType in all other cases.

As in all of the other examples in this chapter, the xsd prefix is assumed to be bound either to the XML Schema namespace or to the SchemaPath one. This last example is not an exception. It means that the error type can be referenced using a qualified name whose prefix stands either for the XML Schema namespace or for the SchemaPath one.

5 Other Issues in Conditional Declarations

In this section we highlight some points of major interest concerning the use of conditional declarations.

5.1 Global and Local Declarations and References

From a grammatical perspective, the distinction between local and global declarations represents an important improvement of XML Schema over DTDs.

SchemaPath keeps this distinction valid for plain declarations of elements and attributes and extends it to conditional ones. Thus, it is possible to declare more than one conditional element (attribute) with the same name and target namespace, provided that they are in different contexts. Obviously, it is not possible to have two element (attribute) declarations with the same name
5 Other Issues in Conditional Declarations

and target namespace and in the same context, even if one is conditional and the other is non-
conditional.

Global conditional element and attribute declarations can be referenced using the `ref` at-
ttribute within the `<xsd:element>` and `<xsd:attribute>` elements.

5.2 Value Constraints

In SchemaPath, value constraints are those defined in XML Schema (i.e., default and fixed
attributes) and can be also applied to conditional elements and attributes.

Both default and fixed values are strongly related to the type assigned to the element or at-
ttribute which they are applied to. For this reason, in a conditional declaration different value
constraints can be supplied for each alternative, using the `default` and `fixed` attributes within
the `<xsd:alt>` element.

As in XML Schema, there is a difference between the semantics of value constraints for el-
ements and those for attributes: default and fixed values apply to empty elements, while they
apply to missing attributes. This difference also holds for conditional elements and conditional
attributes.

Consider the following schema:

```xml
<xsd:schema xmlns:xsd="http://www.cs.unibo.it/SchemaPath/1.0">
  <xsd:element name="invoiceLine">
    <xsd:complexType>
      <xsd:attribute name="unit" type="xsd:string" use="required"/>
      <xsd:attribute ref="quantity"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:attribute name="quantity">
    <xsd:alt cond="../@unit='items'" type="xsd:string" default="0"/>
    <xsd:alt cond="../@unit='meters'" type="xsd:decimal" default="0.0"/>
  </xsd:attribute>
</xsd:schema>
```

In Table 1, we show different instance documents and, for each of those, what value is sup-
plied for the `quantity` attribute in the PSVI.

<table>
<thead>
<tr>
<th>Instance document</th>
<th>quantity attribute in the PSVI</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;invoiceLine unit='items'/&gt;</code></td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td><code>&lt;invoiceLine unit='meters'/&gt;</code></td>
<td>0.0</td>
</tr>
<tr>
<td><code>&lt;invoiceLine unit='other unit'/&gt;</code></td>
<td>Validation error!</td>
</tr>
<tr>
<td><code>&lt;invoiceLine unit='items' quantity=&quot;12&quot;/&gt;</code></td>
<td>&quot;12&quot;</td>
</tr>
<tr>
<td><code>&lt;invoiceLine unit='meters' quantity=&quot;12.2&quot;/&gt;</code></td>
<td>12.2</td>
</tr>
</tbody>
</table>

Table 1. Defaults for a conditional attribute

XML Schema allows schema authors to specify value constraints also in attribute references.
Such constraints take precedence over those defined in the corresponding global declaration.
SchemaPath too provides this feature, which can also be used when the global declaration is
conditional. Although in the general case a schema author does not know which type will be
assigned to the conditional attribute (and thus he or she does not know if the local value con-
straint and such a type are compatible), there may be situations where the complex type where
the attribute reference takes place univocally determines which condition will be satisfied by
the conditional attribute, and thus which type will be assigned to the attribute itself. Thus, the
schema author can provide an appropriate value constraint.

SchemaPath does not try to recognize such cases, and if the local value constraint and the
actual type that will be assigned to the attribute are not compatible, an error occurs.

5.3 Nullable Elements

SchemaPath allows to declare a conditional element as nullable. Like value constraints, the nil-
lableness has to be specified (using the nillable attribute within the <xsd:alt> element) for
each alternative, making it condition-dependent. This choice has been suggested by the fact that
the nilableness influences the content of an element. In fact, when an element is declared as nil-
labile, its content can be empty, even when its type may require the presence of elements or text.
Obviously, the <xsd:alt> element can contain the nillable attribute only when its parent is
<xsd:element>.

An interesting example is represented by the following declaration:

```xml
<xsd:element name="x">
  <xsd:alt cond="number(../@x-length)=string-length()"
    type="xsd:string" nillable="true"/>
</xsd:element>
```

which requires the <x> element to be a string whose length is specified by the x-length at-
ttribute of its parent.

Let us suppose that the x-length attribute is set to 4, and the conditional element to validate
is <x xsi:nil="true"/>. In this case, the XPath expression does not evaluate to true (the
string-length() function returns 0) and thus a validation error occurs.

Thus, in declaring a conditional element as nullable, the schema author has to make sure that
the condition and the nilableness are compatible.

This example shows again that SchemaPath makes no assumption on the validity of the XPath
expressions. Thus, if they are not compatible with the instance document, they will lead to a
validation error.

5.4 Occurrence Constraints

SchemaPath defines the so-called occurrence constraints in the same way as XML Schema. Thus,
the <xsd:element> element may have the minOccurs and maxOccurs attributes, and the
<xsd:attribute> element may have the use attribute.

Such constraints can also be specified by conditional declarations. In this case it might be
useful to observe that occurrence constraints are not conditional, in the sense that they have to be
respected regardless of the conditions specified in the set of alternatives.

For instance, given the SchemaPath snippet

```xml
<xsd:element name="quantity" maxOccurs="unbounded">
  <xsd:alt cond="@unit='items'" type="myInteger"/>
  <xsd:alt cond="@unit='meters'" type="myDecimal"/>
</xsd:element>
```

```xml
<xsd:complexType name="myInteger">
  <xsd:simpleContent>
    <xsd:extension base="xsd:integer">
      <xsd:attribute name="unit" type="unitType"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
```

```xml
<xsd:complexType name="myDecimal">
  <xsd:simpleContent>
    <xsd:extension base="xsd:decimal">
      <xsd:attribute name="unit" type="unitType"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
```
6 Deriving Types

One of the most peculiar feature of XML Schema is the type derivation, which allows schema authors to define new types, extending or restricting existing ones. In this section we discuss how simple and complex types can be derived in SchemaPath.

6.1 Simple Base Types

Given a simple base type, new simple types can be derived either by list, union or restriction. The syntax and semantics of these sorts of derivation are those described in [TBMM01, BM01]. In particular, all of the several facets provided by XML Schema are available in SchemaPath.

As in XML Schema, complex types can be derived from simple types. In this case, the only allowed derivation method is by extension. This method is used to construct a type whose content is simple but which contains an attribute declarations list. In such a list, conditional attribute declarations may appear and SchemaPath does not impose any restriction on their use.
6.2 Complex Base Types

Base types can also be complex. Only complex types are allowed to be derived from a complex type. There are two kinds of derivation: by restriction and by extension.

