Formal Semantics for SchemaPath

Paolo Marinelli   Claudio Sacerdoti Coen   Fabio Vitali

May 2003

Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7
40127 Bologna (Italy)

Formal Semantics for SchemaPath

Paolo Marinelli[1] **Claudio Sacerdoti Coen**[1] **Fabio Vitali**[1]

Technical Report

May 2003

Abstract

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

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

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

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

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

Chapter 1

Introduction

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

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

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

The number of schema languages created to overcome the limitations of DTDs is vast; in [2], a fairly authoritative source, 15 different schema languages for XML are listed besides DTDs, and at least one more, xlinkit [18], is missing. One of these languages, XML Schema [22, 5] is directly backed by the World Wide Web Consortium, and is being toted as the only and *official* schema language for XML documents, the natural substitution for DTDs. ISO, on the other hand, is active in DSDL [3], the international standardization of a couple of schema languages, RELAX NG [9] and Schematron [12], that are having a fair success despite being absolutely ignored by the W3C.

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

- *grammar-based languages*, by which document engineers create a whole context-free grammar according to top-down production rules in a specified formalism. XML Schema and RELAX NG, as well as DTDs themselves, fall into this category.
- *rule-based languages*, by which document engineers list the rules that the XML document must satisfy, providing either an open specification (all that is not forbidden is allowed) or a closed specification (all that is not allowed is forbidden) [23]. Schematron and xlinkit belong to this category.

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

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

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

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

Nonetheless, the problem of co-constraints (also known as *co-occurrence constraints*) is important and it is heavily felt for in many user communities. Several domain-specific standard languages based on XML include lamentations 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.

Indeed, 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 [11], yet it is expressed neither in the DTD nor in the XML Schema. XML Schema itself includes a number of co-constraints that cannot be expressed in the language, such as the mutual incompatibility of the `ref` and `name` attributes in an element or attribute definition.

Imposing constraints that cannot be expressed in the schema language of choice really **is** a serious shortcoming for interchange applications. The validation phase, in these applications, has the overall goal to ensure with minimum effort that the XML data does in fact conform to the 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 [19, 1] to embed a rule-based specification in Schematron within a grammar-based XML Schema document, so that the cooperation of both validations yields the desired control onto the XML documents. We feel that this approach is contrived and hackerish.

In this paper we propose an alternative solution, called *SchemaPath*. SchemaPath is a conservative extension of XML Schema obtained adding conditional type attributions to elements

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

and attributes. Conditional attributions can express co-constraints in a simple and compact syntax, and provide much more than co-constraints, as will be explained in the following. SchemaPath extends the XML Schema language with just one new construct and one new built-in type, so that understanding the differences and adopting the extensions is really trivial, especially comparing with the issues arising with the adoption of a completely different schema language such as Schematron.

In this paper we also provide a formal description of the SchemaPath semantics. The defintion of a formal semantics is important for any language, because it can clarify aspects not suitably dealt with by a prose specification. Consequently the task of writing a correct implementation of a language may be semplified by a formal specification.

Our formal semantics for SchemaPath is obtained adapting the formalization of XML Schema (which SchemaPath extends) presented in [13]. It allows us to prove that all the interesting properties XML Schema benefits also hold for SchemaPath.

A proof-of-concept implementation of a validation engine for SchemaPath has been developed to help understanding the advantages of our approach. It can be tested on-line at `http://genesispc.cs.unibo.it:3333/schemapath.asp`, or even downloaded (at the same address) for local use.

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

Chapter 2

Schema languages for XML

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

Besides DTDs [6] and XML Schema [22, 5], the most well known languages include RELAX [17], TREX [8], RELAX NG [9], Schematron [12], DSD [14] and xlinkit [18].

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

1 DTDs

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

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

Since DTDs precede temporally the advent of namespaces, they provide no support for qualified elements, except through a number of crippled and hardly robust hacks based on fixing the prefix associated to a namespace.

SGML DTDs are richer and more complex than XML DTDs, and in particular they have a feature that would be of wide interest in our discussion: *exclusions*. Exclusions specify that one or more elements cannot appear within an element or any of its children, providing a deep exception to the content model of an element. An example of exclusions is shown in Sect. 4. 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 [22, 5] aimed at substituting 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 DTDs.

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.

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

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

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

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

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

3 RELAX NG

RELAX NG [9] is a schema language for XML developed by an international working group, ISO/IEC JTC1/SC34/WG1. It is based on two preceding languages, TREX [8], designed by James Clark, and RELAX [17], designed by Murata Makoto.

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

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

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

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

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

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

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

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

4 DSD

DSD [14] is a schema language co-developed by AT&T Labs and BRICS. At the time of writing, there are two version of DSD: DSD 1.0 and, recently, DSD 2.0 (DSD2)[1].

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

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

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

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

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

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

The validation process is performed in a top-down traversal of the application document. If the node does not respect the constraint specified by its element definition, the validation process fails. Conversly, if each element node satisfies the constraint imposed by its element definition, the application document is valid.

Finally, during the conformance checking, default elements and attributes can be inserted in the application document.

A problem of DSD is that the boolean expressions used to describe context-dependent and conditional definitions can easily become very verbose.

5 Schematron

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

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

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

In fact, Schematron is commonly implemented as a meta-stylesheet, called *skeleton*. This

1. 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.

skeleton is applied to the Schematron schema and the resulting XSLT is in turn applied to the XML instance document.

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

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

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

6 xlinkit

xlinkit [18] is more than a schema language: it is an application service that provides rule-based link generation and checks the consistency of distributed web content.

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

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

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

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

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

Chapter 3

SchemaPath

In this chapter we illustrate the SchemaPath syntax and its semantics. We also show a few fragments of specifications that demonstrate its flexibility. In Sect. 13, we provide a formalization of SchemaPath.

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. While, just as it is in XML Schema, a declaration is an association of a name with *a* type definition, a *conditional declaration* is an association of a name with *one or more* type definitions, each depending on a condition. A *condition* is an XPath expression, which is evaluated on the instance document.

A *conditional element* is an element node of the instance document whose declaration is condtional. Analogously, a *conditional attribute* is an attribute node of the instance document, whose declaration is conditional.

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

1 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.

2 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.

In SchemaPath, a conditional declaration is an association of a name with one or more type definitions, each depending on a condition expressed as an XPath expression. A conditional element (attribute) validates against its declaration if and only if its name is the one specified in the declaration, it satisfies (at least) one of the specified conditions, and it also satisfies the type corresponding to the holding condition.

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.

```
<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 `xsd:string` 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.

2.1 Syntax

Here, we present the syntax of conditional declarations making use of the representation used in [22].

The syntactic structure of a conditional element declaration is:

```
<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`, `substituitionGroup`, `ref` and `type`), and allows no anonymous type definition.

The syntactic structure of a conditional attribute declaration is:

```
<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 : 0.5
  type = QName
  {any attributes with non-schema namespace . . .}>
  Content: annotation?
</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.

2.2 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 [7]). 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 [16]) 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"
          priority="0"/>
</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.

2.3 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,

```
<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 order to disambiguate these situations, each alternative of a conditional declaration has a *priority*, which is a real number (positive or negative). The priority of an alternative is explicitly set through the optional `priority` attribute of the `<xsd:alt>` element. When such an attribute is not present, it defaults to `0.5`.

When a conditional element or attribute matches more than one alternative with the same priority, SchemaPath adopt the same behaviour as that of XSLT. 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 the schema.

Thus, the declaration above can be rewritten as follows:

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

As another example, consider the rule: *"the `<price>` element is of type integer if the `<currency>` element is `"Yen"`, and decimal in all other cases in which the element is specified. But if the `<free_sample>` element is present, then the `<price>` element must be empty"*.

A naive implementation of these rules could give something like:

```
<xsd:element name="price">
 <xsd:alt cond="../free_sample"    type="empty"/>
 <xsd:alt cond="../currency='Yen'" type="xsd:integer"/>
 <xsd:alt cond="../currency"       type="xsd:decimal"/>
</xsd:element>
```

Unfortunately, all of the specified alternatives have the same priority (`0.5`) and, since the condition checking the presence of the `<free_sample>` element occurs last, if the SchemaPath processor does not signal the error, any `<price>` element having both `<currency>` and `<free_sample>` as siblings, is erroneously assigned the `xsd:decimal` type.

A better solution consists in setting different priorities for the alternatives, as follows:

```
<xsd:element name="price">
 <xsd:alt cond="../currency='Yen'" type="xsd:integer"
          priority="1"/>
 <xsd:alt cond="../currency"       type="xsd:decimal"/>
 <xsd:alt cond="../free_sample"    type="empty"
          priority="1.5"/>
</xsd:element>
```

This guarantees that the third alternative (whose priority is greater than the others) will be checked first and the type `empty` will be assigned to `<price>` whenever the `<free_sample>` element exists, regardless of the presence and value of the `<currency>` element. This also gurantees that the first condition (whose priority is greater than one of the second) will be checked before the second condition, thus assigning the `xsd:decimal` type only when the `<currency>` element is present and its string value is not `"Yen"`.

Precedently, we have discussed about the `cond` attribute. We have highlighted that when it is not present it defaults to `true()`, which is an always true condition. When other alternatives are specified, schema authors should make sure that such an always true condition is assigned a priority lower than those of the other conditions, forcing the SchemaPath processor to checking it till last.

3 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 an `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 happen 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.

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

4 Other issues in conditional declarations

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

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

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

4.2 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 attribute 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 elements 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:

```
<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:integer"
           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 supplied for the `quantity` attribute, in the PSVI.

Table 1. Defaults for a conditional attribute

Instance document	Value of the `quantity` attribute in the PSVI
`<invoiceLine unit='items'/>`	0
`<invoiceLine unit='meters'/>`	0.0
`<invoiceLine unit='other unit'/>`	Validation error!
`<invoiceLine unit='items' quantity="12"/>`	12
`<invoiceLine unit='items' quantity="12.2"/>`	Validation error!

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 constraint 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.

4.3 Nillable elements

SchemaPath allows to declare a conditional element as *nillable*. As value constraints, the nillableness 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 nillableness influence the content of an element. In fact, when an element is declared as nillable, 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 example of nillable element is provided by the following declaration:

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

According to this declaration, the `<quantity>` conditional element is nillable only when the `unit` attribute is set to `"items"`.

A more interesting example is represented by the following declaration

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

which requires that the `<x>` element is a string, whose length is specified by the `x-length` attribute 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 evaluates to true (the `string-length()` function returns `0`) and thus a validation error occurs.

Thus, in declaring a conditional element as nillable, the schema author has to make sure that the condition and the nillableness 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.

4.4 Occurence 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

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

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

<xsd:complexType name="myDecimal">
```

```
 <xsd:simpleContent>
  <xsd:extension base="xsd:decimal">
   <xsd:attribute name="unit" type="xsd:string"/>
  </xsd:extension>
 </xsd:simpleContent>
</xsd:complexType>
```

the following sequence of `<quantity>`s

```
<quantity unit="items">123</quantity>
<quantity unit="meters">1.3</quantity>
<quantity unit="meters">2.5</quantity>>
```

validates against the element declaration.

4.5 The `<xsd:all>` group

XML Schema provides the `<xsd:all>` element to declare unordered content. This element is present in SchemaPath, too. Its use is limited by the same restrictions as those imposed by XML Schema. In particular, all of its participating elements must not have the `maxOccurs` attribute greater than `1`.

In XML Schema, this restriction makes it *impossible* to impose a constraint like: *"within the `<x>` element, five `<y>`s and three `<z>`s must occur, but in no pre-defined order"*.

However, XML Schema allows to define the `<x>`'s type in a *weaker* form:

```
<xsd:element name="x" type="XT"/>

<xsd:complexType name="XT">
 <xsd:choice minOccurs="8" maxOccurs="8">
  <xsd:element name="y" type="YT"/>
  <xsd:element name="z" type="ZT"/>
 </xsd:choice>
</xsd:complexType>
```

which imposes that the `<x>` element has a sequence of eight child elements, and each of these children is either `<y>` or `<z>`. The only point which this weaker type definition leaves out, is the exact number of `<y>`s and the exact number of `<z>`s.

On the other, SchemaPath allows to impose also this further constraint using a conditional declaration for the `<x>` element:

```
<xsd:element name="x">
 <xsd:alt cond="number(child::y)!=5 or number(child::z)!=3"
          type="xsd:error"/>
 <xsd:alt type="XT" priority="0"/>
</xsd:element>
```

where the occurrence constraints for the `<x>` and `<y>` elements are moved within the `cond` attribute of the first alternative.

5 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.

5.1 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 [22, 5]. 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.

5.2 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 is a subset of the values 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 and are explained in Sect. 13. 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 that such a type has to be explicitly declared as final with respect to the derivation by restriction.

On the other, deriving types by extension does not arise any theroretical 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.

6 Using derived type 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 points to a type definition which is derived from the one corresponding to this condition.

