ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für
Technische Informatik und
Kommunikationsnetze

Master Thesis

Representation of XML Schema Components

FELIX MICHEL

Department of Electrical Engineering and Information Technology

ETH Zürich

School of Information

University of California, Berkeley

Abstract

Among the different representations of XML Schema's data model which are available today, none provides access to the Schema components in a unified, extensible, and navigable way from current XML-based technologies like XSLT 2.0. This is a major disadvantage, because XML Schemas contain information which can be highly valuable for applications. We present an XML representation of XML Schema's data model and a path language for XML Schema, together with an abstract definition of the requirements of an *accessible data model*. An accessible data model leads to significantly more powerful, resilient, and adaptable applications; and it opens new fields of Schema-aware applications.

Acknowledgments

Contents

Chapter 1

Introduction

This thesis discusses the issues of representing *XML Schema components*, which are the abstract building blocks of an XML Schema. XML Schema information is represented and exposed in different formats and ways. The abstract data model in the recommendation, the commonly used transfer syntax, the PSVI augmentation of the Infoset, and various XML Schema APIs all represent XML Schema information, or parts thereof. However, we argue that all these approaches cannot solve many current problems, and that they all miss numerous promising opportunities. We specify the requirements for an *accessible data model*, we outline possible ways to achieve data model accessibility in practice, and we present an *XML syntax* for XML Schema components and a *path language* which works on XML Schema components. After evaluating different use cases, we conclude that both approaches are highly useful to XML applications, in particular in the context of *Web-based services*, *XML pipelines*, and *composite schemas*.

Chapter Overview

Chapter 2 studies XML Schema, its data model, its distinctive features, and related technologies. The chapter emphasizes in which ways XML Schema is different from other schema languages for XML (e.g., by analyzing the type-annotation aspect of Schema-validation), and it lists uses and applications of XML Schema. A survey of a few selected technologies that are related to XML Schema, and a critical comment on the recommendation of XML Schema conclude the chapter.

Chapter 3 presents application areas where a lack of data model accessibility causes problems, and it presents application areas which will benefit from an accessible data model, and where new opportunities will be created through a better representation of XML Schema information. In particular, Section 3.1 discusses problems of *versioning and extensibility* of XML vocabularies, and it summarizes different strategies and approaches to deal with these problems.

Chapter 4 includes an introductory overview of formal languages in general, with a special focus on XML grammars. Section 4.4 highlights a few interesting properties, which inspire the implementations described in Chapter 8.

Chapter 5 distinguishes two classes of Schema-processing applications: *stand-alone* application, which work primarily on the Schema itself, and *instance-driven* applications, which utilize Schema information in order to process instance documents.

In Chapter 6, we present the *Schema Components XML Syntax* (SCX), an XML format which aims at representing the Schema components as faithfully as possible. We explain the design rationale which has been employed while defining the format, and we demonstrate how SCX can be the base of *canonicalization* for XML Schema.

Chapter 7 introduces the *XML Schema Path Language* (SPath), an extension of XPath, which permits accessing and navigating XML Schema information. The design considerations are explained, and the syntax is outlined.

Chapter 8 describes the prototype implementation of SCX and an XSLT 2.0-based function library which substantially simplifies working with SCX. In addition, *X2Doc*, an extensible and configurable framework for generating XML Schema documentation using XSLT, is presented.

Chapter 9 evaluates the two technologies proposed in Chapter 6 and 7, alongside with the general concept of an accessible data model from Chapter 5, in the context of the use cases from Chapter 3.

The conclusions in Chapter 10 summarize the results of the evaluation, and possible next steps in the further development of representation of XML Schema information are indicated.

Remarks

Notation of Qualified Names

We use "qualified name" always in the sense of expanded qualified names, or, according to Clark,[1] "universal name". That is, a *qualified name* is a tupel consisting of a namespace URI and a local name.

Where no explicit XML namespace bindings are provided, the XML namespace bindings from Table 1.1 are implied.