Deriving a type by restriction means restricting the content model of the base type, so that the values represented by the derived type are a subset of those represented by the base type. When no conditional element declaration is involved (neither in the restricted type, nor in the base one), there is no problem in the derivation, because it is fully equivalent to the derivation by restriction of XML Schema. On the other hand, when conditional element declarations are involved, some conceptual problems arise. These problems are related to the definition of the subtyping relation. More details can be found in [MSV04b]. For this reason, SchemaPath imposes a severe limitation: a type containing conditional declarations cannot serve as base type for a derivation by restriction. However, SchemaPath does not require such a type to be explicitly declared as final with respect to the derivation by restriction.

On the other hand, deriving types by extension does not arise any theoretical problem, even if conditional declarations are involved. In fact, this kind of derivation works exactly as its counterpart in XML Schema. Thus, deriving a type by extension means “appending” new content to the one declared by the base type. It does not matter whether within the base type or within the added content there are conditional declarations.

7 Using Derived Types in the Instance Document

A peculiar feature of XML Schema is represented by the use of the xsi:type attribute (which is part of the XML Schema instance namespace) within the instance document. The element node which this attribute is applied to is assigned the type specified by the xsi:type attribute itself. Such a type must be derived from the one expected from the schema.

This feature is also present in SchemaPath and the xsi:type can also be applied to conditional elements, but an observation might be useful in this case. The type of a conditional element depends on a condition, which is an XPath expression evaluated in the instance document. Thus, in assigning a type through the xsi:type, one has to pinpoint which holding condition, if any, has the highest priority between those specified in the declaration of the element which the xsi:type is being applied. Once such a condition has been detected, the xsi:type has to point to a type definition which is derived from the one corresponding to this condition.

For instance, consider the following SchemaPath:

```xml
<xsd:schema xmlns:xsd="http://www.cs.unibo.it/SchemaPath/1.0">
  <xsd:element name="x">
    <xsd:alt cond="@a='v1'" type="BT1"/>
    <xsd:alt cond="@a='v2'" type="BT2"/>
  </xsd:element>
  <xsd:complexType name="BT1">
    <xsd:attribute name="a" type="xsd:string"/>
  </xsd:complexType>
  <xsd:complexType name="BT2">
    <xsd:attribute name="a" type="xsd:string"/>
  </xsd:complexType>
  <xsd:complexType name="T1">
    <xsd:complexContent>
      <xsd:extension base="BT1">
        <xsd:sequence>
          <xsd:element name="y"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:schema>
```
8 Restraining the Use of Derived Types

SchemaPath allows to control the use of derived types by the same mechanisms (although with some differences) as those provided by XML Schema.

Thus, a complex type can be declared (setting the abstract attribute of the `<xsd:complexType>` element to "true") as abstract, imposing that such a type is not used as the actual type definition for the validation of element nodes of the instance document. An abstract type can be used in a conditional element declaration. In this case, when its associated condition is satisfied by the element being declared, such an element must have the xsi:type attribute making reference to a type definition which is derived from the abstract one.

Complex and simple types can be declared (using the final attribute) as final with respect to some or all of the derivation methods.

Furthermore, complex types can also be declared (using the block attribute) as blocked with respect to the derivation either by restriction, by extension or by both. Such a type can be used within alternatives of a conditional element declaration. In this case, if the conditional element satisfies the condition corresponding to a blocked type, it must not have the xsi:type attribute making reference to a type definition which is derived by the blocked method (or methods) from that type.

Like a plain element declaration, through the optional block attribute of the `<xsd:element>` element, a conditional element declaration can regulate the use of the xsi:type attribute for the conditional element being declared. Indeed, setting block to "restriction", xsi:type must not make reference to a type definition that is derived by restriction from any of the type definitions present in the declaration; setting block to "extension", xsi:type must not make reference to a type definition that is derived by extension from any of the type definitions present in the declaration; finally, setting block to "#all", xsi:type cannot be used at all.

9 Substitution Groups

XML Schema provides a mechanism, called substitution groups, that allows elements to be substituted for other elements. More specifically, elements can be assigned to a special group of
elements that are said to be substitutable for a particular named element called the head element. Elements in a substitution group have to be global.

XML Schema imposes that elements of a substitution group must have the same type as the head element, or they can have a type that has been derived from the head element's type.

In SchemaPath, there is a conceptual obstacle when the head element is conditional. In that case, it has different types, each depending on a condition, so it is not clear how to force types of elements in the substitution group either to be the same as the head element's one, or to be derived from it.

A similar problem is also present when the head element is not conditional but the substitution group contains a conditional element. In fact, the type of the conditional element should be the same as the one of the head element, or derived from it, but the conditional element has more than one type.

Thus, substitution groups must not involve conditional elements. Consequently, declaring a conditional element as abstract is meaningless, and thus the abstract attribute must not appear within a conditional element declaration.

Furthermore, a conditional element can never be declared as to be blocked with respect to the substitution. Thus, the block attribute of a conditional element declaration has to be used to block the element with respect to only either extension, restriction or both.

Finally, the final attribute must not be used in a conditional element declaration, because its purpose is to impose restrictions on the types of the members of the substitution group headed by the declared element, but a conditional element cannot head any substitution group.

10 Identity-Constraints Definition

SchemaPath provides the possibility of defining identity-constraints in the same way as XML Schema does. Thus, all of the <xsd:unique>, <xsd:key> and <xsd:keyref> can be used within an element declaration. In particular, they can be used within a conditional element declaration, which may contain, as seen in 3.1, an optional sequence of <xsd:unique>, <xsd:key> and <xsd:keyref> elements after the last <xsd:alt> element.

SchemaPath introduces no new restriction on the identity-constraints definition, neither semantic nor syntactic. The XPath expressions used within the xpath attribute of the <selector> and <field> elements have the same restrictions as those imposed by XML Schema (see [TBMM01], section 3.11). Of course, such expressions can involve conditional elements and attributes, and the schema author has to take care of their compatibility with the conditions governing the types of the involved nodes.

An interesting observation comes from the semantics of identity-constraints provided by XML Schema. [TBMM01] points: “The equality and inequality conditions appealed to in checking these constraints apply to the value of the fields selected, so that for example 3.0 and 3 would be conflicting keys if they were both number, but non-conflicting if they were both strings, or one was a string and one a number. Values of differing type can only be equal if one type is derived from the other, and the value is in the value space of both”. Now, in SchemaPath the type of an element (attribute) may depend on a condition. Thus, it is possible that two elements (attributes) have the same type if a condition holds, but differing types if such a condition does not hold. Therefore, two keys may be conflicting depending on a condition.

11 Including, Importing and Redefining in SchemaPath

An important feature of XML Schema is its modularity, which allows schema authors to divide a large schema into sub-schemas and to reuse existing ones, eventually redefining parts of them.