For instance, consider the following SchemaPath:

```
<?xml version="1.0"?>
<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:complexType/>
     </xsd:element>
    </xsd:sequence>
   </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>

 <xsd:complexType name="T2">
  <xsd:complexContent>
   <xsd:restriction base="BT2">
    <xsd:attribute name="a" type="xsd:string" use="required"/>
   </xsd:restriction>
  </xsd:complexContent>
 </xsd:complexType>

</xsd:schema>
```

and the following XML document:

```
<?xml version="1.0"?>
<x xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" a="v1"
   xsi:type="T2"/>
```

In this case an error occurs, because the `xsi:type` makes reference to a type (`T2`) which is not derived from the *actual* base type (`BT1`).

In simple situations like this, it is trivial for the schema author to detect the correct condition (and thus the correct type), but it is not so when complex XPath expression are used.

7 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 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 rispect 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 deriavtion 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 that has to be validated against this declaration 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 bloked method (or methods) from the type specified in the declaration.

As 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 none of the type defintions present in the declaration; setting `block` to `"extension"`, `xsi:type` must not make reference to a type definition that is derived by extension from none of the type definitions present in the declaration; finally, setting `block` to `"#all"`, `xsi:type` cannot be used at all.

8 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*. The head element has 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 respecet 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 be the declared element, but a conditional element cannot head any substitution group.

9 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 2.1, 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 [22], 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. [22] 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.

10 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 an existing one, eventually redefining parts of it.

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 conditional attributes are present.

As in XML Schema, the include mechanism of SchemaPath requires that the target namespace of the included schema is the same as the target namespace of the including one. Once an external schema has been included, all of its components can be referenced.

Similarly, the redefine mechanism requires that the target namespace of the redefined schema is either the same as the target namespace of the redefining one or the null namespace. In the latter case, all of the included components became part of the redefining schema's target namespace. SchemaPath imposes the same restrictions on the definitions within the `<redefine>` element as those imposed by XML Schema. So, 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 themeselves (see [22], Sect. 4.2.2).

Obviously, all references to a component which has been redefined become references to the redefined version, and, as in XML Schema, the schema author should make sure that a redefinition doesn't cause undesired side-effects. In particular, the schema author 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 `s1.xsd`):

```
<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 follwing redefining schema

```
<xsd:schema xmlns:xsd="http://www.cs.unibo.it/SchemaPath/1.0">

 <xsd:redefine schemaLocation="s1.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.

Finally, in SchemaPath the only way to create a schema with components belonging to multiple namesapces is to use the import mechanism, which is activated through the `<xsd:import>` element. Its syntax and semantics are exactly as in XML Schema, as defined in [22], Sect 4.2.3.

11 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 declaration, the `<xsd:element>` and `<xsd:attribute>` elements contains 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 precence of an `<xsd:annotation>` element within a conditional declaration does not affect the validation phase.

12 Post Schema Validation Infoset

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

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

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

13 A Formalization of SchemaPath

In this section we present a formalization of an interesting fragment of SchemaPath obtained by dropping attributes and attribute specifications. We also drop other minor issues — such as default values and nillable elements — that are not sources of major differences between XML Schema and SchemaPath and that would heavily clutter the formalization. All these restrictions are inspired from [13], where the same formalization is given for XML Schema.

The formalization is useful in many ways. First of all it makes the semantics of SchemaPath presented informally in the previous sections clear and unambiguous. Secondly, it will allow us to prove in Sect. 3 the correctness of our prototypical implementation of a SchemaPath validator. Finally, it allows us to prove several theoretical results that already hold for XML Schema and that are extremely interesting from a practical point of view.

The first important result is that the *validation theorem* holds for SchemaPath. The validation theorem, introduced for XML Schema in [13], 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 validation theorem is the base lemma in the proof of the roundtripping and reverse roundtripping properties, that hold both for SchemaPath and XML Schema subject to a limited set of hypotheses.

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[1].

For the rest of the section, we assume the user to be familiar with [13], where the same formalization that we are pursuing is provided for XML Schema. Thus, we will detail here only the inference rules that are influenced by the extensions that SchemaPath makes over XML Schema. The reader must refer to [13] for all the other rules.

The formalization comprises the syntax of SchemaPath schema, the syntax of XML documents, the post-validation infosets, and the matching and validation rules of SchemaPath. Validation of an XML document against a schema returns a post-validation infoset. A post-validation infoset matches a SchemaPath type T if it describes a typed document whose declared type is a sub-type of the expected type T. Matching is a fundamental relation when schemas are used to type functions: any function whose argument is a type T — for instance an XSLT 2.0 named template — can be applied to any value whose type matches T.

In Sect. 3 we will extend our formalization to comprise our implementation and we prove its correctness.

13.1 XML Documents, Schemas and Post-Validation Infosets

In [13], XML Schema named types are modelled on regular tree grammars:

$$Type \quad ::= \quad ()$$

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 istances that satisfy the roundtripping and reverse roundtripping properties are introduced.

$$
\begin{array}{rcl}
 & | & \mathit{Empty} \\
 & | & \mathit{ItemType} \\
 & | & \mathit{Type}, \mathit{Type} \\
 & | & \mathit{Type} | \mathit{Type} \\
 & | & \mathit{Type}\ \mathit{Occurrence} \\
\mathit{Occurrence} & ::= & ? \mid + \mid * \\
\mathit{ItemType} & ::= & \mathit{ElementType} \mid \mathit{AtomicTypeName} \\
\mathit{ElementType} & ::= & \textbf{element}\ \mathit{ElementName}?\ \mathit{OfType}? \\
\mathit{OfType} & ::= & \textbf{of type}\ \mathit{TypeName}
\end{array}
$$

Notice that the grammar does not allow local type definitions for element declarations. Thus, to map a XML Schema named type to this formalization, one needs to generate a new fresh type name, declare it globally and use it in place of the local definition. Since the type name is fresh, the obtained schema is semantically equivalent to the original one.

To model SchemaPath named types we have to introduce a new type *Empty* that models the `Error` type and we have to allow guarded type definitions for element declarations:

$$
\begin{array}{rcl}
\mathit{Type} & ::= & \ldots \\
 & | & \mathit{Empty} \\
\mathit{ElementType} & ::= & \textbf{element}\ \mathit{ElementName}\ \mathit{OfGuardedType}^{*} \\
\mathit{OfGuardedType} & ::= & \mathit{OfType}\ \textbf{when}\ E \\
E & ::= & << \text{any XPath boolean expression that can} \\
 & & \quad \text{happear in an XSLT patter predicate} >>
\end{array}
$$

The list of guarded types in an element definitions is intended to be ordered according to the priority of the guard. Notice also that we drop the possibility of declaring anonymous *ElementType*s. Although quite interesting "*per se*", anonymous *ElementType*s are not present in XML Schema and were introduced in the formalization in [13]. Moreover, their exclusion from the formalization is not required by SchemaPath, being necessary only to prove the correctness of our implementation (in Sect. 3).

A XML Schema schema is modelled as a set of element and types definitions. A schema is *well-formed* when each element and type is defined at most once. In the rest of the report we will always assume schemas to be well-formed, without explicitly stating it. Definitions are formalized as follows:

$$
\begin{array}{rcl}
\mathit{Definition} & ::= & \textbf{define element}\ \mathit{ElementName}\ \mathit{OfType} \\
 & | & \textbf{define type}\ \mathit{TypeName}\ \mathit{TypeDerivation} \\
\mathit{TypeDerivation} & ::= & \textbf{restricts}\ \mathit{AtomicTypeName} \\
 & | & \textbf{restricts}\ \mathit{TypeName}\ \{\ \mathit{Type}\ \} \\
 & | & \textbf{extends}\ \mathit{TypeName}\ \{\ \mathit{Type}\ \}
\end{array}
$$

To model SchemaPath schemas, we need to extend the element definitions with guards. Moreover, we prefer to collect the set of element and type definitions that model a schema in a list where every element and type is defined at most once, whereas in [13] a set is preferred. According to the usual practice, we call the list *context* and we adopt for it the notation Γ. The reification of the set to a context will be useful in the formalization of our prototipical implementation given in Sect. 3.

$$
\Gamma \ ::= \ []
$$

$$\begin{array}{rcl}
 & \mid & \mathit{Definition}\ ;\ \Gamma \\
\mathit{Definition} & ::= & \ldots \\
 & \mid & \textbf{define element}\ \mathit{ElementName}\ \mathit{OfGuardedType}^{+}
\end{array}$$

As a form of syntactic sugar, we will abbreviate "**element** $\mathit{ElementName}\ \mathit{OfType}$ **when** `true`" to "**element** $\mathit{ElementName}\ \mathit{OfType}$" and "**define element** $\mathit{ElementName}\ \mathit{OfType}$ **when** `true`" to "**define element** $\mathit{ElementName}\ \mathit{OfType}$". This form of syntactic sugar is justified by the semantic equivalence of the element definitions guarded by `true()` in SchemaPath with the same unguarded element definitions in XML Schema. Moreover, as usual, the context $d_1\ ;\ \ldots\ ; d_n$ is a shorthand for $d_1\ ;\ \ldots\ ;\ d_n\ ;\ []$ and $\Gamma_1\ ;\ \Gamma_2$ stands for the concatenation of the two contexts Γ_1 and Γ_2.

A reader who knows XML Schema should easily understand how to map the concrete syntax of XML Schema to the previous definitions. For further details, see [13].

To model both PSVIs and XML documents, we introduce the class Value of possibly-typed values:

$$\begin{array}{rcl}
\mathit{Value} & ::= & () \\
 & \mid & \mathit{Item}(,\mathit{Item})* \\
\mathit{Item} & ::= & \mathit{Element} \\
 & \mid & \mathit{Atom} \\
\mathit{Element} & ::= & \textbf{element}\ \mathit{ElementName}\ \mathit{OfType}?\ \{\ \mathit{Value}\ \} \\
\mathit{OfType} & ::= & \textbf{of type}\ \mathit{TypeName} \\
\mathit{Atom} & ::= & \mathit{String}|\mathit{Integer}
\end{array}$$

Post-validation infosets are actually modelled as typed values, while XML documents are modelled by $\mathit{UntypedValue}$ that is the subclass of Value such that every OfType specifier is omitted.

13.2 Erasure, Matching and Validation

We will now redefine some of the judgements of [13]. Due to lack of space, we give only those inference rules that differ from [13].

13.2.1 "Erases to" judgement:

Value **erases to** $\mathit{UntypedValue}$
Informal reading: "*serializing the post-validation infoset* Value *yields the XML document* $\mathit{UntypedValue}$"

Since SchemaPath does not extend the post-validation infosets, the judgement inference rules are unchanged. See [13].

13.2.2 "Yields" judgement:

$\Gamma \vdash \mathit{ElementType}$ **yields** $\mathit{ElementName}\ \mathit{OfType}$ **for** Value
Informal reading: "Given an XML document fragment Value (or, alternatively, a post-validation infoset Value), an element specification $\mathit{ElementType}$, and a schema Γ, the judgement returns the name $\mathit{ElementName}$ and the type OfType that the root element of Value must have."
Informal example: *if ElementType models the declaration* `<xs:element ref="q:x"/>` *and* Γ *comprises the definition* `<xs:element name="q:x"><xs:alt cond="E" type="T"/></xs:element>` *and* `E` *is satisfied when evaluated using* Value *as the current XPath context, then* $\mathit{ElementName}$ *is* `q:x` *and* OfType *is* **of type** `T`.

The **yields** judgement is the one that is the most affected by the SchemaPath extensions. In particular, with respect to the corresponding judgement for XML Schema presented in [13], there are two new arguments. The first one is the context Γ; the second one is the value Value used as the XPath context when evaluating XPath conditions.

$$\frac{\begin{array}{c}(\textbf{define element } \mathit{ElementName}\ \mathit{OfGuardedTypes}) \in \Gamma \\ \mathit{OfGuardedTypes} = \mathit{OfType}_1 \textbf{ when } E_1 \ ; \ldots \ ; \ \mathit{OfType}_n \textbf{ when } E_n \\ \mathit{Value} \textbf{ erases to } \mathit{UntypedValue} \\ \textbf{xpath-eval}(E_i, \mathit{UntypedValue}) = \texttt{false()}\ \forall i \in \{1, \ldots, j-1\} \\ \textbf{xpath-eval}(E_j, \mathit{UntypedValue}) = \texttt{true()}\end{array}}{\Gamma \vdash \textbf{element } \mathit{ElementName} \textbf{ yields } \mathit{ElementName}\ \mathit{OfType}_j \textbf{ for } \mathit{Value}}$$

$$\frac{\begin{array}{c}\mathit{OfGuardedTypes} = \mathit{OfType}_1 \textbf{ when } E_1 \ ; \ldots \ ; \ \mathit{OfType}_n \textbf{ when } E_n \\ \mathit{Value} \textbf{ erases to } \mathit{UntypedValue} \\ \textbf{xpath-eval}(E_i, \mathit{UntypedValue}) = \texttt{false()}\ \forall i \in \{1, \ldots, j-1\} \\ \textbf{xpath-eval}(E_j, \mathit{UntypedValue}) = \texttt{true()}\end{array}}{\Gamma \vdash \textbf{element } \mathit{ElementName}\ \mathit{OfGuardedTypes} \textbf{ yields } \mathit{ElementName}\ \mathit{OfType}_j \textbf{ for } \mathit{Value}}$$

In the rest of the section, Γ_0 will stand for the context that models the "current" schema (which is supposed to be fixed).