Prefix	XML Namespace URI
xml	http://www.w3.org/XML/1998/namespace
xs	http://www.w3.org/2001/XMLSchema
xsi	http://www.w3.org/2001/XMLSchema-instance
xsl	http://www.w3.org/1999/XSL/Transform
rng	http://relaxng.org/ns/structure/1.0
xhtml	http://www.w3.org/1999/xhtml
wsdl	http://schemas.xmlsoap.org/wsdl/
ex	http://people.ee.ethz.ch/%7Efemichel/namespaces/example
scx	http://people.ee.ethz.ch/%7Efemichel/namespaces/schema-components
scf	http://people.ee.ethz.ch/%7Efemichel/namespaces/schema-component-functions
occ	http://people.ee.ethz.ch/%7Efemichel/namespaces/occurrence
val	http://people.ee.ethz.ch/%7Efemichel/namespaces/validation

Table 1.1: Implied XML namespace bindings

Code Samples

Sample code is always displayed in framed boxes. The excerpts have been chosen to be as compact as possible, while still being meaningful. Lines that start with `saxon@work>` indicate sample output. The code examples all have been actually executed, and they have been executed only using the technologies indicated. We used the Saxon-SA 8.8 XSLT processor from Saxonica[2] for all examples.

---

[1]See Clark's article on XML namespaces: http://www.jclark.com/xml/xmlns.htm
[2]http://saxonica.com/

Chapter 2

XML Schema

XML Schema [10, 91] is the XML schema language recommended by the *World Wide Web Consortium* (W3C).[1] In essence, XML schema languages define constraints which are used for describing a class of XML documents. An important class of schema languages are *document grammars*, and XML Schema is an example thereof. XML Schema has been developed as a successor to DTDs, a subset of SGML document grammars and part of the initial recommendation of XML [18], and evolved from different competing proposals (most notably SOX [35], DCD [15], and XDR [43]).

The main characteristics which distinguish XML Schema from DTDs are:

1. Support of XML Namespaces [17]
2. Presence of an XML syntax
3. Introduction of *type derivation*, akin to inheritance in object-oriented programming
4. Support for definition of *faceted simple types*, and definition of a set of *built-in types*
5. Generalization of the concept of *Identity Constraints*
6. Re-introduction of the *all group*, a very restricted subset of SGML's interleave operator, which had been omitted in DTDs

The above features make XML Schema more powerful and expressive, but at the same time harder to deal with than DTDs. It is debatable to which extent schema languages should comprise features for data modeling such as type inheritance. In fact, today's competitors of XML Schema like DSD [70] and RELAX NG [31] (which we briefly examine in Section 2.4.2) chose to limit themselves to being pure document grammars.

Although not part of the XML recommendation, XML Schema has become a core part in many of the W3C's XML-related recommendations. The most recent generation of

---

[1]The choice of "XML Schema" as a name for a XML schema language may lead to confusion. In this report, we shall strictly use "schema" with lower-case "s" where we intend to denote the general class of schema languages, and "Schema" with upper-case "S" when referring to the W3C's specific schema language.

XML technologies, i.e., XPath 2.0 [5], XSLT 2.0 [54], and XQuery 1.0 [11], all build upon XML Schema. At present, Version 1.1 of XML Schema is in preparation [82, 93]. As of February 2007, it has the status of a Working Draft, and despite being announced as a minor version, it will introduce new language constructs and relax some of the current restrictions of XML Schema. Many of these changes are addressing problems that arise from the need for schema extensibility and schema versioning, which we discuss in Section 3.1.

2.1 Definition and Format

In Section 2 of the first part of the recommendation [91], the conceptual framework of XML Schema is described. It defines XML Schema "in terms of an abstract data model" which is composed of *Schema components*, the latter being defined as "the building blocks that comprise the abstract data model of the schema". The recommendation then states:

> The abstract model for schemas is conceptual only, and does not mandate any particular implementation or representation of this information. To facilitate interoperation and sharing of schema information, a normative XML interchange format for schemas is provided.

Documents written in this normative XML format are what is commonly known as XML Schemas. In this report however, we shall carefully distinguish the abstract data model from the XML syntax, for the following reasons:

1. Components of one Schema are likely to be scattered over different documents in the XML syntax, while the abstract data model always includes all components.

2. The elements of the XML syntax are considerably different from the Schema components. They introduce new properties (e.g., the `id` attribute), while delegating others to document-wide settings (e.g., to the `targetNamespace` attribute).

3. The abstract data model contains more information than what is contained in the set of relevant documents in the XML syntax (e.g., the built-in types, and default values from the recommendation).

Altogether, the differences between the abstract data model and the XML syntax are significant. Section 6 demonstrates how difficult it can be in practice to retrieve the Schema components from their XML representation.

In the following, "XML Schema" always refers to the abstract *Schema as a whole*, while representations thereof, defined in the normative XML syntax, are referred to as "XML Schema documents". Finally, the normative XML syntax is simply called the "transfer syntax", in accordance to common practice in the W3C.