SchemaPath inherits from XML Schema the three modularity mechanisms: inclusions, imports and redefinitions. Of course, such mechanisms can also be used when conditional elements or attributes are present, and have the same restrictions concerning target namespaces as those imposed by XML Schema and described in Sect. 2.5.
As it is in XML Schema, only simple types, complex types, groups and attribute groups can be redefined, and such redefinitions are restricted to be redefinitions of components in terms of themselves (see [TBMM01], Sect. 4.2.2). Schema authors should make sure that a redefinition doesn’t cause undesired side-effects. In particular, they should make sure that a redefinition doesn’t make the XPath expression specified in a condition of a conditional declaration meaningless. For example, consider the following schema (stored in the file sl.xsd):

```xml
<xsd:schema xmlns:xsd="http://www.cs.unibo.it/SchemaPath/1.0">
  <xsd:simpleType name="T0">
    <xsd:restriction value="xsd:string">
      <xsd:enumeration value="v1"/>
      <xsd:enumeration value="v2"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:element name="r">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="x">
          <xsd:alt cond="../@a='v1'" type="T1"/>
          <xsd:alt cond="../@a='v2'" type="T2"/>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="a" type="T0"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

and the following redefining schema

```xml
<xsd:schema xmlns:xsd="http://www.cs.unibo.it/SchemaPath/1.0">
  <xsd:redefine schemaLocation="sl.xsd">
    <xsd:simpleType name="T0">
      <xsd:restriction base="T0">
        <xsd:enumeration value="v1"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:redefine>
</xsd:schema>
```

In this case, the second alternative of the conditional element will never be chosen.

12 Annotations

SchemaPath inherits annotations from XML Schema. Thus, all of `<xsd:annotation>`, `<xsd:documentation>` and `<xsd:appinfo>` can be used to enrich a SchemaPath schema. In particular, when used for conditional declarations, the `<xsd:element>` and `<xsd:attribute>` elements contain an optional `<xsd:annotation>` element. Furthermore, the `<xsd:alt>` element too contains an optional `<xsd:annotation>` element.

As for all of the other components, the presence of an `<xsd:annotation>` element within a conditional declaration does not affect the validation phase.

13 Post Schema Validation Infoset

An important contribution of XML Schema is the PSVI. This is the information associated to the memory representation of the nodes of an XML document after having been parsed and vali-
dated. This information can then be used by a downstream application for performing specific computations.

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 [Rob, com] of embedding Schematron-like rules within an 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 assignments. 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 to reject 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.

14 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. We have obtained a formalization of SchemaPath, adapting that of XML Schema presented in [JS03]. In this dissertation we present just an hint to the formalization. The interested reader can find it in [MSV04b].

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.

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

---

1. 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.
Co-Constraints in SchemaPath

In this section we give some examples of constraints taken from the real-world, and show how they can be formalized by SchemaPath.

15.1 No Nesting of <a> Elements in XHTML

In Appendix B of the XHTML 1.0 recommendation [ea00], 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 one. For instance, the normative XHTML 1.0 strict DTD for strictly conforming XHTML document provides the following element type definitions:

```xml
<!ELEMENT a %a.content;>
```

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 [SM00]. Of course this rapidly leads to unmanageable specifications, given their size and complexity.

A possible solution in SchemaPath is the following one:

```xml
<xsd:element name="a">
  <xsd:alt cond=".//x:a" type="xsd:error"/>
  <xsd:alt type="x:a.type"/>
</xsd:element>
```

It uses a conditional declaration with two alternatives for the <a> element: the former assigns the xsd:error type whenever other <a> elements appear as descendants of the element being declared, while the latter is used to assign the type for inline elements (whose definition actually allows other <a> elements as descendants) in all other cases. The first alternative has a priority greater than the second one.

15.2 Variables in XSLT

In XSLT, a variable is represented by the <variable> element. In [Cla99] section 11.2, we find the following constraint:

If the variable-binding element has a select attribute, then the value of the attribute must be an expression and the value of the variable is the object that results from evaluating the expression. In this case, the content must be empty.
This means that the select attribute and the content are mutually exclusive.

DTD and XML Schema provide no way to enforce this constraint, while a SchemaPath solution follows:

```xml
<xsd:element name="variable">
  <xsd:alt cond="@select and (child::* or text()!='')" type="xsd:error"/>
  <xsd:alt type="xsl:variableType"/>
</xsd:element>
<xsd:complexType name="variableType" mixed="true">
  <xsd:sequence>
    <xsd:group ref="xsl:templateContent"/>
  </xsd:sequence>
  <xsd:attribute name="select" type="xsl:expr"/>
</xsd:complexType>
```

It defines only a type, where the select attribute is optional and the content allowed. The `variable` element is declared as conditional: a first alternative assigns the xsd:error type whenever the co-constraint is violated, whereas a second one assigns the defined type in all other cases.

### 15.3 Named templates in XSLT

In XSLT, within a `<template>` element match and name attributes may appear. The XSLT recommendation [Cla99] 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”.

In order to formalize such a constraint, SchemaPath defines only a type, where both match and name attributes are declared as optional. Then a conditional declaration is provided for the `<template>` element. Such a declaration assigns the xsd:error type whenever both match and name attributes are absent, and the defined type in all other cases. The described solution follows:

```xml
<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>
```

### 15.4 Elements in XML Schema

One of the requirements for XML Schema listed in [MM99] states that XML Schema should be self-describing, i.e., it should be possible to write an XML Schema schema that fully describes all of the syntactic constraints that a XML Schema document must observe.
On the other hand, there are some syntactic requirements imposed on XML Schema documents that cannot be described by XML Schema itself. For instance, section 3.3.3 of [TBMM01] imposes, in addition to those described by the normative XML Schema schema for schemas, the following conditions to an `<element>` element information item:

- default and fixed must not both be present.
- If the item’s parent is not `<schema>`, then all of the following must be true:
  - One of ref or name must be present, but not both.
  - If ref is present, then all of `<complexType>`, `<simpleType>`, `<key>`, `<keyref>`, `<unique>`, nillable, default, fixed, form, block and type must be absent, i.e. only minOccurs, maxOccurs, id are allowed in addition to ref, along with `<annotation>`.
- type and either `<simpleType>` or `<complexType>` are mutually exclusive.

For simplicity, we restrain an `<element>` element to only contain `<complexType>` or `<simpleType>` elements and to only have name, ref, type as possible attributes. Thus, we change the constraint above into:

- If the item’s parent is not `<schema>`, then all of the following must be true:
  - One of ref or name must be present, but not both.
  - If ref is present, then all of `<complexType>`, `<simpleType>`, and type must be absent, i.e., anything other than ref is not allowed.
- type and either `<simpleType>` or `<complexType>` are mutually exclusive.

There is a further constraint (which is described by the XML Schema schema for schemas) imposing that within an `<element>` element whose parent is `<schema>` the ref attribute must not appear and name is required.

In SchemaPath the solution is to write a single type definition and a single top-level conditional declaration for the `<element>` element. The type definition is a plain XML Schema type definition, and is satisfied by all of the valid global and local element declarations, but it does not enforce any co-constraint. On the other hand, each alternative of the conditional declaration but the last checks whether a co-constraint is violated, and in that case the xsd:error type is assigned. The last alternative is used to assign the defined type when the conditions of all other alternatives are not satisfied by the element. Such a solution follows:

```xml
<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"/>
```

45
15.5 FpML Validation Rules

FpML is an industry-standard protocol for complex financial products. In order to be correct, an FpML document must satisfy several constraints, among which a number of co-constraints. FpML 4.0 [FpM03] provides an official XML Schema specification (composed of more than one thousand element declarations) defining the structure of FpML documents. The schema is divided into modules, each defining the structure of a subset of the FpML elements. FpML 4.0 also defines a set of additional rules (known as validation rules), that are not enforced by the official XML Schema. The current version of the FpML 4.0 specification defines 56 validation rules just for the elements describing the Interest Rate Derivative products (IRD), whose module provides two hundred element declarations. These validation rules are expressed in natural language, and describe simple and complex relationships among the IRD elements.

A rather complex validation rule for IRD products, rule ird-23, applies to all `<stubCalculationPeriodAmount>` elements, and states:

"initialStub should only be present if the calculationPeriodDates element referenced by calculationPeriodDatesReference/@href contains at least one of firstPeriodStartDate and firstRegularPeriodStartDate."

SchemaPath can enforce the above constraint providing the following conditional declaration for the `<stubCalculationPeriodAmount>` element:

```xml
<xsd:element name="stubCalculationPeriodAmount">
  <xsd:alt cond="f:initialStub and
                 not(f:calculationPeriodDatesReference/@href=
                 //f:calculationPeriodDates[f:firstPeriodStartDate or
                 f:firstRegularPeriodStart/Date]/@id)"
      type="xsd:error"/>
  <xsd:alt type="StubCalculationPeriodAmount"/>
</xsd:element>
```

The condition in the first alternative checks whether the validation rule is violated, in which case the `xsd:error` type is assigned to the element. The second alternative is chosen only if the validation rule is not violated, and always assigns the `StubCalculationPeriodAmount` type, that is exactly the type defined within the XML Schema schema for FpML 4.0, and where the `<initialStub>` element is declared as optional.
Chapter 4

Implementation

SchemaPath draws important design decisions from XSLT: SchemaPath conditions use the same XPath expressions that XSLT accepts as predicates of template patterns; alternatives of conditional declarations have a priority, just as it is for template patterns, and when a conditional element or attribute matches more than one alternative with the same priority, a processor is left to decide whether to signal an error or to give precedence to the alternative occurring last in lexical order.

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 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 pre-processor 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 it 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\(^1\), and can be ported to any programming language in just a few minutes.

Our implementation can be tested on-line at the URLs

- [http://genesispc.cs.unibo.it:3333/schemapath.asp](http://genesispc.cs.unibo.it:3333/schemapath.asp) (APS and MSXML technologies), and

It can be downloaded for local tests from the first 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 stylesheet \(T'\) and the meta-stylesheet \(MT\). Our implementation has some well-known limits: they are explained in Sect. 3.

1 Transforming the Source and the Schema

The goal of \(MT\) and \(T''\) is to transform conditional elements and attributes into new elements and attributes manifesting the condition that holds with the highest priority, between those specified in the set of alternatives of the corresponding conditional declaration in \(S\). In brief, given a conditional element (attribute) declaration, \(MT\) creates a template for each alternative in the declaration, which is applied to all elements (attributes) whose name is that specified by the declaration, and satisfying the condition of the corresponding alternative.

On the other hand, \(T'\) transforms \(S\) into a correct XML Schema document \(S'\), mapping conditional declarations into plain XML Schema declarations which can be validated by those new elements and attributes created by \(T''\).

1.1 Conditional Elements

\(T''\) is constructed in a way such that every conditional element in \(X\) is inserted within a new element called **wrapper**, which is in turn inserted within another element, called **meta-wrapper**. By its name, the wrapper element manifests the condition that holds with the highest priority, among those specified in the set of alternatives of the corresponding conditional element declaration. Indeed, its name is obtained combining the one of the conditional element, the XPath expression of the holding condition with the highest priority, and such highest priority. Of course, an arbitrary XPath expression contains a number of characters that cannot be used in an XML element name. For this reason, these characters are actually escaped so that they can serve as an XML element name, using a dot followed by their hexadecimal value.

\(^1\) Excluding the back conversion of the validation errors.
The name of a meta-wrapper is obtained by the one of the conditional element, adding the string "mtWr" before it. Meta-wrapper and wrapper elements belong to the namespace of the conditional element.

To illustrate, consider the following SchemaPath snippet

```xml
<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>
```

and the two following instance document snippets.

```xml
<invoiceLine>
  <unit>items</unit>
  <quantity>125</quantity>
</invoiceLine>

<invoiceLine>
  <unit>meters</unit>
  <quantity>2.5</quantity>
</invoiceLine>
```

The first `<invoiceLine>` element is transformed into:

```xml
<invoiceLine>
  <unit>items</unit>
  <mtWrquantity>
    <wrquantity>0.5.2E.2E.0Aunit.40.3Ditems.3D>
      <quantity>125</quantity>
    </wrquantity>
  </mtWrquantity>
</invoiceLine>
```

while the second one into:

```xml
<invoiceLine>
  <unit>meters</unit>
  <mtWrquantity>
    <wrquantity>0.5.2E.2E.0Aunit.40.3Dmeters.3D>
      <quantity>2.5</quantity>
    </wrquantity>
  </mtWrquantity>
</invoiceLine>
```

When a conditional element does not satisfy any of the specified conditions, it is copied in X′ as it appears in X, without any wrapper element around it. In our example, this situation occurs when the `<unit>`'s value is neither "meters" nor "items", or when `<unit>` is not present at all.

Now, we show how T′ transforms S in order to take care of conditional element declarations. Each conditional element declaration is mapped into a meta-wrapper declaration. The meta-wrapper’s type is anonymously defined and consists of a choice among wrapper elements: there is a wrapper declaration for each alternative specified in the conditional declaration. The type of a wrapper element too is anonymously defined, and consists of a sequence of an element. This
element has the same name as the conditional element's one, and its type is that specified in the corresponding alternative.

In our example, the conditional element declaration is transformed into:

```xml
<xsd:element name="mtWrquantity">
  <xsd:complexType>
    <xsd:choice>
      <xsd:element name="wrquantity0.5.2E.2E.0Aunit.40.3Ditems.3D">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="quantity" type="xsd:integer"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="wrquantity0.5.2E.2E.0Aunit.40.3Dmeters.3D">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="quantity" type="xsd:decimal"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

Note that both the first and second transformations of the two conditional elements `<quantity>` performed by $T''$ and that we have shown above, validate against this declaration.

Now, suppose in $X'$ the conditional element `<quantity>` does not satisfy any of the specified conditions. As aforementioned, in this case it is just copied in $X'$ as it is in $X$. Thus, $X'$ does not validate against $S'$, because where a conditional element was expected from $S$, now a meta-wrapper is expected from $S'$, but this meta-wrapper is not present in $X'$, and thus a validation error occurs.

Now, suppose that `<quantity>` satisfies a condition, but its type is not the one expected from the corresponding alternative. A situation of this kind is showed below (the `<quantity>`'s type should be an integer, whereas it is a decimal).

```xml
<invoiceLine>
  <unit>items</unit>
  <quantity>12.5</quantity>
</invoiceLine>
```

Again, $X'$ does not validate against $S'$, because the wrapper element `<quantity>` is copied within, is declared in $S'$ as an element containing a `<quantity>` child whose type is an integer. For a formal proof of correctness of the implementation, see [MSV04b].

### 1.2 Conditional Attributes

While conditional elements can be inserted within wrappers and meta-wrappers by $T''$, conditional attributes cannot: it is well known that in XML, attributes cannot contain other attributes or elements. Thus, $T''$ maps a conditional attribute into another attribute whose name manifests the condition that holds with the highest priority among those specified in the corresponding declaration in $S$, and whose value is the one of the conditional attribute. In order to maintain a consistency in the terminology, such an attribute is called *wrapper*. A wrapper attribute belongs to the namespace of the corresponding conditional attribute.