13.2.3 "Matches" judgement:

Value **matches** Type
Informal reading: *"The post-validation infoset Value is an element of the type Type, and the declared type is a sub-type of Type"*

The only inference rule affected by the extensions SchemaPath makes over XML Schema is the rule that deals with elements:

$$\frac{\begin{array}{c}\Gamma_0 \vdash \mathit{ElementType} \textbf{ yields } \mathit{BaseElementName} \textbf{ of type } \mathit{BaseTypeName} \textbf{ for } \mathit{Value} \\ \mathit{BaseTypeName} \textbf{ resolves to } \mathit{Type} \\ \mathit{ElementName} \textbf{ substitutes for } \mathit{BaseElementName} \\ \mathit{TypeName} \textbf{ derives from } \mathit{BaseTypeName} \\ \mathit{Value} \textbf{ matches } \mathit{Type}\end{array}}{\textbf{element } \mathit{ElementName} \textbf{ of type } \mathit{TypeName}\ \{\ \mathit{Value}\ \} \textbf{ matches } \mathit{ElementType}}$$

13.2.4 "Strictly matches" judgement:

Value **strictly matches** Type
Informal reading: *"The post-validation infoset Value is an element of the type Type, which is also the declared type"*

The judgement **strictly matches** is a stronger version of the **matches** judgement. The only difference with respect to **matches** is that the derive relation is not considered: a typed value v **strictly matches** a type T when the type declared by v (i.e. the type of v in the PSVI) is exactly equal to the expected type T. The use of the judgement, which was not defined in [13], will be explained in the paragraph that deals with the **validate as** judgement.

$$\frac{\begin{array}{c}\Gamma_0 \vdash \mathit{ElementType} \textbf{ yields } \mathit{BaseElementName} \textbf{ of type } \mathit{BaseTypeName} \textbf{ for } \mathit{Value} \\ \mathit{BaseTypeName} \textbf{ resolves to } \mathit{Type} \\ \mathit{ElementName} \textbf{ substitutes for } \mathit{BaseElementName} \\ \mathit{Value} \textbf{ strictly matches } \mathit{Type}\end{array}}{\textbf{element } \mathit{ElementName} \textbf{ of type } \mathit{BaseTypeName}\ \{\ \mathit{Value}\ \} \textbf{ strictly matches } \mathit{ElementType}}$$

All the other inference rules are exactly equal to those of **matches** and can be found in [13].

13.2.5 "Validate as" judgement:

validate as $Type$ { $UntypedValue$ } $\Rightarrow$ $Value$
Informal reading:*"The XML document $UntypedValue$ is valid against the type specification $Type$. Its post-validation infoset is $Value$".*

The only inference rule affected by the extensions SchemaPath makes over XML Schema is the rule that deals with elements:

$$\frac{\begin{array}{c}\Gamma_0 \vdash ElementType\ \textbf{yields}\ BaseElementName\ \textbf{of type}\ BaseTypeName\ \textbf{for}\ UntypedValue \\ BaseTypeName\ \textbf{resolves to}\ Type \\ ElementName\ \textbf{substitutes for}\ BaseElementName \\ \textbf{validate as}\ Type\ \{\ UntypedValue\ \}\ \Rightarrow\ Value\end{array}}{\begin{array}{c}\textbf{validate as}\ ElementType\ \{\ \textbf{element}\ ElementName\ \{\ UntypedValue\ \}\ \}\ \Rightarrow \\ \textbf{element}\ ElementName\ \textbf{of type}\ BaseTypeName\ \{\ Value\ \}\end{array}}$$

All the other inference rules are exactly equal to the corresponding rules for XML Schema and they can be found in [13].

Notice that there is a difference between the conclusion of the rule and the conclusion given in [13], where $BaseTypeName$ is replaced by $TypeName$, which does not appear anywhere else in the rule. The prose just below the inference rule in [13] suggests the missing premise $TypeName$ **derives from** $BaseTypeName$. Although, that premise would not be correct: the fact that $UntypedValue$ can be validated as having the type $Type$ obtained resolving $BaseTypeName$ does not imply $Value$ **matches** $TypeName$. Moreover, even if the rule was correct, it would not be very useful when the judgement is used as a partial function and not as a relation, since the choice of $TypeName$ would be non-deterministic and the built post-validation infoset would be non-deterministic as well. We can only conclude that the rule in [13] is incorrect; we also claim our version to be the correct one.

13.3 The Validation Theorem

We are now ready to prove a version of the validation theorem stronger than the one proved in [13]: since we improved the definition of **validates** by restricting the possible types in the post-validation infoset, we have to replace the **matches** judgement in the statement with the stronger **strictly matches** judgement. The effect of these changes in the definition of *ambiguous for validation* are discussed after the proof of the theorem.

Theorem 1 (Validation Theorem)

$$\textbf{validate as}\ Type\ \{\ UntypedValue\ \}\ \Rightarrow\ Value$$

if and only if

$$Value\ \textbf{strictly matches}\ Type \quad \textit{and} \quad Value\ \textbf{erases to}\ UntypedValue$$

Proof:
The proof is by induction over derivations. We consider here only the cases that differ from [13].
If

$$\begin{array}{c}\textbf{validate as}\ ElementType\ \{\ \textbf{element}\ ElementName\ \{\ UntypedValue\ \}\ \}\ \Rightarrow \\ \textbf{element}\ ElementName\ \textbf{of type}\ BaseTypeName\ \{\ Value\ \}\end{array}$$

then

$$\textbf{element}\ ElementName\ \textbf{of type}\ BaseTypeName\ \{\ Value\ \}\ \textbf{strictly matches}\ ElementType$$

and

$$\textbf{element } ElementName \textbf{ of type } BaseTypeName \; \{ \; Value \; \} \\ \textbf{erases to} \\ \textbf{element } ElementName \; \{ \; UntypedValue \; \}$$

because:

- $Value$ **erases to** $UntypedValue$ *by induction hypothesis*
- $Value$ **strictly matches** *Type by induction hypothesis*

If

$$\textbf{element } ElementName \textbf{ of type } TypeName \; \{ \; Value \; \} \textbf{ strictly matches } ElementType$$

and

$$\textbf{element } ElementName \textbf{ of type } TypeName \; \{ \; Value \; \} \\ \textbf{erases to} \\ \textbf{element } ElementName \; \{ \; UntypedValue \; \}$$

then

$$\textbf{validate as } ElementType \; \{ \; \textbf{element } ElementName \; \{ \; UntypedValue \; \} \; \} \; \Rightarrow \\ \textbf{element } ElementName \textbf{ of type } BaseTypeName \; \{ \; Value \; \}$$

because:

- $Value$ **strictly matches** *Type (by analysis of the premises of the first hypothesis)*
- $Value$ **erases to** $UntypedValue$ *(by analylys of the premises of the second hypothesis)*
- *hence* **validate as** *Type* $\{ \; UntypedValue \; \} \; \Rightarrow \; Value$ *by induction hypothesis*

□

13.4 Rountripping and Reverse-Roundtripping

Following [13], we introduce the notions of unambiguous types for validation and unambiguous types for erasure, and we prove the roundtripping and reverse-roundtripping property for non ambiguous types.

Definition 1 (Unambiguous for validation) *A type Type is* unambiguous for validation *when for each untyped value $UntypedValue$ there is at most a value $Value$ such that*

$$\textbf{validate as } Type \; \{ \; UntypedValue \; \} \; \Rightarrow \; Value$$

The definition is syntactically equal to the one presented in [13], but, since we changed the definition of **validate as**, it is semantically stronger. In [13], indeed, if at least a type T' derives from a type T, then T is ambiguous for validation, as shown in the following example.

Example 1 *Let us consider the following schema formalization:*

define type *Base* **restricts** `xs:anyType` {(**element** *A* **of type** ())∗ }
define type *Derived* **restricts** *Base* {**element** *A* **of type** (), **element** *A* **of type** () }

Let u be the untyped value **element** *A* { () }, **element** *A* { () }
Let t be the typed value **element** *A* **of type** () { () }, **element** *A* **of type** () { () }

According to the definition of **validate as** *given in [13], the following two judgements hold:*

1. $\textbf{validate as element } X \textbf{ of type } Base \; \{\textbf{element } X \; \{ \; u \; \} \; \} \; \Rightarrow \; \textbf{element } X \textbf{ of type } Base \; \{ \; t \; \}$
2. $\textbf{validate as element } X \textbf{ of type } Base \; \{\textbf{element } X \; \{ \; u \; \} \; \} \; \Rightarrow \; \textbf{element } X \textbf{ of type } Derived \; \{ \; t \; \}$

The conclusion is that $Base$ is ambiguous for validation.

Generalizing the example, we conclude that every type derived at least once is ambiguous for validation in [13]. Note, however, that this is a property of the formalization and not of XML Schema: using our formalization of **validate as** and the corresponding **strictly matches** predicate, $Base$ is no longer ambiguous for validation even in the XML Schema formalization.

The definition of *unambiguous for erasure* is also unchanged:

Definition 2 (Unambiguous for erasure) *A type $Type$ is* unambiguous for erasure *when for each typed value $Value$ there is exactly one untyped value $UntypedValue$ such that*

$$Value \textbf{ erases to } UntypedValue$$

The set of types unambiguous for erasure for SchemaPath is equal to the set of types unambiguous for erasure for XML Schema. The same happens[2] for the unambiguous for validation property, provided that our definition of **validate as** is adopted also for the formalization of XML Schema.

Theorem 2 *The rountripping/reverse-roundtripping property holds provided that no type ambiguous for validation/ambiguous for erasure occurs in the SchemaPath schema.*

Proof:
The proof, which relies on the validation theorem, is syntactically equal to the one given in [13]. A posteriori, this is an obvious consequence of the fact that the post-validation infoset of the two languages is the same. □

We can conclude that SchemaPath is a conservative extension of XML Schema with respect to the roundtripping and reverse-roundtripping properties.

In Sect.3 we will prove the correctness of a prototypical implementation of SchemaPath. The formalization of the implementation will be based on the formalization of SchemaPath that we have described in this section.

2. The proof of the statement is trivial.

Chapter 4

Some examples of co-constraints in SchemaPath

In this chapter we provide a comparison between SchemaPath and other schema languages already described in Chapter 2. The comparison is by examples on co-constraints. We have chosen a number of "famous" examples, mostly taken by important W3C specifications. The solutions are reported in appendix 6. Note that such solutions are thought to formalize just the co-constraints explained in the following sections, and thus they omit the other syntatic aspects that the actual specification may require.