As a typographical convention, properties of Schema components are always surrounded by curly braces, while properties of the PSVI (see below) are typeset in fixed-width font and surrounded by brackets.

2.2 Unique Features

We assume the reader to be familiar with XML Schema and its XML syntax. Therefore, we only very briefly list the distinctive features of XML Schema and refer to the *XML Schema Primer* [40] for further explanations.

XML Namespaces: XML Schema supports XML namespaces. Schemas usually define a *target namespace*, and the components defined by a Schema then have to be used by their qualified names. Given a target namespace which is mapped to the prefix `ex`, global components thus have to be referenced as follows:

```
<xs:element name="paragraph" type="xs:string"/>
<xs:element ref="ex:paragraph" minOccurs="0"/>
```

Imports and Includes: XML Schema permits *including* of XML Schema documents with the same target namespace, and *importing* of components from XML Schemas with different target namespaces. The `xs:import` element may or may not specify a `schemaLocation`.

```
<xs:include schemaLocation="address.xsd"/>
<xs:import namespace="http://www.w3.org/1999/xhtml"/>
```

Complex and Simple Types: Complex types describe the content model of elements, i.e., which child elements and attributes are permitted. Simple types restrict the literal content of elements.

```
<xs:complexType name="NameType">
  <xs:sequence>
    <xs:element name="First" type="xs:string"/>
    <xs:element name="Last" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Type Derivation: Both complex and simple types can be used for type derivation. Complex types can be derived by *restriction* or *extension*; simple types can only be restricted. Restriction narrows the set of permissible values, extension appends additional elements to the end of the model group.

In addition, simple types can be *constructed* by *list* or *union*. Union types and list types can then be further restricted by derivation.

```
<xs:complexType name="PersonType">
  <xs:complexContent>
    <xs:extension base="ex:NameType">
      <xs:sequence>
        <xs:element name="Age" type="ex:AgeType"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs;simpleType name="AgeType">
  <xs:union memberTypes="xs:positiveInteger">
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="unknown"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>
```

Wildcards: XML Schema provides wildcards, for both elements and attributes. A wildcard matches any element or attribute in an instance, but the set of matching elements and attributes can be restricted to a set of namespaces. Through the attribute `processContents`, the validation processor can be advised to perform *strict* or *lax* validation, or to *skip* validation completely for the matching items.

```
<xs:complexType name="anyType" mixed="true">
  <xs:sequence>
    <xs:any minOccurs="0" maxOccurs="unbounded"
            processContents="lax"/>
  </xs:sequence>
  <xs:anyAttribute processContents="lax"/>
</xs:complexType>
```

Type Substitution: Types can be substituted by derived types in instances. Substitutions always must be indicated by the `xsi:type` attribute. Type substitution can be controlled in the Schema through the attribute `block`.

Substitution Groups: An element declaration can be member of a *substitution group*. To form a substitution group, element declarations reference a certain element as their substitution group head. The types of elements within a substitution group must be derived from the type of the group head.

In an instance, occurrences of the head element are allowed to be replaced by elements from the substitution group.

Identity Constraints: XML Schema generalizes the concept of ID/IDREF known from DTDs. If offers three kinds of constraints: a *unique* constraint, a *key* constraint, and a *key reference* constraint. All constraints are defined within element

declarations, and they are defined to act on declarations, i.e., on elements or attributes. A *selector* defines the set of nodes among which the constraint should be asserted, and one or more *fields* determine the constrained values of those nodes.

```
<xs:unique>
  <xs:selector xpath=".//ex:person"/>
  <xs:field xpath="ex:First"/>
  <xs:field xpath="ex:Last"/>
</xs:unique>
```

All Groups: XML Schema re-introduces the *interleave operator*, which is part of SGML content models, but has been omitted in DTDs. However, its use is very restricted: *All groups* must be the only model group (i.e., they must not be part of hierarchically nested model groups), and particles in *all groups* may occur at most once.

Element Declaration Consistent Rule: The *element declaration consistent rule* (EDC) prohibits elements with the same name to have different types, within a given model group.

Unique Particle Attribution: The *unique particle attribution* (UPA) is a leftover from the DTD specification, which requires content models to be *deterministic*.[2] This means that it must be possible to determine the matching particle of an instance node without looking ahead in the instance. The following Schema violates UPA, because given an instance `<a /><a />`, it is not decidable for the first node whether it matches the first or second element declaration in the Schema.