To illustrate, given the SchemaPath snippet

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

the element <invoiceLine unit="items" quantity="123"/>
is transformed by $T''$ into:

<invoiceLine unit="items"
  wrquantity0.5.2E.2E.0A.2Funit.40.3Ditems.3D="123"/>

while the element <invoiceLine unit="meters" quantity="2.5"/>
is transformed into:

<invoiceLine unit="meters"
  wrquantity0.5.2E.2E.0A.2Funit.40.3Dmeters.3D="2.5"/>

As for elements, when a conditional attribute does not satisfy any of the specified conditions,
it is just copied in $X'$ without alterations.

Also $T'$ handles conditional attribute declarations differently from conditional element ones.
As previously discussed, roughly, a conditional element declaration is transformed into a choice
among other elements (wrappers). Unfortunately, XML Schema does not provide a choice opera-
tor for attributes, thus $T'$ maps a conditional attribute declaration into a list of (optional) wrapper
attribute declarations. There is a wrapper declaration for each alternative.

In our example, $T'$ produces the following output:

<xsd:element name="invoiceLine">
  <xsd:complexType>
    <xsd:attribute name="unit" type="unitType"/>
    <xsd:attribute name="wrquantity0.5.2E.2E.0A.2Funit.40.3Ditems.3D" type="xsd:integer"/>
    <xsd:attribute name="wrquantity0.5.2E.2E.0A.2Funit.40.3Dmeters.3D" type="xsd:decimal"/>
  </xsd:complexType>
</xsd:element>

Now, suppose that the <invoiceLine> element has a quantity attribute that does not
satisfy any of the specified conditions. In this case quantity is copied in $X'$ without alterations.
Thus, $X'$ does not validate against $S'$, because there, as shown above, the quantity attribute
is not declared.

Now, consider the <invoiceLine unit="items" quantity="one"/> element. In this
case, quantity matches the XPath expression of the first alternative, but its value is not an
integer as required. $T''$ maps the <invoiceLine> element into:

<invoiceLine unit="items"
  wrquantity0.5.2E.2E.0A.2Funit.40.3Ditems.3D="one"/>

which generates a validation error, because the "one" string does not belong to the value space
of the wrapper attribute's type (xsd:integer).

Finally, note that the conditional attribute in the example is declared as optional. Details on
required conditional attributes will be provided in the following subsection.

1.3 Occurrence Constraints
As known, in XML Schema both element and attribute declarations specify the so called occurrence
constraints, which regulate the allowed number of occurrences of the element or attribute
being declared.

In a conditional element declaration, occurrence constraints are specified by the minOccurs
and maxOccurs attributes within the <xsd:element> element. Since there is a one-to-one
relation between meta-wrapper declarations and conditional element declarations, it is natural
for our implementation to apply these constraints to the meta-wrapper declaration, moving the 
maxOccurs and minOccurs attributes within it.

For example, the conditional element declaration

```xml
<xsd:element name="x" maxOccurs="unbounded">
  <xsd:alt cond="@a='v1'" type="T1"/>
  <xsd:alt cond="@a='v2'" type="T2"/>
</xsd:element>
```

is transformed by $T'$ into:

```xml
<xsd:element name="mtWrx" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:choice>
      <xsd:element name="wrx0.5.2Fa.40.3Dv1.3D">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="x" type="T1"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="wrx0.5.2Fa.40.3Dv2.3D">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="x" type="T2"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

$T''$ does not need special code for the management of occurrence constraints of conditional 
elements: each conditional element is just put into the appropriate wrapper and meta-wrapper.

From the example above, note that where a sequence of conditional elements was required by 
$S$, a sequence of meta-wrappers is now required by $S'$.

For conditional attributes, occurrence constraints are specified by the use attribute within 
the <xsd:attribute> element. While occurrence constraints for conditional element declara-
tions are easily and naturally handled by $T'$, those for conditional attribute declarations require a 
special treatment by both $T'$ and $T''$, in particular, when the conditional attribute is mandatory 
(use="required").

Basically, $T''$ transforms each conditional attribute into a wrapper attribute. When the condition-
al attribute is declared as mandatory, a choice operator for the wrapper attributes within 
which copying the occurrence constraints of the conditional attribute declaration would be the 
perfect solution for $T'$. But this operator does not exist in XML Schema, and thus a conditional 
attribute declaration is transformed by $T'$ into a list of optional wrapper attribute declarations. 
Then, if the conditional attribute is required, a new mandatory attribute is declared, whose name 
only depends on the one of the conditional attribute, and whose value is fixed. This attribute 
manifests the obligatoriness of the conditional attribute. Consequently, $T''$ maps a required con-
ditional attribute into a pair of attributes: the wrapper and the attribute manifesting the obliga-
toriness.

To clarify, given the following SchemaPath fragment

```xml
<xsd:element name="invoiceLine">
  <xsd:complexType>
    <xsd:attribute name="unit" type="unitType"/>
    <xsd:attribute name="quantity" use="required">
      <xsd:alt cond="../@unit='items'" type="xsd:integer"/>
    </xsd:attribute>
  </xsd:complexType>
</xsd:element>
```

52
transforms it into:

\[
\begin{align*}
&<\text{invoiceLine unit="items" quantity="123"/>} \\
&<\text{quantity type="xsd:integer" default="123"/>} \\
&<\text{quantity type="xsd:decimal" fixed="2.5"/>} \\
&\text{}/>
\end{align*}
\]

Note that, if the quantity attribute was not present, the required reqquantity attribute would not be created and \(X'\) would not validate against \(S'\).
Of course, a conditional attribute can be declared as prohibited. In this case, all wrapper attributes are also declared as prohibited.

### 1.4 Value Constraints

As seen in Chap. 3, each alternative of a conditional declaration specifies its own value constraints.

For conditional element declarations, value constraints of each alternative are copied by \(T'\) within the element declaration occurring within the anonymous type definition of the corresponding wrapper.

Thus, given the conditional declaration

\[
\begin{align*}
&<\text{element name="quantity"/>} \\
&<\text{alt cond="../@unit='items'" type="xsd:integer"/>} \\
&<\text{alt cond="../@unit='meters'" type="xsd:decimal" default="123"/>} \\
&<\text{alt cond="../@unit='meters'" type="xsd:decimal" fixed="2.5"/>} \\
&\text{}/>
\end{align*}
\]

\(T'\) generates

\[
\begin{align*}
&<\text{element name="mtWrquantity"/>} \\
&<\text{complexType/>} \\
&<\text{choice/>} \\
&<\text{element name="wrquantity0.5.2E.2E.0A.2Funit.40.3Ditems.3D"/>} \\
&<\text{complexType/>} \\
&<\text{element name="quantity" type="xsd:integer"/>} \\
\end{align*}
\]
1. Transforming the Source and the Schema

T'' does not need special code to handle conditional elements whose declaration provides value constraints.

Unfortunately, our current implementation of SchemaPath does not correctly handle value constraints for attributes (see Sect. 3).

1.5 References to Global Elements and Attributes

Our implementation also handles references to global conditional elements and attributes. T'' handles global conditional element declarations in the same way as local ones, i.e., it maps them into meta-wrapper declarations. This implies that each reference to a global conditional element has to be transformed into a reference to the corresponding meta-wrapper.

For example, given the following reference

```xml
<xsd:element ref="x"/>
```

and assuming that the `<x>` global element is conditional, T'' transforms it into:

```xml
<xsd:element ref="wrX"/>
```

T'' does not need special code to handle references to global conditional elements.

A reference to a global conditional attribute is differently treated by T''. As a local one, a global conditional attribute declaration is transformed into a list of wrapper attribute declarations. Thus, each of its references is transformed into a sequence of global wrapper attribute references.

For example, given the global conditional attribute declaration

```xml
<xsd:attribute name="quantity">
  <xsd:alt cond="../@unit='items'" type="xsd:integer"/>
  <xsd:alt cond="../@unit='meters'" type="xsd:decimal"/>
</xsd:attribute>
```

and the reference `<xsd:attribute ref="quantity"/>`, T'' maps the reference into:

```xml
<xsd:attribute ref="wrquantity0.5.2E.2E.0A.2Funit.40.3Ditems.3D"/>
<xsd:attribute ref="wrquantity0.5.2E.2E.0A.2Funit.40.3Dmeters.3D"/>
```

Now, suppose that the reference to the `quantity` attribute is mandatory. In this case, T'' transforms it into:

```xml
<xsd:attribute ref="wrquantity0.5.2E.2E.0A.2Funit.40.3Ditems.3D"/>
<xsd:attribute ref="wrquantity0.5.2E.2E.0A.2Funit.40.3Dmeters.3D"/>
<xsd:attribute ref="reqquantity" use="required"/>
```

and also transforms the global conditional attribute declaration into:

```xml
<xsd:attribute name="wrquantity0.5.2E.2E.0A.2Funit.40.3Ditems.3D" type="xsd:integer"/>
<xsd:attribute name="wrquantity0.5.2E.2E.0A.2Funit.40.3Dmeters.3D"
```
Furthermore, \( T'' \) maps the element

\[
\text{<invoiceLine unit="meters" quantity="2.5"/>}
\]

into:

\[
\text{<invoiceLine unit="meters"}
\text{ wrquantity0.5.2E.2E.0A.2Funit.40.3Dmeters.3D="2.5"
\text{ reqquantity="required"/>}
\]

Note that if the quantity attribute was not present, the required reqquantity attribute would not be created by \( T'' \), and thus a validation error would be arisen.

### 2 The XSLT Code

In the previously section we have seen which transformations \( T'' \) and \( T' \) XSLT stylesheets operate on a SchemaPath \( S \) and on a document \( X \) in order to obtain an XML Schema \( S' \) and an XML document \( X' \) such that \( X \) validates against \( S \) if and only if \( X' \) validates against \( S' \). On the other hand, in this section we analyse, although at high level, the XSLT code of the three stylesheets \( MT \), \( T'' \), and \( T' \). We do not enter in details, but rather we just show the core code needed to handle conditional element and attribute declarations.

Thus in the following we show how \( MT \) maps conditional declarations into templates, and how \( T' \) maps them into plain and standard XML Schema declarations, providing a simpler and cleaner version of code than the actual one. In particular, the management of qualified and unqualified names, and the management of imported and included schemas are completely left out.

#### 2.1 How \( T'' \) is Generated by \( MT \)

\( T'' \) is automatically generated by \( MT \), which is basically an identity stylesheet, but it adds the necessary templates for the management of conditional elements and attributes.

\( MT \) creates a template for each alternative of every conditional declaration. The pattern of such a template is matched by all elements (attributes) of the XML instance document \( X \), whose name is the one specified in the corresponding element (attribute) declaration, and for which the XPath expression of the corresponding alternative evaluates to true when they are used as context nodes. \( MT \) handles conditional element declarations using the following meta-template:

\[
<xsl:template match="xsd:element/xsd:alt">
  <xsl:variable name="localname" select="string(../@name)"/>
  <xsl:variable name="cond">
    <xsl:choose>
      <xsl:when test="@cond">
        <xsl:value-of select="@cond"/>
      </xsl:when>
      <xsl:otherwise>true()</xsl:otherwise>
    </xsl:choose>
  </xsl:variable>
  <xsl:variable name="priority">
    <xsl:choose>
      <xsl:when test="@cond">
        <xsl:value-of select="@cond"/>
      </xsl:when>
      <xsl:otherwise>true()</xsl:otherwise>
    </xsl:choose>
  </xsl:variable>
</xsl:template>
\]
Such a template applies to all the alternatives of a conditional element declaration, and creates a template for each of them. The match attribute consists of the conditional element's name (stored in the $localname variable) followed by a predicate containing the XPath expression of the alternative (stored in the $cond variable). The priority attribute is set to the priority of the alternative (stored in the $priority variable). The rest of the meta-template is used to create the necessary code to insert a matching conditional element within the proper wrapper and meta-wraper. The name of a wrapper is computed by the parameterized compute_wrapper_name template.

Thus, given the following conditional element declaration

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

MT generates the following templates:

```xml
<xsl:template match="quantity[../unit='items']" priority="0.5">
  <xsl:element name="mtWrquantity"/>
  <xsl:copy>
    <xsl:apply-templates select="@*"/>
    <xsl:apply-templates/>
  </xsl:copy>
</xsl:element>
```
The XSLT Code

handles conditional attribute declarations similarly to conditional element ones, but the meta-template used for them has additional code specifically written for required attributes. The meta-template is shown below:

```xml
<xsl:template match="xsd:attribute/xsd:alt">
  <xsl:variable name="localname" select="string(.//@name)"/>
  <xsl:variable name="cond">
    <xsl:choose>
      <xsl:when test="@cond">
        <xsl:value-of select="@cond"/>
      </xsl:when>
      <xsl:otherwise>true()</xsl:otherwise>
    </xsl:choose>
  </xsl:variable>
  <xsl:variable name="priority">
    <xsl:choose>
      <xsl:when test="$alt/@priority">
        <xsl:value-of select="number($alt/@priority)"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:call-template name="calculate_default_priority">
          <xsl:with-param name="alt" select="."/>
          <xsl:with-param name="expr" select="$cond"/>
        </xsl:call-template>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:variable>
  <xsl:attribute name="@{$localname}[[{$cond}]]" priority="{$priority}"/>
</xsl:template>
```

It is very similar to the meta-template for conditional elements, but instead of making create a meta-wrapper, it makes create a wrapper attribute. Then, it checks whether the conditional attribute is declared as required. In such a case, it makes also create a further attribute manifesting such an obligatoriness. On the contrary, if the conditional attribute is not declared as required, it checks (through the named template find_required_attribute_use) whether it is global and whether there is a required reference to it. Also in this case it makes create the attribute manifesting obligatoriness.

Thus, given the following conditional attribute declaration

```
<xsd:attribute name="quantity" use="required">
  <xsd:alt cond="../@unit='items'" type="xsd:integer"/>
  <xsd:alt cond="../@unit='meters'" type="xsd:decimal"/>
</xsd:attribute>
```

MT generates:

```
<xsl:template match="@quantity[../@unit='items']" priority="0.5">
  <xsl:attribute name="wrquantity0.5.2E.2E.0A.2Funit.40.3Ditems.3D">
    <xsl:value-of select="."/>
  </xsl:attribute>
  <xsl:attribute name="reqquantity">required</xsl:attribute>
</xsl:template>