1 No nesting of <a> elements in XHTML

In Appendix B of the XHTML 1.0 recommendation [11], 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 defintion:

```
<!-- %Inline; covers inline or "text-level" elements -->
<!ENTITY % Inline "(#PCDATA | %inline; | %misc.inline;)*">

<!-- a elements use %Inline; excluding a -->
<!ENTITY % a.content
   "(#PCDATA | %special; | %fontstyle; | %phrase; | %inline.forms;
   | %misc.inline;)*">

<!-- content is %Inline; except that anchors shouldn't be nested -->
<!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 [20]. Of course this rapidly leads to unmanageable specifications, given their size and complexity.

RELAX NG has the same problem, and must use a similar solution with similar limitations.

SGML DTDs had the exact construct needed, exclusions, but it was later removed from XML DTDs. On the other hand, SchemaPath, xlinkit, Schematron and DSD all allow reasonable definitions of the constraint.

Both xlinkit and Schematron use a single rule that is applied to each `<a>` element of the instance document. The rule enforces that the element it is applied to do not contain other `<a>` elements.

DSD applies a boolean constraint to the definition of the `<a>` element. Such a constraint enforces, using a context pattern, that no other `<a>` element occurs in the context of the element being defined.

SchemaPath use 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.

The solutions described above are proposed in A. Within the solutions of xlinkit, Schematron and SchemaPath, the `x` prefix is bound to the namespace URI of XHTML. Due to their complexity and size, RELAX NG and XML Schema solutions are not reported.

2 Variables in XSLT

In XSLT, a variable is represented by the `<variable>` element. In [7] 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.

On the other, xlinkit enforces this constraint using a single consistency rule, which may be translated in natural language as *"for all `<variable>` elements, the presece of `select` attribute implies that both child and text nodes must be absent"*.

Schematron too uses a single rule which is applied to all `<variable>` elements. Such a rule reports an error message whenever the `select` attribute and either a child node or a text node are present.

DSD provides an element definition for the `<variable>` element. Such definition declares the `select` attribute as optional and assigns to the element being declared the template content. It also defines a conditional contraint which imposes that if the `select` attriubte is present then the content must be empty.

RELAX NG defines the `<variable>` element as a choice between a two patterns: one requiring the presence of the `select` attribute but disallowing any element, and the other allowing the presence of a content but prohibiting the `select` attribute.

Finally, SchemaPath 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.

The different solutions are presented in B. In some of them, the `xsl` prefix is used. It is assumed to be bound to the XSLT namespace URI.

3 Named templates in XSLT

In XSLT, within a `<template>` element `match` and `name` attributes may appear. The XSLT recommendation [7] describes the relation between `match` and `name` attributes as follows:

¡¡

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

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

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

xlinkit uses a consitency rule which directly formalize this new restatement of the constraint and that is applied to all `<template>` elements.

Schematron too uses a single rule which is applied to all `<template>` elements, but it formalize the constraint in another logically equivalent form: it outputs an error message when both `name` and `match` attributes are missing.

DSD provides a definition for the `<template>` element where both `name` and `match` attributes are declared as optional, and where a conditional constraint imposes that if `name` is absent then `match` must be present.

RELAX NG has no conditional statements, so all of the valid combinations of `match` and `name` attributes appear as arguments of a choice operator.

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 assignes the `xsd:error` type whenever both `macth` and `name` attributes are absent, and the defined type in all other cases.

Finally, neither XML Schema nor DTD provide a way to define this constraint.

In C we report the solutions described above. Again, the `xsl` prefix stands for the XSLT namesapce URI.

4 Elements in XML Schema

One of the requirements for XML Schema listed in [15] 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, there are some syntactic requirements imposed on an XML Schema document that cannot be described by XML Schema itself. For instance, section 3.3.3 of [22] 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.

Again, XML Schema and DTD cannot be used to specify such a constraint.

In xlinkit and in Schematron it can be defined declaring two rules: one for `<element>`s with a `ref` attribute, and one for `<element>`s without it. The former enforces that

- `<element>`'s parent is not `<schema>` and
- all of `type`, `name`, `<complexType>` and `<simpleType>` are absent.

The latter enforces that

- `name` is present and
- `type` and either `<simpleType>` or `<complexType>` are not both present.

DSD provides two element definitions: one for global elements and the other for local ones. Within the former the `name` attribute is declared as required, while `type` as optional. The `ref` attribute is not declared at all. The content is an optional choice among the `<simpleType>` and `<complexType>` elements. Finally, a condtional constraint enforces that if `type` is present them the content must be empty. Within the element definition for local element declarations, all of the `name`, `type` and `ref` attributes are declared as optional. Again, the content is an optional choice among the `<simpleType>` and `<complexType>` elements. A conditional constraint enforces that if `ref` is present then the content must be empty and both `type` and `name` must be absent; otherwise, if `type` is present the content must be empty.

RELAX NG defines two patterns: one for global elements and the other for local ones. The former requires the `name` attribute and uses a choice operator among the `type` attribute and an optional choice between the `<simpleType>` and `<complexType>` elements. The latter is a choice among the pattern described above and the `ref` attiribute.

In SchemaPath the solution is to write a single type defintion and a single top-level conditional delcaration for the `<element>` element. The type defintion 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, each alternative of the conditional declaration but the last checks whether a co-constraint is violated, in which case the `xsd:error` type is assigned. The last alternative is used to assgin the defined type when all of the conditions of the other alternatives are not satisfied by the element.

These solutions are shown in appendix D.

5 FpML validation rules

FpML (Financial products Markup Language) [4] is an industry-standard protocol for complex financial products. In order to be correct, an FpML document must satisfy several co-constraints. A list of them can be found in [21]. FpML explicitly refers to xlinkit for the validation of such rules.

One such co-constraint is that defined in rule `r3`: ¡¡

> In `%FpML_BusinessDayAdjustments`: neither `businessCentersReference` nor `BusinessCenters` must exist if and only if the value of `businessDayConvention` is `NONE`. `%FpML_BusinessDayAdjustments` defines `calculationPeriodDatesAdjustments`, `DateAdjustments`, `paymentDatesAdjustments`, and `resetDatesAdjustments`.

As usual, neither DTDs nor XML Schema provide any way to express these constraints, while xlinkit directly formalize it as it is described above.

Both RELAX NG and DSD allow to formalize this rule using three different patterns respectively content expressions: one where the `<bisunessDayConvention>` element cannot have the value `NONE` and where at least one of `<businessCentersReference>` and `<businessCenters>` elements is required; a second one where the `<bisunessDayConvention>` element can assume only the `NONE` value, and where neither `<businessCenters>` nor `<businessCentersRefernce>` are allowed; a third one, which is a choice among the first two.

Schematron uses two rules: one for elements in `%FpML_BusinessDayAdjustments` having the `<businessDayConvention>` element set to `'NONE'`, and the other for elements in `%FpML_BusinessDayAdjustments` having the `<businessDayConvention>` element set to a value other than `'NONE'`. The first rule reports an error if at least one of `<businessCentersReference>` and `<businessCenters>` is present. The second rule report an error when neither `<businessCentersReference>` nor `<businessCenters>` are present.

SchemaPath defines only a type for each element in `%FpML_BusinessDayAdjustments`. Such a type is validated by all elements satisfing the co-constraing, but also by those not obeying it. All of the elements in `%FpML_BusinessDayAdjustments` are declared as conditional: the first two alternatives of each declaration are used to assing the defined type whenever the co-constraint is satisfied, while the last alternative assigns the `xsd:error` type in all other cases.

The described solutions are reported in appendix E.

Chapter 5

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 the schema.

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.

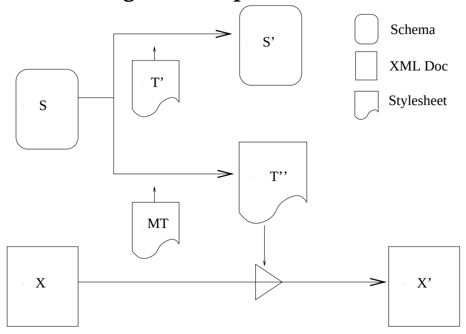
Just like a Schematron specification really is an XSLT transformation in disguise, our SchemaPath pre-processor is actually based on a couple of XSLT stylesheets, that create a derived XML Schema and a derived XML document that are the ones being used for the actual XML Schema validation.

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 shows 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

Figure 1. Implementation



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 URL `http://genesispc.cs.unibo.it:3333/schemapath.asp`, and can be downloaded for local tests from the same address. The downloadable package consists of a zip file containing an ASP script, the T" and MT stylesheets, and an XML document and a SchemaPath specification that can be used for testing.

In the next section we give further details on our implementation, explaining the operations performed by the stylesheet T' and the meta-stylesheet MT. Our implementation has some well-known limits: they are explained in Sect. 2.

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.

On the other, T' has to transform 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 condtional element declaration. Indeed, its name is obtained combining the one of the conditional element and the XPath expression of the holding condition. 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.

The meta-wrapper's name 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

```
<xsd:element name="invoiceLine">
 <xsd:complexType>
  <xsd:sequence>
```

1. Excluding the back conversion of the validation errors.

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

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

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

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

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

while the second one into:

```
<invoiceLine>
 <unit>meters</unit>
 <mtWrquantity>
  <wrquantity.2E.2E.0Aunit.40.3Dmeters.3D>
   <quantity>2.5</quantity>
  </wrquantity.2E.2E.0Aunit.40.3Dmeters.3D>
 </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 name of the conditional element's one and its type is that specified in the correspondig alternative.

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

```
<xsd:element name="mtWrquantity">
 <xsd:complexType>
  <xsd:choice>
   <xsd:element name="wrquantity.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="wrquantity.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 the second transformations of the two conditional elements `<quantity>` performed by T'' and that we have shown above, validate against this declaration.

Now, suppose that in X the conditional element `<quantity>` does not satisfy any of the specified condtions. As aforementioned, in this case it is just copied in X' as is. 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).

```
<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 Sect. **??**.

1.2 Conditional attributes

While conditional elements can be inserted in 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, between those specified in the corresponding declaration in S, and whose value is the one of the conditional attribute. In order to mantain a consistency in the terminology, such an attribute is called *wrapper*.

A wrapper attribute belongs to the namespace of the corresonding conditional attribute.

To illustrate, given the SchemaPath snippet

```
<xsd:element name="invoiceLine">
 <xsd:complexType>
  <xsd:attribute name="unit" type="xsd:string"/>
  <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"
             wrquantity.2E.2E.0A.2Funit.40.3Ditems.3D="123"/>
```

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

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

As for elements, when a conditional attribute does not satisfies 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 operator 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="xsd:string"/>
  <xsd:attribute name="wrquantity.2E.2E.0A.2Funit.40.3Ditems.3D"
                 type="xsd:integer"/>
  <xsd:attribute name="wrquantity.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 condtions. 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"
             wrquantity.2E.2E.2F.40unit.3D.27items.27="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 delcarations, 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

```
<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:

```
<xsd:element name="mtWrx" maxOccurs="unbounded">
 <xsd:complexType>
  <xsd:choice>
   <xsd:element name="wrx.2Fa.40.3Dv1.3D">
    <xsd:complexType>
     <xsd:sequence>
```

```
      <xsd:element name="x" type="T1"/>
     </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
   <xsd:element name="wrx.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, occurence constraints are specified by the `use` attribute within the `<xsd:attribute>` element. While occurrence constraints for conditional element declarations are easly 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 conditional 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 attrubute, and whose value is fixed. This attribute manifests the obligatoriness of the condtional attribute. Consequently, T'' maps a required conditional attribute into a pair of attributes: the wrapper and the attribute manifesting the obligatoriness.

To clarify, given the following SchemaPath fragment

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

T' transforms it into

```
<xsd:element name="invoiceLine">
 <xsd:complexType>
  <xsd:attribute name="unit" type="xsd:string"/>
  <xsd:attribute name="wrquantity.2E.2E.0A.2Funit.40.3Ditems.3D"
                 type="xsd:integer"/>
  <xsd:attribute name="wrquantity.2E.2E.0A.2Funit.40.3Dmeters.3D"
                 type="xsd:decimal"/>
  <xsd:attribute name="reqquantity" use="required">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
```

```
        <xsd:enumeration value="required"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:attribute>
 </xsd:complexType>
</xsd:element>
```

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

```
<invoiceLine unit="items"
             wrquantity.2E.2E.0A.2Funit.40.3Ditems.3D="123"
             reqquantity="required"/>
```

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

```
<xsd:element name="quantity">
 <xsd:alt cond="../unit='items'"  type="xsd:integer"
         default="123"/>
 <xsd:alt cond="../unit='meters'" type="xsd:decimal"
         fixed="2.5"/>
</xsd:element>
```

T' generates

```
<xsd:element name="mtWrquantity">
 <xsd:complexType>
  <xsd:choice>
   <xsd:element name="wrquantity.2E.2E.0Aunit.40.3Ditems.3D">
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element name="quantity" type="xsd:integer" default="123"/>
     </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
   <xsd:element name="wrquantity.2E.2E.0Aunit.40.3Dmeters.3D">
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element name="quantity" type="xsd:decimal" fixed="2.5"/>
     </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
  </xsd:choice>
 </xsd:complexType>
</xsd:element>
```

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. 2).

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

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

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

```
<xsd:element ref="mtWrx"/>
```

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 reference is transformed into a sequence of global wrapper attribute references.

For example, given the global conditional attribute declaration

```
<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:

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

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

and also transforms the global conditional attribute declaration into:

```
<xsd:attribute name="wrquantity.2E.2E.0A.2Funit.40.3Ditems.3D"
               type="xsd:integer"/>
<xsd:attribute name="wrquantity.2E.2E.0A.2Funit.40.3Dmeters.3D"
               type="xsd:decimal"/>
<xsd:attribute name="reqquantity">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="required"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:attribute>
```

Furthermore, T'' maps the element `<invoiceLine unit="meters" quantity="12.5"/>` into:

```
<invoiceLine unit="meters"
             wrquantity.2E.2E.0A.2Funit.40.3Dmeters.3D="12.5"
             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 arose.

1.6 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.

To illustrate, consider the following SchemaPath document

```
<xsd:schema xmlns:xsd="http://www.cs.unibo.it/SchemaPath/1.0">

 <xsd:element name="doc">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="invoiceLine" type="invoiceLineType"
                 maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:complexType name="invoiceLineType">
  <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:simpleType name="unitType">
  <xsd:restriction base="xsd:string">
   <xsd:enumeration value="items"/>
   <xsd:enumeration value="meters"/>
  </xsd:restriction>
 </xsd:simpleType>

</xsd:schema>
```

The meta-XSLT MT generates the stylesheet T'':

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:output method="xml" indent="yes"/>
 <xsl:template match="/|*|@*|text()" priority="-100">
  <xsl:copy>
   <xsl:apply-templates select="@*"/>
   <xsl:apply-templates/>
  </xsl:copy>
 </xsl:template>
 <xsl:template match="quantity[../unit='items']">
  <xsl:element name="mtWrquantity">
   <xsl:element name="wrquantity.2E.2E.0Aunit.40.3Ditems.3D">
    <xsl:copy>
     <xsl:apply-templates select="@*"/>
     <xsl:apply-templates/>
    </xsl:copy>
   </xsl:element>
  </xsl:element>
 </xsl:template>
 <xsl:template match="quantity[../unit='meters']">
  <xsl:element name="mtWrquantity">
```

```
   <xsl:element name="wrquantity.2E.2E.0Aunit.40.3Dmeters.3D">
    <xsl:copy>
     <xsl:apply-templates select="@*"/>
     <xsl:apply-templates/>
    </xsl:copy>
   </xsl:element>
  </xsl:element>
 </xsl:template>
</xsl:stylesheet>
```

If the `invoiceLineType` type was defined as:

```
<xsd:complexType name="invoiceLineType">
 <xsd:attribute name="unit" type="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>
```

the MT would had generated the following T'':

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:output method="xml" indent="yes"/>
 <xsl:template match="/|*|@*|text()" priority="-100">
  <xsl:copy>
   <xsl:apply-templates select="@*"/>
   <xsl:apply-templates/>
  </xsl:copy>
 </xsl:template>
 <xsl:template match="@quantity[../@unit='items']">
  <xsl:attribute name="wrquantity.2E.2E.0Aunit.40.3Ditems.3D">
   <xsl:value-of select="."/>
  </xsl:attribute>
 </xsl:template>
 <xsl:template match="@quantity[../@unit='meters']">
  <xsl:attribute name="wrquantity.2E.2E.0Aunit.40.3Dmeters.3D">
   <xsl:value-of select="."/>
  </xsl:attribute>
 </xsl:template>
</xsl:stylesheet>
```

Actually, T'' is a little subtler than the ones showed above, because it has to handle qualified and unqualified names.