```
<xs:sequence>
  <xs:element name="a" minOccurs="0"/>
  <xs:element name="a"/>
</xs:sequence>
```

Often, but not always, content models which violate UPA can be rewritten in a compliant way. The example above, for instance, is easily rewritable:

```
<xs:sequence>
  <xs:element name="a" maxOccurs="2"/>
</xs:sequence>
```

2.3   Use and Applications

The name of XML Schema's predecessor reflects the primary use of a schema language very clearly: DTD [18] stands for *Document Type Definition*. A precise definition of which documents are legal members of a certain class of documents is helpful both for producers and for consumers of documents. It helps the former to produce conforming

---

[2]Unfortunately, a more detailed definition of *deterministic content models* can only be found in a non-normative part of the XML recommendation. Furthermore, one of the main reasons to require deterministic content models is compatibility with SGML; however, SGML calls them *unambiguous*.

documents, and it lets the latter *validate* documents; i.e., it lets one decide whether a given document conforms to the schema. This is especially important in the distributed or *loosely coupled* scenarios where XML is typically used, since it is utilized as a transfer format very often.

Furthermore, an *a priori* specification of a class of documents allows to make assumptions about the documents being processed, and hence, it simplifies and improves the development of software applications processing these documents. The type concept introduced in XML Schema intends to emphasize these aspects by increasing the possibilities of code reuse and by providing for typed processing.

This section discusses the areas of use that are deemed to be the most important in the context of this report. It shows that XML Schema has its applications beyond simply being a document grammar. Knowing the predominant areas of use, the problems described in Section 3 become evident, and so does the potential of our contribution.

2.3.1 Document Validation

DTDs perform document validation on a mere pass/fail basis. If a document instance fails to pass document validation against a given DTD, an application cannot retrieve any more information about the incriminated document in a standardized way, and in the general case, it has to discard it. XML Schema enhances the concept of document validation in multiple ways. First, the term "validation" is misleading in the case of XML Schema, because *Schema-validation* actually comprises two functionally different parts: Document validation, and type annotation.

Schema-validation yields more informative and fine-grained results. It adds these results as an augmentation to the *XML Information Set* (Infoset) [34], known as the *Post Schema-Validation Information* (PSVI).[3] Unfortunately enough, the Infoset does not specify a unified way of how such augmentations should be appended and exposed, and the PSVI's contents are only laid out in the XML Schema recommendation [91]. Together, this makes the format and contents of the PSVI processor-dependant, a fact we address in more detail in Section 3.

Schema-validation consists of different steps: Assembling of the Schema from different Schema documents, local validation of element and attribute information items of an input Infoset, computation of the overall Schema-validity, and augmentation of this Infoset. The recommendation uses the term *Schema assessment* for the overall process.

In contrast to the pass/fail paradigm of DTD-validation, Schema-validation outcomes are more nuanced. In fact, the validation outcome has two dimensions: `[Validation attempted]`, which assumes values in (full, partial, none), and `[Validity]`, which is one

---

[3]XML Schema works on the Infoset of XML documents — both on the input and on the output side. This implies *well-formedness*, correct handling of namespaces (i.e., *namespace validity*), and the canonicalizations described in [34].

of (valid, invalid, notKnown).[4] Additionally, `[Validation context]` points to the nearest globally declared Schema component, and `[Schema specified]` indicates whether the value of the information item validated has been set through a default value in the Schema.

The fact that there are different levels of `[Validation attempted]` is due to the possibility to control the *processing mode* of Schema-validation. This can be done by setting the initial processing mode of a Schema processor, or it can be set by means of the `processContents` attribute on wildcard components. The possible values are:

1. `strict`: The Schema must contain declarations for all information items.
2. `lax`: If a declaration can be found, validity assessment is applied.
3. `skip`: No validation is carried out, even if declarations are present.

Recalling the twofold service of Schema-validation, partial validation is especially useful: The second goal of Schema-validation is *type annotation*, and the type annotations appended to the Infoset may be of importance to an application even if `[Validation attempted]` did not evaluate to "full" for every information item.

Type annotation is a very substantial addition to the universe of XML, because it turns XML instance nodes into typed nodes, and because it changes the role of the schema language. XML Schema moves from being a simple gauge, which measures, but does not affect, instance documents under validation, to a format containing external information which is partially added to the instances during validation, possibly changing the content of the instance and its interpretation.[5] The most recent additions to the family of XML technologies, i.e., XPath 2.0, XSLT 2.0, and XQuery 1.0, are the first generation of technologies which work on the Infoset plus the PSVI. Many of their principal new abilities and improvements rely on type annotations. Section 2.4.1 exemplifies this in the context of XSLT 2.0.

Besides serving as an input filter for exchange documents in the aforementioned scenarios, document validation has also proven to be highly useful in internal applications, and for application development. This field of application can be considered to become more and more important. The recommendations of both XSLT 2.0 and XQuery 1.0 describe the possibility of *Schema-aware* processors. They include capabilities such as static type-checking and they essentially facilitate better development and debugging of applications. The benefits of document validation between internal processing steps increase together with the complexity of the XML vocabularies involved. Developers thus highly recommend the use of Schema-aware processing [53].