<xsl:template match="@quantity[../@unit='meters']" priority="0.5">
  <xsl:attribute name="wrquantity0.5.2E.2E.0A.2Funit.40.3Dmeters.3D">
    <xsl:value-of select="."/>
  </xsl:attribute>
  <xsl:attribute name="reqquantity">required</xsl:attribute>
</xsl:template>
```

2.2 How T' Transforms S

T' maps conditional declarations into standard and plain XML Schema declarations, so that S' may be validated by X'. The template transforming a conditional element declaration follows:

```
<xsl:template match="xsd:element[xsd:alt]">
  <xsl:variable name="localname" select="string(@name)="/>
  <xsd:element name="mtWr{$localname}">
    <xsl:apply-templates select="@minOccurs|@maxOccurs"/>
    <xsd:complexType>
      <xsd:choice>
        <xsd:element name="mtwr{$localname}"/>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
</xsl:template>
```
<xsl:for-each select="xsd:alt">
  <xsl:variable name="cond">
    <xsl:choose>
      <xsl:when test="@cond">
        <xsl:value-of select="@cond"/>
      </xsl:when>
      <xsl:otherwise>true()</xsl:otherwise>
    </xsl:choose>
  </xsl:variable>
  <xsl:variable name="priority">
    <xsl:choose>
      <xsl:when test="$alt/@priority">
        <xsl:value-of select="number($alt/@priority)"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:call-template name="calculate_default_priority">
          <xsl:with-param name="alt" select="."/>
          <xsl:with-param name="expr" select="$cond"/>
        </xsl:call-template>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:variable>
  <xsd:element>
    <xsl:attribute name="name">
      <xsl:call-template name="compute_wrapper_name">
        <xsl:with-param name="localname" select="$localname"/>
        <xsl:with-param name="priority" select="$priority"/>
        <xsl:with-param name="cond" select="$cond"/>
      </xsl:call-template>
    </xsl:attribute>
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="{$localname}"
          apply-templates select="@*"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsl:for-each>
</xsd:choice>
</xsd:complexType>
</xsd:element>
</xsl:template>

It is interesting to note the importance of the meta-wrapper declaration. Indeed, it could appear unnecessary, and one could argue that just a simple choice among wrapper declarations is sufficient. However, in this case it would not be possible to declare a conditional element within a <xsd:all> group. In fact, XML Schema requires <xsd:all> elements to contain just element declarations as child elements, and thus the presence of a choice operator within it would cause an incorrect XML Schema S'. That is the main reason why conditional elements are mapped into meta-wrappers.

Subtler is the template transforming conditional attribute declarations:

<xsl:template match="xsd:attribute[xsd:alt]">
<xsl:variable name="localname" select="string(@name)"/>
<xsl:for-each select="xsd:alt">
  <xsl:variable name="cond">
    <xsl:choose>
      <xsl:when test="@cond">
        <xsl:value-of select="@cond"/>
      </xsl:when>
      <xsl:otherwise>true()</xsl:otherwise>
    </xsl:choose>
  </xsl:variable>
  <xsl:variable name="priority">
    <xsl:choose>
      <xsl:when test="$alt/@priority">
        <xsl:value-of select="number($alt/@priority)"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:call-template name="calculate_default_priority">
          <xsl:with-param name="alt" select="."/>
          <xsl:with-param name="expr" select="$cond"/>
        </xsl:call-template>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:variable>
  <xsd:attribute>
    <xsl:attribute name="name">
      <xsl:call-template name="compute_wrapper_name">
        <xsl:with-param name="localname" select="$localname"/>
        <xsl:with-param name="priority" select="$priority"/>
        <xsl:with-param name="cond" select="$cond"/>
      </xsl:call-template>
    </xsl:attribute>
    <xsl:apply-templates select="../@use[string()!='required']"/>
    <xsl:apply-templates select="@*"/>
    <xsl:apply-templates/>
  </xsd:attribute>
</xsl:for-each>
<xsl:choose>
  <xsl:when test="@use='required'">
    <xsd:attribute name="req{$localname}"
      <xsl:copy-of select="@use"/>
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="required"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:attribute>
</xsl:when>
<xsl:otherwise>
  <xsl:variable name="found">
    <xsl:call-template name="find_required_attribute_use">
      <xsl:with-param name="localname" select="$localname"/>
    </xsl:call-template>
  </xsl:variable>
  <xsl:if test="$found!=''">
    60
It creates a wrapper attribute declaration for each alternative, and then it checks whether the conditional attribute is declared as required. In such a case, it creates a required attribute declaration manifesting the obligatoriness. The simple type of such an attribute is defined to be a restriction of the `xsd:string` type, and its value space consists just of the "required" string. On the contrary, if the conditional attribute declaration does not require obligatoriness, the template checks (through the same named template as that used by $MT$) whether it is global, and whether there is a required reference to it. Also in such a case, the attribute manifesting the obligatoriness is declared. However, it is declared with the same occurrence constraint as that of the conditional declaration (either optional or prohibited).

3 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.

3.1 Possible Naming Conflicts Generated by $T'$

As we have seen, given the name of a conditional element or attribute, $T'$ generates new names for new element or attribute declarations. For example, the name of a meta-wrapper is obtained adding the "mtWr" string before the conditional element’s name; the name of a wrapper attribute is obtained adding the "wr" string before the conditional attribute’s name, appending the priority to the result, and then adding the obtained string before the escaped form of the corresponding XPath expression.

During the creation of these new element (attribute) declarations, $T'$ assumes that there isn’t a non-conditional element (attribute) declaration with the same name and in the same context. If such a declaration exists, $S'$ will not be a correct XML Schema document.

For example, given the SchemaPath snippet

```
<xsd:complexType name="T">
  <xsd:sequence>
    <xsd:element name="x">
      <xsd:alt cond="@a='v1'" type="T1"/>
      <xsd:alt cond="@a='v2'" type="T2"/>
    </xsd:element>
    <xsd:element name="mtWrx" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```

$T'$ generates the following XML Schema:

```
<xsd:complexType name="T">
  <xsd:sequence>
    <xsd:element name="mtWrx"/>
  </xsd:sequence>
</xsd:complexType>
```
which is an ambiguous (and thus illegal) type definition.

3.2 The xsd:error Type

The xsd:error type is implemented as a simple type which is defined in every S' document created by T', and whose name is XXXerrorXXX. All references to the xsd:error type are consequently mapped into a reference to the XXXerrorXXX type.

The XXXerrorXXX type is defined as follows:

```xml
<xsd:simpleType name="XXXerrorXXX">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="xxxNoSuchValuexxx"/>
  </xsd:restriction>
</xsd:simpleType>
```

This implementation has two little problems. It makes the assumption that there isn’t another type in S whose name is XXXerrorXXX; and it also makes the assumption that the value of an attribute or element whose type is xsd:error, is not "xxxNoSuchValuexxx". In other words, xsd:error is not implemented as a type whose value space is actually empty, but it contains the "xxxNoSuchValuexxx" string.

3.3 Homonymous Local Conditional Elements and Attributes

A more severe limitation regards the interactions between local conditional elements (attributes) with the same name. In theory, homonymous local elements (attributes) 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 (attributes) 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:

```xml
<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>
```

```xml
<xsd:complexType name="anotherType">
  <xsd:sequence>
    <xsd:element name="quantity">
      <xsd:alt cond="../unit" type="xsd:string"/>
    </xsd:element>
    ...
  </xsd:sequence>
</xsd:complexType>
```
3 Implementation Limits

In this case, in $T''$ there are three templates: one matching all of the `<quantity>` elements having a sibling `<unit>` whose string value is "items"; another matching all of the `<quantity>` elements having a sibling `<unit>` whose string value is "meters"; and a third one matching all of the `<quantity>` elements just having a sibling `<unit>`. These templates have the same priority, 0.5.