When the `priority` attribute of the `<xsd:alt>` element is explicitly specified, it is copied within the corresponding `<xsl:template>`. On the other, when it is not explicitly specified, MT should insert the `priority` attribute set to `0.5` within the `<xsl:template>` element. However, it is not necessary, because all of the patterns are constructed in a way that the default priority of their template is already `0.5` (see [7] Section 5.5).

2 Implementation limits

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

2.1 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, a meta-wrapper's name is obtained adding the "mtWr" string before the conditional element's name; a wrapper attribute's name is obtained adding the wr string before the conditional attribute's name, and then adding the result 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:complexType>
    <xsd:choice>
     <!-- choice among wrappers -->
    </xsd:choice>
   </xsd:complexType>
  </xsd:element>
  <xsd:element name="mtWrx" type="xsd:string"/>
 </xsd:sequence>
</xsd:complexType>
```

which is an ambiguous (and thus illegal) type definition.

Our implementation does not try to recognize these situations and thus it does not generate any error or warning message. The only error the user will be noticed with is the one of the XML Schema processor.

2.2 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 `ts:XXXerrorXXX`, where the `ts` prefix is associated to the target namespace defined in the schema.

The `XXXerrorXXX` type is defined as:

```
<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, the `xsd:error` is not implemented as a type whose value space is actually empty, but it contains the `"xxxNoSuchValuexxx"` string.

2.3 Homonymous local conditional elements and attributes

A more severe limitation regards the interactions between local conditional elements (attributes) with the same name. In theory, local elements (attributes) have independent lives even if using the same names, 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, some conditions that should be checked could be skipped and some other conditions that should be skipped could be checked.

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

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

</xsd:complexType>
<xsd:complexType name="anotherType">
 <xsd:sequence>
  <xsd:element name="quantity">
   <xsd:alt cond="../unit"  type="xsd:string"/>
  </xsd:element>
  ...
 </xsd:sequence>

</xsd:complexType>
```

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>`. All of these templates has the same priority, `0.5`.

Thus, all of the `<quantity>` 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 [7], in such a case an XSLT processor either signals an error or applies the template rule that occurs last in the stylesheet. In the latter case, if the template corresponding to the alternative of the second declaration occurs last, all of the `<quantity>` elements that are valid against the first declaration will not be wrapped by the proper element.

Our implementation is able to automatically detect those schemas that could be handled incorrectly due the aforementioned limitations, and it notifies the user with a warning message.

However, a workaround exists for this limitation, even if it cannot be easily implemented. 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.

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

```
<xsd:complexType name="anotherType">
 <xsd:sequence>
  <xsd:element name="quantity">
   <xsd:alt cond="../unit"          type="xsd:string"
            priority="1"/>
```

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

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

2.4 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:

```
<xsd:attribute name="quantity">
 <xsd:alt cond="../@unit='items'"  type="xsd:integer"
          default="123"/>
 <xsd:alt cond="../@unit='meters'" type="xsd:decimal"
          default="2.5"/>
</xsd:attribute>
```

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.

Conversly, T' maps the above declaration into:

```
<xsd:attribute name="wrquantity.2E.2E.2F.40unit.3D.27items.27"
               type="xsd:integer" default="123"/>
<xsd:attribute name="wrquantity.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 both 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.

2.5 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 undetstood.

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 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. altType)`, the element

```
<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: one whose prefix is used for qualified elements within the schema, and the other for references to schema components.

Thus, the example above could be rewritten as follows:

```
<xsd:element name="alt" type="xs: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.

2.6 Modification of PSVI

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

3 Correctness and Completeness of the Implementation

In this section we formalize the prototypical implementation just described and we prove its correctness. The proof of correctness is partial since it is based on the formalization of SchemaPath given in Chap. 13, that only deals with a subset of SchemaPath. Thus, for instance, we will consider only schemas and instancecs without attributes.

Since we provide a faithful formalization of the implementation, our proof of correctness holds only for schemas that avoid the limitations described in Sect. 2.

The formalization of the implementation consists in two functions, T_S and T_D, that model respectively the transformation T' from a SchemaPath schema S to a XML Schema schema S' and the (meta)-transformation MT that given a SchemaPath schema S returns a transformation from SchemaPath document instances X to XML Schema document instances X'. The correctness of T_S and T_D means that for each SchemaPath document instance X and for each SchemaPath schema S, if X validates against S in SchemaPath then $(T_D(S))(X)$ validates against $T_S(S)$ in XML Schema. As a corollary, we obtain by modus tollens that if $(T_D(S))(X)$ is not valid against $T_S(S)$, then X is not valid against S in SchemaPath.

3.1 The T_D transformation:

$T_D : Context \rightarrow UntypedValue \rightarrow UntypedValue$

The T_D transformation is the meta-trasformation that, given a context Γ, returns a transformation from untyped values to untyped values:

```
$T_D(\Gamma) =$
 $\lambda UntypedValue.$
  match $UntypedValue$ with
      element $ElementName$ { $UntypedValue'$ }, $UntypedValue'' \Rightarrow$
         $L^\Gamma(\Gamma,\ ElementName,\ UntypedValue'), T_D(\Gamma)(UntypedValue'')$
  | $Atom, UntypedValue' \Rightarrow Atom, T_D(\Gamma)(UntypedValue')$
  | $() \Rightarrow ()$


$L^\Gamma([],\ ElementName,\ UntypedValue) =$
 element $ElementName$ { $T_D(\Gamma)(UntypedValue)$ }

$L^\Gamma($define type $TypeName\ TypeDerivation\ ;\Gamma',\ ElementName,\ UntypedValue) =$
 match $TypeDerivation$ with
     restricts $AtomicTypeName \Rightarrow L^\Gamma(\Gamma',\ ElementName,\ UntypedValue)$
  | restricts $TypeName$ { $Type$ }
  | extends $TypeName$ { $Type$ } $\Rightarrow$
      match $L^\Gamma_V(Type, ElementName, UntypedValue)$ with
         NotFound $\Rightarrow L^\Gamma(\Gamma',\ ElementName,\ UntypedValue)$
       | $v \Rightarrow v$

$L^\Gamma($define element $ElementName\ OfGuardedTypes\ ;\Gamma',\ ElementName',\ UntypedValue) =$
 if $ElementName = ElemementName'$ then
   $G^\Gamma(OfGuardedTypes, ElementName, UntypedValue)$
 else
   $L^\Gamma(\Gamma',\ ElementName',\ UntypedValue)$

$L^\Gamma_V((), ElementName, UntypedValue) =$ NotFound

$L^\Gamma_V(Empty, ElementName, UntypedValue) =$ NotFound

$L^\Gamma_V(ItemType, ElementName, UntypedValue) =$
 match $ItemType$ with
     $AtomicTypeName \Rightarrow$ NotFound
  | element $ElementName'\ OfGuardedTypes \Rightarrow$
      if $ElementName = ElemementName'$ then
        $G^\Gamma(OfGuardedTypes, ElementName, UntypedValue)$
      else
        NotFound

$L^\Gamma_V(Type_1, Type_2, ElementName, UntypedValue) =$
 match $L^\Gamma_V(Type_1)$ with
     NotFound $\Rightarrow L^\Gamma_V(Type_2)$
  | $v \Rightarrow v$

$L^\Gamma_V(Type_1|Type_2, ElementName, UntypedValue) =$
 match $L^\Gamma_V(Type_1)$ with
     NotFound $\Rightarrow L^\Gamma_V(Type_2)$
  | $v \Rightarrow v$

$L^\Gamma_V(Type\ Occurrence, ElementName, UntypedValue) = L^\Gamma_V(Type, ElementName, UntypedValue)$

$G^\Gamma([],\ ElementName,\ UntypedValue) =$
 element $ElementName$ { $T_D(\Gamma)(UntypedValue)$ }
```

```
$G^{\Gamma}(\textbf{of type } TypeName \textbf{ when } E\ ; OfGuardedTypes,\ ElementName,\ UntypedValue) =$
 $\textbf{if xpath-eval}(E, UntypedValue) = \texttt{true ()}$
 $\textbf{then}$
   $\textbf{element}\ < ElementName, \diamond >\ \{$
     $\textbf{element}\ < ElementName, E >\ \{$
       $\textbf{element}\ ElementName\ \{\ T_D(\Gamma)(UntypedValue)\ \}$
     $\}$
   $\}$
 $\textbf{else}$
   $G^{\Gamma}(OfGuardedTypes,\ ElementName,\ UntypedValue)$
```

3.2 The T_S transformation:

$T_S :\ Context \rightarrow Set\ of\ Definitions$

The T_S transformation maps a SchemaPath context to a set of definitions in XML Schema:

```
$T_S([]) = []$

$T_S(\textbf{define element}\ ElementName\ OfGuardedTypes;\ \Gamma) =$
     $\textbf{let}\ (Type, \Gamma') = T_G(ElementName,\ OfGuardedTypes)\ \textbf{in}$
     $\textbf{let}\ TypeName = \texttt{mk\_fresh\_name()}\ \textbf{in}$
       $\{\textbf{define element}\ < ElementName, \diamond >\ \textbf{of type}\ TypeName$
        $\textbf{define type}\ TypeName\ \{\ Type\ \}\}\ \cup\ \Gamma'\ \cup\ T_S(\Gamma)$

$T_S(\textbf{define type}\ TypeName\ TypeDerivation;\ \Gamma) =$
  $\textbf{let}\ (TypeDerivation', \Gamma')\ =\ T_R(TypeDerivation)\ \textbf{in}$
     $\{\textbf{define type}\ TypeName\ TypeDerivation'\}\ \cup\ \Gamma'\ \cup\ T_S(\Gamma)$

$T_G(ElementName,\ []) = (), \emptyset$
$T_G(ElementName,\ \textbf{of type}\ TypeName\ \textbf{when}\ E\ ;\ OfGuardedTypes) =$
     $\textbf{let}\ (Type, \Gamma) = T_G(ElementName,\ OfGuardedTypes)\ \textbf{in}$
     $\textbf{let}\ TypeName' = \texttt{mk\_fresh\_name()}\ \textbf{in}$
       $\textbf{element}\ < ElementName, E >\ \textbf{of type}\ TypeName'\ |\ Type,$
       $\{\textbf{define type}\ TypeName'\ \{\textbf{element}\ ElementName\ \textbf{of type}\ TypeName\}\}\ \cup\ \Gamma$

$T_R(\textbf{restricts}\ AtomicTypeName) =$
  $\textbf{restricts}\ AtomicTypeName, \emptyset$
$T_R(\textbf{restricts}\ TypeName\ \{\ Type\ \}) =$
  $\textbf{let}\ (Type', \Gamma)\ =\ T_T(Type)\ \textbf{in}$
    $\textbf{restricts}\ TypeName\ \{\ Type'\ \}, \Gamma$
$T_R(\textbf{extends}\ TypeName\ \{\ Type\ \}) =$
  $\textbf{let}\ (Type', \Gamma)\ =\ T_T(Type)\ \textbf{in}$
    $\textbf{extends}\ TypeName\ \{\ Type'\ \}, \Gamma$

$T_T(()) = (), \emptyset$

$T_T(Type_1, Type_2) =$
  $\textbf{let}\ (Type_1', \Gamma_1)\ =\ T_T(Type_1)\ \textbf{in}$
  $\textbf{let}\ (Type_2', \Gamma_2)\ =\ T_T(Type_2)\ \textbf{in}$
     $Type_1', Type_2',\ \Gamma_1 \cup \Gamma_2$

$T_T(Type_1 | Type_2) =$
  $\textbf{let}\ (Type_1', \Gamma_1)\ =\ T_T(Type_1)\ \textbf{in}$
  $\textbf{let}\ (Type_2', \Gamma_2)\ =\ T_T(Type_2)\ \textbf{in}$
     $Type_1' | Type_2',\ \Gamma_1 \cup \Gamma_2$
```

```
$T_T(Type\ Occurrence) =$
  $\textbf{let}\ (Type', \Gamma)\ =\ T_T(Type)\ \textbf{in}$
    $Type'\ Occurrence,\ \Gamma$

$T_T(AtomicTypeName) = AtomicTypeName, \emptyset$

$T_T(\textbf{element}\ ElementName) =$
  $\textbf{element}\ < ElementName, \diamond >, \emptyset$

$T_T(\textbf{element}\ ElementName\ OfGuardedTypes) =$
    $\textbf{let}\ (Type, \Gamma) = T_G(ElementName,\ OfGuardedTypes)\ \textbf{in}$
    $\textbf{let}\ TypeName = \texttt{mk\_fresh\_name()}\ \textbf{in}$
      $\textbf{element}\ < ElementName, \diamond >\ \textbf{of type}\ TypeName,$
      $\{\textbf{define type}\ TypeName\ \{\ Type\ \}\}\ \cup\ \Gamma$
```