Recently, the W3C released a first Working Draft for *XProc* [94], which aims at defining a standard language for *XML pipelines*. XML pipelines are expected to be an important

---

[4]An overview of the legal combinations can be found in tabular form on `http://www.w3.org/XML/2001/06/validity-outcomes.html`

[5]Actually, DTDs also augment the Infoset of the documents under validation (e.g., with default values). RELAX NG, in contrast, strictly leaves the instance untouched.

paradigm in the near future. Here, document validation will clearly contribute both to development and to operation.

2.3.2 Type System

An essential — and the distinctively unique — feature of XML Schema is its type system. In XML Schema, the notion of *types* is employed in at least three slightly different senses, which may cause some confusion. Let us recall the general definition of types in computer science: They define a set of permissible values (in terms of their *lexical space* and their *value space*), and they define which operations are possible on a value of that type. The categories of types we might encounter in the context of XML Schema are:

1. Document types
2. Complex types
3. Simple types

The third category, simple types, clearly fits very well in the above classical definition: Simple types define a lexical space, a value space, and a set of legal operations. XPath 2.0 works on values of such types in the usual way typed programming languages work; it can type-check values, and it can signal type errors (e.g., if a numeric operation is attempted to be applied to a string value).

At first sight, the first two categories fall into a more general notion of types — i.e., generalization, or abstraction, of a set, or extension, of instances — because the aspect of permitted operations seems less relevant. As described in Section 4.2.2, complex types can be merely seen as a representation of production rules in context-free grammars without further importance during the processing of instances. Schema-aware versions of XSLT 2.0 processors, however, allow for working with complex types in the classical manner. Document types also contain all three aspects of types in the narrower sense. In a more coarse-grained perspective, rejection of invalid document instances by document-processing applications is equivalent to the signaling of type errors in traditional programming languages. The lexical space of complex and document types is their XML serialization, whereas the Infoset represents the respective value space.

Thus, all three categories above are to be considered types in the classical sense of computer science, while still being disjunct subcategories that are not to be confused.

The type system of XML Schema has its applications and use outside XML Schema as well because it is used by third-party technologies. We already mentioned that the type system of XSLT 2.0 and XQuery 1.0 is based on the type system of XML Schema. RELAX NG, incidentally the most prominent competitor of XML Schema as a schema language for XML, does not specify a type system of its own. Instead, it provides the possibility of importing an external type system. Usually, the simple types of XML Schema are imported.

A special example of external use of XML Schema's type system is *XJ* [81], the *XML Enhancements for Java.* It is a framework which extends Java to integrate XML Schema types as first-class constructs. It enables the developer to "import XML schemas just as one does Java classes." It then lets one use XML elements, Schema types, and language constructs (i.e., XPath expressions) within Java. It requires a special compiler and execution environment and it is a promising approach to the rapid development of robust and maintainable XML applications.

2.3.3   Structural Metadata

From the above it becomes evident that XML Schema is more than a language for pure document grammars. It also comprises features for data modeling. The foremost examples of such features are *type derivation* and *identity constraints.* The information expressed through these constructs usually cannot be retrieved from the XML instance alone. In order to interpret the instance, the XML Schema has to be considered as well.[6] The following two brief examples demonstrate how XML Schema can be employed for modeling structures which could not be expressed using XML alone.

Type Derivation: If used properly, type derivation reflects a semantic relationship between the types involved. Imagine a base type that models a *person* and the properties needed (like name, address, et cetera). Now a great number of types might be derived thereof, e.g., types for describing specialized kinds of persons: *author*, *employee*, *professional kite surfer*, and so on. Although these derived types may greatly vary in their structure, the knowledge of their relationship can be valuable both to the interpretation of instance data and for application development.

Modeling General Graphs: XML is inherently limited to represent trees. However, the data to be represented often has the structure of a more general graph. A directed graph $G = (V, A)$ is defined as a pair of a set of vertices V and a set of arcs A. Figure 2.1 presents the — perhaps naive, but most general — definition of an XML representation of graphs.

The sample instance in Figure 2.2 uses the simplest non-tree graph (a circular graph containing two mutually dependent vertices) in order to illustrate that XML Schemas can contain information that is essential for the interpretation of XML instances. A conceptually simple structure becomes hard to read and cumbersome to be retrieved when represented as XML. The expressiveness of XML Schema is an advantage in that it permits capturing such structures. This in turn limits the extent to which XML can be considered an external data format.[7]

---

[6]And even then it might be less than trivial to retrieve the model properties. This is mostly due to the transfer syntax of XML Schema. See Sections 3.3 and 9.2 for a discussion.

[7]Siméon and Wadler [88] claim that XML does not fulfill the conditions of an external data format