Thus, all `<quantity>`s satisfying a condition in the first declaration also satisfy the condition in the second declaration, i.e., in $T''$ there are two matching templates for those elements. As stated in [Cla99], it is an error.

Our implementation is able to automatically detect those schemas that could be handled incorrectly due the aforementioned 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 definitions of both `anotherType` and `aType` complex types need to change to:

```xml
<xsd:complexType name="anotherType">
  <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>
```

With this trick, both the semantics of the first and second conditional declarations are preserved, and each possible wrapper that could be inserted around a `<quantity>` by $T''$ is declared within both `anotherType` and `aType` complex types in the $S''$ schema.

3.4 Value Constraints on Conditional Attribute Declarations

As aforementioned, our implementation does not correctly take care of value constraints in conditional attribute declarations.

For instance, consider the following declaration:

```xml
<xsd:attribute name="quantity">
  <xsd:alt cond="../@unit='items'" type="xsd:integer" default="123"/>
  <xsd:alt cond="../@unit='meters'" type="xsd:decimal"
```
3 Implementation Limits

and the `<invoiceLine unit="items"/>` element.

In this case, once $X$ has been validated against $S$, the PSVI should add the `quantity="123"` attribute to the `<invoiceLine>` element.

Conversely, $T'$ maps the above declaration into:

```xml
<xsd:attribute name="wrquantity05.2E.2E.2F.40unit.3D.27items.27" type="xsd:integer" default="123"/>
<xsd:attribute name="wrquantity05.2E.2E.2F.40unit.3D.27meters.27" type="xsd:decimal" default="2.5"/>
```

and $T''$ copies the `<invoiceLine>` element as it appears in $X$, because the `quantity` attribute is not present and thus there is no applicable template. This implies that, the PSVI adds the two wrapper attributes to the `<invoiceLine>` element in $X'$. Obviously, no validation error occurs, but the PSVI is not the one expected. This problem also holds for the `fixed` value constraint.

### 3.5 Identity-Constraint Definitions

As highlighted in the previous chapter, SchemaPath allows identity-constraint to be freely defined. On the other hand, our implementation has problems when the XPath expression within the `select` attribute of the `<selector>` and `<field>` elements involves conditional elements or attributes. Indeed, our implementation does modify the instance document, but not the XPath expression within a `select` attribute. Thus, such an expression could reference elements or attributes no longer present in $X'$, and thus, when evaluated by the XML Schema processor, it could identify a node set differing from the one expected. Of course, in such a case, the semantics of the identity constraints is changed.

### 3.6 Namespaces within the SchemaPath for SchemaPaths

In creating a SchemaPath schema for SchemaPath schemas, some points concerning the namespace handling of our implementation should be understood.

In SchemaPath, just as in XML Schema, schema components have a name, which consists of a namespace URI and a local part. In order to reference them, there are some attributes (`type`, `ref`, `base`, etc.) whose value is a qualified name. The prefix of such a qualified name is resolved to a namespace URI using actual namespace declarations in the scope of the element within which the attribute occurs.

On the other hand, each element within $S$ has a qualified name, whose prefix is bound either to the namespace URI of SchemaPath or to that of XML Schema. $T'$ maps it into an element having the same qualified name, but whose prefix is always bound to the namespace URI of XML Schema.

By this way, the namespace URI associated to the prefix of a qualified name specified within an attribute could change. For instance, assuming that the `xsd` prefix is associated to the SchemaPath namespace and that there is a type definition named (http://www.cs.unibo.it/SchemaPath/1.0, `altType`), the element

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

is transformed by $T'$ into an identical element, but the `xsd` prefix is associated to the XML Schema namespace, and thus the `type` attribute does not reference any type definition.

For this reason, in defining the SchemaPath schema for SchemaPaths, one should use two namespace declarations for the SchemaPath namespace and that there is a type definition named (http://www.cs.unibo.it/SchemaPath/1.0, `altType`), the element

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

where both `xsd` and `xs` are associated to the SchemaPath namespace URI. Indeed, the latter prefix still continues to be associated to the SchemaPath namespace also within the $S'$ XML Schema schema.
Note that the problem described above arises only when the target namespace of $S$ is the SchemaPath namespace URI.

3.7 Modification of PSVI
As discussed in Chapter 3, SchemaPath does not modify the content of the PSVI for valid documents. On the other hand, our current implementation does modify it, inserting conditional elements within wrappers and meta-wrapper, and mapping conditional attributes into wrapper attributes.
Chapter 5

Conclusions and Future Work

In this paper we have shown that it is possible to extend a grammar-based language introducing type assignments depending on values of the instance document. In particular, we have proved that it is possible to extend XML Schema introducing conditional declarations, i.e., declarations associating to elements or attributes one among a set of type definitions, according to conditions specified as XPath predicates. Such extension is called SchemaPath, and it is a conservative extension to XML Schema that allows the definition of a large class of co-constraints. We have shown its syntax, semantics, and a number of examples demonstrating its expressiveness, flexibility, and usefulness. We have also discussed about a simple implementation based on XSLT, and we have shown which limitations such implementation has.

A future work is studying the application of SchemaPath in the software engineering field. Rather than using SchemaPath to check the consistency of specifications (maybe a task beyond the possibility of the language), it might be used as a schema language for declaring syntactic requirements of those XML-based languages used in software engineering, such as XMI (XML Metadata Interchange) [Obj00], an XML format for the interchange of, among others, UML (Unified Modeling Language) specifications.

Surely, more work is required to improve the current implementation. Firstly, error back-conversion is needed. Indeed, at the moment, error messages provided are exactly those arisen by the underlying XML Schema processor. Thus, often they make reference to wrapper and meta-wrapper elements, which should be hidden to the user.

Moreover, as discussed in 3.7, being performed on a derived document and a derived schema, the actual XML Schema validation of our implementation modifies the PSVI of valid documents, thus breaking one of the most interesting properties of SchemaPath. To overcome this limitation, a second couple of XSLT stylesheets could be added to $T'$ and $M' T''$ and $M'T'$. The idea is, given a document $X$ and a SchemaPath $S$, to use $T'$ and $M'T$ to obtain a document $X'$ and an XML Schema schema $S'$ as discussed in this dissertation. Then, once $X'$ has been successfully validated against $S'$ in XML Schema, $T''$ could be applied to $S$ obtaining an XML Schema specification $S''$ where conditional declarations are mapped into plain declarations assigning the xsd:anyType type, and $M'T'$ could be applied to $S$ obtaining an XSLT stylesheet transforming $X$ into an equivalent document $X''$, where conditional elements are assigned the correct type through the xsi:type attribute. In this way, validating $X''$ against $S''$ in XML Schema, the expected PSVI is generated.

Another limitation of our implementation is the inability of XSLT template rules to distinguish among homonymous conditional elements declared in different complex types. A solution could be to abandon XSLT, and to adopt a programming language (e.g., Java) to transform the XML document $X$ into $X'$. However, the dependence of such transformation on the SchemaPath schema $S$ seems to heavily complicate the implementation of this solution.
References


REFERENCES