3.3 Schema Validity and Implementation Correctness

Informally, a XML Schema schema is *valid* if there are no dandling references to undefined elements or types and if for every restriction **restricts** $TypeName\ \{\ Type\ \}$ every value that has type $Type$ has also type $TypeName$. The formal definition is based on the **validate as** judgement:

Definition 3 (Valid schema) *A XML Schema schema* Γ *is* valid *if*

1. *for each element type occurrence* **element** $ElementName$ *in* Γ *there is an element definition* **define element** $ElementName\ OfGuardedTypes$ *in* Γ
2. *for each element type occurrence*

$$\textbf{element}\ ElementName\ \textbf{of type}\ Type_1\ \textbf{when}\ E_1\ \dots\ \textbf{of type}\ Type_n\ \textbf{when}\ E_n$$

all the types $Type_1, \dots, Type_n$ *are defined in* Γ *by means of type definitions*

$$\textbf{define type}\ TypeName_i\ TypeDerivation_i$$

3. *for every restriction* **define type** $TypeName'$ **restricts** $TypeName\ \{\ Type\ \}$ *and for every untyped value* $UntypedValue$*, if*

$$\textbf{validate as}\ TypeName'\{\ UntypedValue\ \}\ \Rightarrow\ Value$$

then

$$\textbf{validate as}\ TypeName\{\ UntypedValue\ \}\ \Rightarrow\ Value'$$

for some typed values $Value$ *and* $Value'$*.*

The same definition holds also for SchemaPath schema. Our first important result is the preservation of validity:

Lemma 1 (Validity preservation) *Let* Γ *be a SchemaPath context. If* Γ *is valid then* $T_S(\Gamma)$ *is valid.*

Proof:
By inspection of the $T_S(\_)$ *rules.* □

Corollary 1 *Let* Γ *be a valid SchemaPath context,* $BaseTypeName$ *a type name and* $Type$ *a type expression such that* $BaseTypeName$ **resolves to** $Type$ *in* Γ*. Then* $T_S(\Gamma) \cup \pi_2(T_T(Type)) = T_S(\Gamma)$*.*

Preservation of validity is a fundamental property of our implementation, since invalid schemas are rejected by XML Schema validators.

Since our formalization is faithful to the implementation, we need to define a predicate that states that a SchemaPath schema avoids the limitations of our implementation:

Definition 4 (Schema without conflicts) *A SchemaPath schema S is* without conflicts *if for each element name $ElementName$, every local or global definition of $ElementName$ has the same guards.*

The "schema without conflicts" is the limitation already discussed in Sect. 2.3. Our current implementation is able to automatically detect schemas that suffer from this limitation (but it is unable to automatically fix them). The lack of conflicts will be an important hypothesis in the proof of correctness.

Notice that all the others limitations discussed in the previous sections have no effect on the formalization.

The following fundamental property holds for the T_S transformation.

Lemma 2 *Let Γ be a SchemaPath schema. Then $T_S(\Gamma)$ is well-formed.*

Proof:
The T_S transformation preserves the type definitions and introduces new definitions of types choosing only fresh type names. Moreover, all the element definitions in Γ are dropped.

Thus, to prove that $T_S(\Gamma)$ is well-formed, we just have to prove that all the element definitions introduced in $T_S(\Gamma)$ are well-formed, i.e. that each element is defined at most once.

By inspection of the inference rules of T_S, the only element definitions that can be introduced in $T_S(\Gamma)$ are of the form

$$\textbf{define element } < ElementName, \diamond > \textbf{ of type } TypeName$$

Moreover, such a definition is generated only when the corresponding definition

$$\textbf{define element } ElementName\ OfGuardedTypes$$

is found in Γ. Thus, since Γ is well-formed by hypothesis (i.e. at most one definition of $ElementName$ occurs in Γ), we conclude that $T_S(\Gamma)$ is also well-formed (i.e. at most one definition of $< ElementName, \diamond >$ occurs in $T_S(\Gamma)$). $\square$

We are now ready to state the correctness theorem.

Theorem 3 (Correctness of the implementation)
If **validate as** $Type$ { $UntypedValue$ } $\Rightarrow$ $Value$ *according to a valid SchemaPath context Γ without conflicts such that* $\Gamma;$ **define type** $\diamond$ { $Type$ } *is valid and without conflicts, then there exists a value $Value'$ such that*

$$T_S(\Gamma) \cup \pi_2(T_T(Type)) \vdash \textbf{validate as } \pi_1(T_T(Type)) \ \{ \ T_D(\Gamma)(UntypedValue) \ \} \ \Rightarrow \ Value'$$

Note: the technical hypothesis that requires $\Gamma;$ **define type** $\diamond$ { $Type$ } to be without conflicts is just a consequence of the formalization that does not require $Type$ to occur in Γ. In SchemaPath, instead, $Type$ "always occurs" in Γ, since a document is validated against a schema and not against a schema and a type for the root. Therefore the technical hypothesis is always satisfied in the real world model.
Proof:
We proceed by structural induction over $UntypedValue$.

Since T_T and T_D performs recursion over values and types until the case of an element is met, the case of the empty sequence, the case of the atomic value and the inductive case of the sequence operator can be trivially proved by applying the inductive hypotheses.

The only case left is that of an element instance. Thus we need to consider the value

$$\textbf{element}\ ElementName\ \{\ UntypedValue'\ \}$$

matched against the element type $ElementType$ such that

$$\Gamma \vdash ElementType\ \textbf{yelds}\ ElementName\ \textbf{of type}\ BaseTypeName\ \textbf{for}\ UntypedValue' \tag{1}$$

and

$$BaseTypeName\ \textbf{resolves to}\ Type' \tag{2}$$

By inspection of the inference rules of the **validate as** *judgement we know that*

$$\textbf{validate as}\ Type'\ \{\ UntypedValue'\ \} \Rightarrow Value'$$

Thus, by induction hypothesis, we also know that

$$T_S(\Gamma) \cup \pi_2(T_T(Type')) \vdash \textbf{validate as}\ \pi_1(T_T(Type'))\ \{\ T_D(\Gamma)(UntypedValue')\ \}\ \Rightarrow\ Value'' \tag{3}$$

for some value $Value''$.

Moreover, by inspection of the inference rule of the **yelds** *judgement, from 1 we deduce that one of the following two cases holds:*

- *Case* $\boldsymbol{C}_1$

$$\begin{array}{l}
ElementType = \textbf{element}\ ElementName \\
(\textbf{define element}\ ElementName\ OfGuardedTypes) \in \Gamma \\
OfGuardedTypes = OfType_1\ \textbf{when}\ E_1\ ;\ \ldots\ ;\ OfType_n\ \textbf{when}\ E_n \\
\textbf{xpath} - \textbf{eval}(E_i, UntypedValue') = \texttt{false()}\ \forall i \in \{1, \ldots, j-1\} \\
\textbf{xpath} - \textbf{eval}(E_j, UntypedValue') = \texttt{true()} \\
OfType_j = \textbf{of type}\ BaseTypeName
\end{array}$$

- *Case* $\boldsymbol{C}_2$

$$\begin{array}{l}
ElementType = \textbf{element}\ ElementName\ OfGuardedTypes \\
OfGuardedTypes = OfType_1\ \textbf{when}\ E_1\ ;\ \ldots\ ;\ OfType_n\ \textbf{when}\ E_n \\
\textbf{xpath} - \textbf{eval}(E_i, UntypedValue') = \texttt{false()}\ \forall i \in \{1, \ldots, j-1\} \\
\textbf{xpath} - \textbf{eval}(E_j, UntypedValue') = \texttt{true()} \\
OfType_j = \textbf{of type}\ BaseTypeName
\end{array}$$

We will now analyse the two cases independently.

- *Case* $\boldsymbol{C}_1$
  By hypothesis we know that

$$\begin{array}{l}
\textbf{validate as}\ ElementType\ \{\ \textbf{element}\ ElementName\ \{\ UntypedValue'\ \}\ \}\ \Rightarrow \\
\quad \textbf{element}\ ElementName\ \textbf{of type}\ BaseTypeName\ \{\ TypedValue\ \}
\end{array}$$

  By definition of T_D we have

$$\begin{array}{l}
T_D(\Gamma)(\textbf{element}\ ElementName\ \{\ UntypedValue'\ \}) = \\
\quad L^\Gamma(\Gamma, ElementName, UntypedValue')
\end{array}$$

  We know that $(\textbf{define element}\ ElementName\ OfGuardedTypes) \in \Gamma$. *Therefore, by inspection of the definition of L^Γ, either*

$$\begin{array}{l}
L^\Gamma(\Gamma, ElementName, UntypedValue') = \\
\quad G^\Gamma(OfGuardedTypes, ElementName, UntypedValue')
\end{array}$$

or

$$\begin{array}{l} L^{\Gamma}(\Gamma, ElementName, UntypedValue') = \\ \ L_V^{\Gamma}(Type_x, ElementName, UntypedValue') = \\ \ L_V^{\Gamma}(\textbf{element}\ ElementName\ OfGuardedTypes') = \\ \ G^{\Gamma}(OfGuardedTypes', ElementName, UntypedValue') \end{array}$$

for some type $Type_x$ and some list of guarded types $OfGuardedTypes'$.

In the second case, since Γ is without conflicts by hypothesis, we know that the guards of $OfGuardedTypes$ and those of $OfGuardedTypes'$ must be the same.

By inspection of the definition of G^{Γ}, we know that G^{Γ} only inspects the guards of its first argument and not the associated types.

Thus, in both cases, we conclude that:

$$\begin{array}{l} L^{\Gamma}(\Gamma, ElementName, UntypedValue') = \\ \ G^{\Gamma}(OfGuardedTypes, ElementName, UntypedValue') \end{array}$$

Since $\textbf{xpath}-\textbf{eval}(E_i, UntypedValue') = \texttt{false()}\ \forall i \in \{1, \ldots, j-1\}$ *and* $\textbf{xpath}-\textbf{eval}(E_j, UntypedValue') = \texttt{true()}$ *then by definition of G^{Γ}*

$$\begin{array}{l} G^{\Gamma}(OfGuardedTypes, ElementName, UntypedValue') = \\ \quad \textbf{element}\ < ElementName, \diamond >\ \{ \\ \quad\quad \textbf{element}\ < ElementName, E_j >\ \{ \\ \quad\quad\quad \textbf{element}\ ElementName\ \{\ T_D(\Gamma)(UntypedValue')\}\}\} \end{array}$$

By definition of T_T we have

$$\begin{array}{l} \pi_1(T_T(ElementType)) = \\ \ \pi_1(T_T(\textbf{element}\ ElementName)) = \\ \ \pi_1(\textbf{element}\ < ElementName, \diamond >, \emptyset) = \\ \ \textbf{element}\ < ElementName, \diamond > \end{array}$$

and $\pi_2(T_T(ElementType)) = \emptyset$.

Thus, to prove the thesis, we have to show that

$$\begin{array}{l} T_S(\Gamma) \vdash \\ \quad \textbf{validate as element}\ < ElementName, \diamond >\ \{ \\ \quad\quad \textbf{element}\ < ElementName, \diamond >\ \{ \\ \quad\quad\quad \textbf{element}\ < ElementName, E_j >\ \{ \\ \quad\quad\quad\quad \textbf{element}\ ElementName\ \{\ T_D(\Gamma)(UntypedValue')\}\}\} \\ \quad \}\ \Rightarrow\ Value' \end{array}$$

for some value $Value'$.

Since $(\textbf{define element}\ ElementName\ OfGuardedTypes) \in \Gamma$ *then, by definition of T_S,*

$$(\textbf{define element}\ < ElementName, \diamond >\ \textbf{of type}\ TypeName'') \in T_S(\Gamma)$$

where $TypeName'' \notin \Gamma$ and

$$(\textbf{define type}\ TypeName''\ \{Type''\}) \in T_S(\Gamma)$$

and

$$(Type'', \Gamma') = T_G(ElementName, OfGuardedTypes) \tag{4}$$

where Γ' is a sublist of $T_S(\Gamma)$.

Thus

$$\begin{array}{l} T_S(\Gamma) \vdash \\ \quad \textbf{element}\ < ElementName, \diamond >\ \textbf{yelds}\ < ElementName, \diamond > \\ \quad \textbf{of type}\ TypeName''\ \textbf{for} \\ \quad\quad \textbf{element}\ < ElementName, \diamond >\ \{ \\ \quad\quad\quad \textbf{element}\ < ElementName, E_j >\ \{ \\ \quad\quad\quad\quad \textbf{element}\ ElementName\ \{\ T_D(\Gamma)(UntypedValue')\}\}\} \end{array} \tag{5}$$

and

$$TypeName''\ \textbf{resolves to}\ Type'' \tag{6}$$

Thus, to prove the thesis, we can just pick the appropriate inference rule of the **validate as** *judgement and, using 5 and 6, reduce the thesis to proving*

$$\begin{array}{l} T_S(\Gamma) \vdash \\ \quad \textbf{validate as}\ Type''\ \{ \\ \quad\quad \textbf{element}\ < ElementName, E_j >\ \{ \\ \quad\quad\quad \textbf{element}\ ElementName\ \{\ T_D(\Gamma)(UntypedValue')\}\} \\ \quad \}\ \Rightarrow\ Value''' \end{array}$$

for some value $Value'''$.

By 4 and by definition of T_G we have

$$\begin{array}{l} (Type'', \Gamma') = \\ \quad T_G(ElementName, OfGuardedTypes) = \\ \quad \textbf{element}\ < ElementName, E_1 >\ \textbf{of type}\ TypeName'_1\ | \\ \quad \ldots\ | \\ \quad \textbf{element}\ < ElementName, E_n >\ \textbf{of type}\ TypeName'_n, \\ \quad \{\textbf{define type}\ TypeName'_1\ \{\ \textbf{element}\ ElementName\ OfType_1\}, \\ \quad\ \ldots, \\ \quad\ \textbf{define type}\ TypeName'_n\ \{\ \textbf{element}\ ElementName\ OfType_n\}\} \end{array} \tag{7}$$

In particular $TypeName'_j$ **resolves to element** *$ElementName\ OfType_j$ that by hypothesis is equivalent to*

$$TypeName'_j\ \textbf{resolves to element}\ ElementName\ \textbf{of type}\ BaseTypeName$$

From 7, by applying the appropriate inference rules for **validate***, we can reduce the thesis to*

$$\begin{array}{l} T_S(\Gamma) \vdash \\ \quad \textbf{validate as element}\ ElementName\ \textbf{of type}\ BaseTypeName\ \{ \\ \quad\quad \textbf{element}\ ElementName\ \{\ T_D(\Gamma)(UntypedValue')\} \\ \quad \}\ \Rightarrow\ Value^{IV} \end{array}$$

for some value $Value^{IV}$.

By inspection of the inference rules of the **resolves to** *judgement, from 2 we conclude that either*

$$(\textbf{define type}\ BaseTypeName\ \textbf{restricts}\ BaseTypeName'\ \{\ Type'\ \}) \in \Gamma$$

or

$$\begin{array}{l} (\textbf{define type}\ BaseTypeName\ \textbf{extends}\ BaseTypeName'\ \{\ Type''\ \}) \in \Gamma \\ \Gamma \vdash BaseTypeName'\ \textbf{resolves to}\ Type_0 \\ Type' = Type_0; Type'' \end{array}$$

In both cases, by induction over the length of the derivation 2 and by definition of T_S, we can easily prove that

$$T_S(\Gamma) \vdash \mathit{BaseTypeName}\ \textbf{resolves to}\ \pi_1(T_T(\mathit{Type}'))$$

Hence, by applying the appropriate inference rule of the **validate** *judgement, the thesis is reduced to*

$$T_S(\Gamma) \vdash \textbf{validate as}\ \pi_1(T_T(\mathit{Type}'))\ \{\ T_D(\Gamma)(\mathit{UntypedValue}')\ \}\ \Rightarrow\ \mathit{Value}^V$$

for some value Value^V.

The thesis follows directly from 3 and corollary 1 choosing Value''' for Value^V.

- *Case* $\boldsymbol{C}_2$.
  By hypothesis we know that

$$\begin{array}{l}\textbf{validate as}\ \mathit{ElementType}\ \{\ \textbf{element}\ \mathit{ElementName}\ \{\ \mathit{UntypedValue}'\ \}\ \}\ \Rightarrow\\ \quad\textbf{element}\ \mathit{ElementName}\ \textbf{of type}\ \mathit{BaseTypeName}\ \{\ \mathit{TypedValue}\ \}\end{array}$$

By definition of T_D we have

$$\begin{array}{l}T_D(\Gamma)(\textbf{element}\ \mathit{ElementName}\ \{\ \mathit{UntypedValue}'\ \}) =\\ \quad L^\Gamma(\Gamma, \mathit{ElementName}, \mathit{UntypedValue}')\end{array}$$

By inspection of the definition of L^Γ, either

$$\begin{array}{l}(\textbf{define element}\ \mathit{ElementName}\ \mathit{OfGuardedTypes}') \in \Gamma\\ L^\Gamma(\Gamma, \mathit{ElementName}, \mathit{UntypedValue}') =\\ \quad G^\Gamma(\mathit{OfGuardedTypes}', \mathit{ElementName}, \mathit{UntypedValue}')\end{array}$$

or

$$\begin{array}{l}L^\Gamma(\Gamma, \mathit{ElementName}, \mathit{UntypedValue}') =\\ \quad L^\Gamma_V(\mathit{Type}_x, \mathit{ElementName}, \mathit{UntypedValue}') =\\ \quad L^\Gamma_V(\textbf{element}\ \mathit{ElementName}\ \mathit{OfGuardedTypes}') =\\ \quad G^\Gamma(\mathit{OfGuardedTypes}', \mathit{ElementName}, \mathit{UntypedValue}')\end{array}$$

for some type Type_x and some list of guarded types $\mathit{OfGuardedTypes}'$.

Moreover $\mathit{TypeName} =$ **element** *$\mathit{ElementName}\ \mathit{OfGuardedTypes}$.*

Since $\Gamma;$ **define type** $\diamond\ \{\ \mathit{ElementType}\ \}$ *— that is equivalent to*

$$\Gamma; \textbf{define type}\ \diamond\ \{\textbf{element}\ \mathit{ElementName}\ \mathit{OfGuardedTypes}\ \}$$

— is without conflicts by hypothesis, we know that the guards of $\mathit{OfGuardedTypes}$ and those of $\mathit{OfGuardedTypes}'$ must be the same.

By inspection of the definition of G^Γ, we know that G^Γ only inspects the guards of its first argument and not the associated types.

Thus, in both cases, we conclude that:

$$\begin{array}{l}L^\Gamma(\Gamma, \mathit{ElementName}, \mathit{UntypedValue}') =\\ \quad G^\Gamma(\mathit{OfGuardedTypes}, \mathit{ElementName}, \mathit{UntypedValue}')\end{array}$$

Since $\textbf{xpath} - \textbf{eval}(E_i, \mathit{UntypedValue}') =$ `false()` $\forall i \in \{1, \ldots, j-1\}$
and $\textbf{xpath} - \textbf{eval}(E_j, \mathit{UntypedValue}') =$ `true()` *then by definition of G^Γ*

$$\begin{array}{l}G^\Gamma(\mathit{OfGuardedTypes}, \mathit{ElementName}, \mathit{UntypedValue}') =\\ \quad \textbf{element}\ < \mathit{ElementName}, \diamond >\ \{\\ \qquad \textbf{element}\ < \mathit{ElementName}, E_j >\ \{\\ \qquad\quad \textbf{element}\ \mathit{ElementName}\ \{\ T_D(\Gamma)(\mathit{UntypedValue}')\}\}\}\end{array}$$

By definition of T_T we have

$$\begin{array}{l} T_T(ElementType) = \\ \quad T_T(\textbf{element}\ ElementName\ OfGuardedTypes) = \\ \quad \textbf{element}\ < ElementName, \diamond >\ \textbf{of type}\ Typename, \\ \quad \{\textbf{define type}\ TypeName''\ \{\ Type''\ \}\}\ \cup\ \Gamma' \end{array}$$

where $TypeName''$ is a fresh type name and

$$(Type'', \Gamma') = T_G(ElementName, OfGuardedTypes) \tag{8}$$

Thus, to prove the thesis, we have to show that

$$\begin{array}{l} T_S(\Gamma) \cup \pi_2(T_T(ElementType)) \vdash \\ \quad \textbf{validate as element}\ < ElementName, \diamond >\ \textbf{of type}\ Typename\ \{ \\ \quad\quad \textbf{element}\ < ElementName, \diamond >\ \{ \\ \quad\quad\quad \textbf{element}\ < ElementName, E_j >\ \{ \\ \quad\quad\quad\quad \textbf{element}\ ElementName\ \{\ T_D(\Gamma)(UntypedValue')\}\}\} \\ \quad \}\ \Rightarrow\ Value' \end{array}$$

for some value $Value'$.

We observe that

$$\begin{array}{l} \vdash \\ \quad \textbf{element}\ < ElementName, \diamond >\ \textbf{of type}\ TypeName'' \\ \quad \textbf{yelds}\ < ElementName, \diamond >\ \textbf{of type}\ TypeName''\ \textbf{for} \\ \quad\quad \textbf{element}\ < ElementName, \diamond >\ \{ \\ \quad\quad\quad \textbf{element}\ < ElementName, E_j >\ \{ \\ \quad\quad\quad\quad \textbf{element}\ ElementName\ \{\ T_D(\Gamma)(UntypedValue')\}\}\} \end{array} \tag{9}$$

and that in $\pi_2(T_T(ElementType))$

$$TypeName''\ \textbf{resolves to}\ Type'' \tag{10}$$

Thus, to prove the thesis, we can just pick the appropriate inference rule of the **validate as** *judgement and, using 9 and 10, reduce the thesis to proving*

$$\begin{array}{l} T_S(\Gamma) \cup \pi_2(T_T(ElementType)) \vdash \\ \quad \textbf{validate as}\ Type''\ \{ \\ \quad\quad \textbf{element}\ < ElementName, E_j >\ \{ \\ \quad\quad\quad \textbf{element}\ ElementName\ \{\ T_D(\Gamma)(UntypedValue')\}\} \\ \quad \}\ \Rightarrow\ Value''' \end{array}$$

for some value $Value'''$.

By 8 and by definition of T_G we have

$$\begin{array}{l} (Type'', \Gamma') = \\ \quad T_G(ElementName, OfGuardedTypes) = \\ \quad \textbf{element}\ < ElementName, E_1 >\ \textbf{of type}\ TypeName'_1\ | \\ \quad \ldots\ | \\ \quad \textbf{element}\ < ElementName, E_n >\ \textbf{of type}\ TypeName'_n, \\ \quad \{\textbf{define type}\ TypeName'_1\ \{\ \textbf{element}\ ElementName\ OfType_1\}, \\ \quad\ \ldots, \\ \quad\ \textbf{define type}\ TypeName'_n\ \{\ \textbf{element}\ ElementName\ OfType_n\}\} \end{array} \tag{11}$$

In particular $TypeName'_j$ **resolves to element** $ElementName\ OfType_j$ *that by hypothesis is equivalent to*

$$TypeName'_j \textbf{ resolves to element } ElementName \textbf{ of type } BaseTypeName$$

From 11, by applying the appropriate inference rules for **validate***, we can reduce the thesis to*

$$\begin{array}{l} T_S(\Gamma) \cup \pi_2(T_T(ElementType)) \vdash \\ \quad \textbf{validate as element } ElementName \textbf{ of type } BaseTypeName\ \{ \\ \quad\ \ \textbf{element } ElementName\ \{\ T_D(\Gamma)(UntypedValue')\} \\ \quad \}\ \Rightarrow\ Value^{IV} \end{array}$$

for some value $Value^{IV}$*.*

By inspection of the inference rules of the **resolves to** *judgement, from 2 we conclude that either*

$$(\textbf{define type } BaseTypeName \textbf{ restricts } BaseTypeName'\ \{\ Type'\ \}) \in \Gamma$$

or

$$\begin{array}{l} (\textbf{define type } BaseTypeName \textbf{ extends } BaseTypeName'\ \{\ Type''\ \}) \in \Gamma \\ \Gamma \vdash BaseTypeName' \textbf{ resolves to } Type_0 \\ Type' = Type_0; Type'' \end{array}$$

In both cases, by induction over the length of the derivation 2 and by definition of T_S*, we can easily prove that*

$$T_S(\Gamma) \vdash BaseTypeName \textbf{ resolves to } \pi_1(T_T(Type'))$$

Hence, by applying the appropriate inference rule of the **validate** *judgement, the thesis is reduced to*

$$T_S(\Gamma) \cup \pi_2(T_T(ElementType)) \vdash \textbf{validate as } \pi_1(T_T(Type'))\ \{\ T_D(\Gamma)(UntypedValue')\ \}\ \Rightarrow\ Value^V$$

for some value $Value^V$*.*

The thesis follows directly from 3 and corollary 1 choosing $Value'''$ *for* $Value^V$*.*

$\Box$

Chapter 6

Solutions of the proposed examples

In this appendix we show solutions of the various languages we discussed about in 2 to the examples of co-constraints proposed in 4.

A No nesting of <a> elements in XHTML

A.1 The SGML DTD solution

```
<!ENTITY % Inline "(#PCDATA | %inline; | %misc.inline;)*">
<!ELEMENT a - - (%Inline) -(a) >
```

A.2 The xlinkit solution

```
<consistencyrule id="r">
 <forall var="x" in="//x:a">
  <not>
   <exists var="y" in="$x//x:a"/>
  </not>
 </forall>
</consistencyrule>
```

A.3 The Schematron solution

```
<rule context="x:a">
 <report test=".//x:a"
 >a must not contain other
  a elements</report>
</rule>
```

A.4 The DSD solution

```
<ElementDef ID="a">
 <Not>
  <Context>
   <Element Name="a"/>
   <SomeElements/>
   <Element Name="a"/>
  </Context>
 </Not>
 <Content IDRef="Inline"/>
</ElementDef>
```

A.5 The SchemaPath solution

```
<xsd:element name="a">
 <xsd:annotation>
  <xsd:documentation>
   content is "Inline" except that anchors shouldn’t be nested
  </xsd:documentation>
 </xsd:annotation>
 <xsd:alt cond=".//x:a" type="xsd:error"/>
 <xsd:alt               type="x:a.type" priority="0"/>
</xsd:element>
```

B Variables in XSLT

B.1 The xlinkit solution

```
<consistencyrule id="r">
 <forall var="x" in="//xsl:variable">
  <implies>
   <exists var="y" in="$x/@select"/>
   <and>
    <not>
     <exists var="y" in="$x/*"/>
    </not>
    <notequal op1="$x/text()" op2="’’"/>
   </and>
  </implies>
 </forall>
</consistencyrule>
```

B.2 The Schematron solution

```
<rule context="xsl:variable">
 <report test="@select and (* or text()!=’’)"
 >@select implies an empty content</report>
</rule>
```

B.3 The DSD solution

```
<ElementDef ID="variable">
 <AttributeDecl Name="select" Optional="yes">
  <StringType IDRef="expr"/>
 </AttributeDecl>
 <If>
  <Attribute Name="select"/>
  <Then>
   <Empty/>
  </Then>
 </If>
 <Content IDRef="templateContent"/>
</ElementDef>
```

B.4 The RELAX NG solution

```
<element name="variable">
 <choice>
  <attribute name="select">
```

```
   <ref name="expr"/>
  </attribute>
  <ref name="templateContent"/>
 </choice>
</element>
```

B.5 The SchemaPath solution

```
<xsd:element name="variable">
 <xsd:alt cond="@select and (child::* or text()!='')"
          type="xsd:error"/>
 <xsd:alt type="xsl:variableType" priority="0"/>
</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>
```

C Named templates

C.1 The xlinkit solution

```
<consistencyrule id="r">
 <forall var="x" in="//xsl:template">
  <implies>
   <not>
    <exists var="y" in="$x/@name"/>
   </not>
   <exists var="z" in="$x/@match"/>
  </implies>
 </forall>
</consistencyrule>
```

C.2 The Schematron solution

```
<rule context="xsl:template">
 <report test="not(@name) and not(@match)"
 >Error</report>
</rule>
```

C.3 The DSD solution

```
<ElementDef ID="template">
 <AttributeDecl Name="match" Optional="yes">
  <StringType IDRef="Pattern"/>
 </AttributeDecl>
 <AttributeDecl Name="name" Optional="yes">
  <StringType IDRef="NCName"/>
 </AttributeDecl>
 <If>
  <Not>
   <Attribute Name="name"/>
  </Not>
  <Then>
   <Attribute Name="match"/>
  </Then>
 </If>
 <Content IDRef="templateContent"/>
</ElementDef>
```

C.4 The RELAX NG solution

```
<element name="template">
 <choice>
  <group>
   <attribute name="name">
    <data type="NCName"/>
   </attribute>
   <attribute name="match">
    <ref name="Pattern"/>
   </attribute>
  </group>
  <attribute name="name">
   <data type="NCName"/>
  </attribute>
  <attribute name="match">
   <ref name="Pattern"/>
  </attribute>
 </choice>
 <ref name="templateContent"/>
</element>
```

C.5 The SchemaPath solution

```
<xsd:element name="template">
 <xsd:alt cond="not(@match) and not(@name)" type="xsd:error" />
 <xsd:alt                                   type="xsl:templateType"
          priority="0"/>
</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>
```

D Elements in XML Schema

D.1 The xlinkit solution

```
<consistencyrule id="local-ref">
 <forall var="x" in="//xsd:element[@ref]">
  <and>
   <not>
    <exists var="y" in="$x/parent::xsd:schema"/>
   </not>
   <and>
```

```
    <not>
     <exists var="y" in="$x/@name or $x/@type"/>
    </not>
    <not>
     <exists var="y" in="$x/xsd:simpleType or $x/xsd:complexType"/>
    </not>
   </and>
  </and>
 </forall>
</consistencyrule>

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

D.2 The Schematron solution

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

D.3 The DSD solution

```
<ContentDef ID="elementContent">
 <Optional>
  <Union>
   <Element IDRef="simpleType"/>
   <Element IDRef="complexType"/>
  </Union>
 </Optional>
</ContentDef>
```

```
<ElementDef ID="globalElement" Name="element">
 <AttributeDecl Name="name">
  <StringType IDRef="NCName"/>
 </AttributeDecl>
 <AttributeDecl Name="type" Optional="yes">
  <StringType IDRef="QName"/>
 </AttributeDecl>
 <If>
  <Attribute Name="type"/>
  <Then><Empty/></Then>
 </If>
 <Content IDRef="elementContent"/>
</ElementDef>

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

D.4 The RELAX NG solution

```
<define name="globalElement">
 <element name="element">
  <attribute name="name">
   <data type="NCName"/>
  </attribute>
  <ref name="namedOrAnon"/>
 </element>
</define>
```

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

<define name="namedOrAnon">
 <choice>
  <attribute name="type">
   <data type="QName"/>
  </attribute>
  <optional>
   <choice>
    <ref name="complexType"/>
    <ref name="simpleType"/>
   </choice>
  </optional>
 </choice>
</define>
```

D.5 The SchemaPath solution

```
<xsd:complexType name="element">
 <xsd:sequence>
  <xsd:choice minOccurs="0">
   <xsd:element name="simpleType" type="xsd:localSimpleType"/>
   <xsd:element name="complexType" type="xsd:localComplexType"/>
  </xsd:choice>
 </xsd:sequence>
 <xsd:attribute name="name" type="xsd:NCName"/>
 <xsd:attribute name="ref" type="xsd:QName"/>
 <xsd:attribute name="type" type="xsd:QName"/>
</xsd:complexType>

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

E FpML validation rules

E.1 The xlinkit solution

```
<consistencyrule id="r3">
 <forall var="x" in="//calculationPeriodDatesAdjustments
                   |//DateAdjustments
                   |//paymentDatesAdjustments
                   |//resetDatesAdjustments">
  <iff>
   <equal op1="$x/businessDayConvention/text()" op2="'NONE'"/>
   <and>
    <not>
     <exists var="y" in="$x/businessCentersReference"/>
    </not>
    <not>
     <exists var="y" in="$x/businessCenters"/>
    </not>
   </and>
  </iff>
 </forall>
</consistencyrule>
```

E.2 The Schematronsolution

```
<pattern name="fpml-r3">
  <rule context=
     "//calculationPeriodDatesAdjustments
        [businessDayConvention/text()='NONE']|
      //dateAdjustments
        [businessDayConvention/text()='NONE']|
      //paymentDatesAdjustments
        [businessDayConvention/text()='NONE']|
      //resetDatesAdjustments
        [businessDayConvention/text()='NONE']">
    <report test="businessCentersReference">Error</report>
    <report test="businessCenters">Error</report>
  </rule>
  <rule context=
     "//calculationPeriodDatesAdjustments
        [businessDayConvention/text()!='NONE']|
     //dateAdjustments
        [businessDayConvention/text()!='NONE']|
     //paymentDatesAdjustments
        [businessDayConvention/text()!='NONE']|
     //resetDatesAdjustments
        [businessDayConvention/text()!='NONE']">
    <assert test="businessCentersReference or
                  businessCenters">Error</assert>
  </rule>
</pattern>
```

E.3 The DSD solution

```
<ContentDef ID="globalContent">
 <Union>
  <Content IDRef="withCenters"/>
  <Content IDRef="withoutCenters"/>
 </Union>
</ContentDef>

<ContentDef ID="withCenters">
 <Sequence>
  <Element IDRef="businessDayConventionWith"/>
  <Union>
   <Sequence>
    <Element IDRef="businessCentersReference"/>
    <Element IDRef="businessCenters"/>
   </Sequence>
   <Element IDRef="businessCentersReference"/>
   <Element IDRef="businessCenters"/>
  </Union>
 </Sequence>
</ContentDef>

<ContentDef ID="withoutCenters">
 <Element IDRef="businessDayConvention-NONE"/>
</ContentDef>

<ElementDef ID="businessDayConvention-NONE"
            Name="businessDayConvention">
 <StringType>
  <String Value="NONE"/>
 </StringType>
</ElementDef>

<ElementDef ID="businessDayConventionWith"
            Name="businessDayConvention">
 <StringType>
  <Complement>
   <String Value="NONE"/>
  </Complement>
 </StringType>
</ElementDef>
```

E.4 The RELAX NG solution

```
<define name="globalContent">
 <choice>
  <ref name="withCenters"/>
  <ref name="withoutCenters"/>
 </choice>
</define>
```

```
<define name="withCenters">
 <element name="businessDayConvention">
  <data type="string">
   <except>
    <value>NONE</value>
   </except>
  </data>
 </element>
 <choice>
  <element name="businessCentersReference">
   <ref name="bcr-pattern"/>
  </element>
  <element name="businessCenters">
   <ref name="bc-pattern"/>
  </element>
  <group>
   <element name="businessCentersReference">
    <ref name="bcr-pattern"/>
   </element>
   <element name="businessCenters">
    <ref name="bc-pattern"/>
   </element>
  </group>
 </choice>
</define>

<define name="withoutCenters">
 <element name="businessDayConvention">
  <value>NONE</value>
 </element>
</define>
```

E.5 The SchemaPath solution

```
<xsd:element name="calculationPeriodDatesAdjustments">
 <xsd:alt cond="businessDayConvention/text()='NONE' and
          (not(businessCentersReference) and not(businessCenters))"
          type="BusinessDayAdjustmentsType"/>
 <xsd:alt cond="businessDayConvention/text()!='NONE' and
          (businessCentersReference or businessCenters)"
          type="BusinessDayAdjustmentsType"/>
 <xsd:alt type="xsd:error" priority="0"/>
</xsd:element>

<xsd:element name="dateAdjustments">
 <xsd:alt cond="businessDayConvention/text()='NONE' and
          (not(businessCentersReference) and not(businessCenters))"
          type="BusinessDayAdjustmentsType"/>
 <xsd:alt cond="businessDayConvention/text()!='NONE' and
          (businessCentersReference or businessCenters)"
          type="BusinessDayAdjustmentsType"/>
 <xsd:alt type="xsd:error" priority="0"/>
</xsd:element>

<xsd:element name="paymentDatesAdjustments">
```

```
 <xsd:alt cond="businessDayConvention/text()=’NONE’ and
          (not(businessCentersReference) and not(businessCenters))"
          type="BusinessDayAdjustmentsType"/>
 <xsd:alt cond="businessDayConvention/text()!=’NONE’ and
          (businessCentersReference or businessCenters)"
          type="BusinessDayAdjustmentsType"/>
 <xsd:alt type="xsd:error" priority="0"/>
</xsd:element>

<xsd:element name="resetDatesAdjustments">
 <xsd:alt cond="businessDayConvention/text()=’NONE’ and
          (not(businessCentersReference) and not(businessCenters))"
          type="BusinessDayAdjustmentsType"/>
 <xsd:alt cond="businessDayConvention/text()!=’NONE’ and
          (businessCentersReference or businessCenters)"
          type="BusinessDayAdjustmentsType"/>
 <xsd:alt type="xsd:error" priority="0"/>
</xsd:element>

<xsd:complexType name="BusinessDayAdjustmentsType">
 <xsd:sequence>
   <xsd:element name="businessDayConvention"    type="xsd:string"/>
   <xsd:element name="businessCentersReference" type="bcrType"
                minOccurs="0"/>
   <xsd:element name="businessCenters" type="bcType"
                minOccurs="0"/>
 </xsd:sequence>
</xsd:complexType>
```

References

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

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

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

[4] The Financial products Markup Language. http://www.fpml.org.

[5] P. V. Biron and A. Malhotra. *XML Schema Part 2: Datatypes*. Technical report, W3C, May 2001.

[6] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. *Extensible Markup Language (XML) 1.0 (Second Edition)*. Technical report, W3C, October 2000.

[7] J. Clark. *XSL Transformation (XSLT) Version 1.0*. Technical report, W3C, November 1999.

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

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

[10] S. DeRose, E. Maler, and D. Orchard. XML Linking Language (XLink) Version 1.0. Technical report, W3C, June 2001.

[11] S. P. et alt. *XHTML 1.0 The Extensible HyperText Markup Language (Second Edition)*. Technical report, W3C, January 2000.

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

[13] P. W. Jérôme Siméon. The Essence of XML. In *Proceedings of the 30th ACML SIGPLAN Symposium on Principles of Programming Languages*, New Orleans, January 2003.

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

[15] A. Malhotra and M. Maloney. *XML Schema Requirements*, February 1999.

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

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

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

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

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

[21] B. Thal. FpML Validation. Technical report, USB Warburg, University College of London and Systemwire, 2002.

[22] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema Part 1: Structures*. Technical report, W3C, May 2001.

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