```
<xs:element name="vertex">
 <xs:complexType>
  <xs:sequence><xs:element name="name" type="xs:string"/></xs:sequence>
  <xs:attribute name="vID" type="xs:ID"/>
 </xs:complexType>
</xs:element>

<xs:element name="arc">
 <xs:complexType>
  <xs:sequence><xs:element name="role" type="xs:token"/></xs:sequence>
  <xs:attribute name="tail" type="xs:IDREF"/>
  <xs:attribute name="head" type="xs:IDREF"/>
 </xs:complexType>
</xs:element>
```

Figure 2.1: An XML Schema describing a directed graph

```
<vertex vID="A"><name>Alice</name></vertex>
<vertex vID="B"><name>Bob</name></vertex>

<arc tail="A" head="B"><role>marriedTo</role></arc>
<arc tail="B" head="A"><role>marriedTo</role></arc>
```

Figure 2.2: An XML representation of a directed graph

So far we have only investigated the ways in which XML Schema may contain structural metadata itself. We have seen that XML Schema indeed offers a set of modeling facilities. However, there are other languages and technologies which are explicitly targeting structural or semantic modeling, and which are thus better suited for describing structural metadata. The most prominent examples emerged from the context of ontological frameworks and the semantic Web, for example the *Resource Description Framework* (RDF) [55] and the *Web Ontology Language* (OWL) [4]. The discussion of those languages, however, is beyond the scope of this report. Nevertheless, it is important to note that there are means to connect XML Schemas to external metadata, and that this is done in practice, e.g., in order to handle interoperability in the context of very large vocabularies [49]. One of the mechanisms that can be used in order to connect an XML Schema to external metadata is the *Gleaning Resource Descriptions from Dialects of Languages* GRDDL [32], which is currently developed by the W3C.

XML Schema itself provides the possibility to embed the required information and to incorporate pointers to external metadata by means of the *annotation* Schema component. The corresponding constructs in the transfer syntax are the `xs:appinfo` element and the attribute wildcard present for every element of the transfer syntax. Both can be regarded as *hooks*. The former is especially powerful because it can contain structured content. The latter is readily amenable to RDF, which heavily uses URIs that perfectly fit into XML attributes.

anyway.

2.4 Related Technologies

A few related technologies, which we will encounter in the following sections, are briefly introduced in the following. They are related in different ways; two of them utilize or integrate XML Schema, while RELAX NG is XML Schema's most serious competitor. It is not the goal of this section to discuss these technologies in detail. The reader may refer to the respective authoritative descriptions cited.

2.4.1 XSLT 2.0

As mentioned above, XSLT 2.0 [54] is a *typed* language, and its type system is based on the type system of XML Schema. More precisely, it is "typed" in two ways: First, it can be employed as a *strongly typed* programming language, and second, it can operate on type-annotated input trees.

The former aspect means that XSLT 2.0 works on typed values. It performs type checking and may throw errors, if operations are attempted to be carried out on values which have a type that does not allow this operation to be applied. An important subclass of type-specific operators are comparison operators. XSLT 2.0 offers typed comparison. Furthermore, it enables the developer to declare the type of variables and input parameters, and it comprises language construct for testing and casting types such as `instance of`, `castable as`, and `cast as`. This dimension of typed behavior clearly distinguishes XSLT 2.0 from its predecessor, XSLT 1.0, which was only a *weakly typed* language. (XSLT 1.0 only knows very basic types — nodes, numbers, text, and boolean — and tries to silently typecast values, which makes the language very hard to debug.) Typed programming is available to all XSLT 2.0 processors, although basic processors only support a subset of the built-in simple types from XML Schema. A brief excerpt of sample code illustrates how an arithmetical operation fails when attempted to be applied to a string value:

```
<xsl:variable name="x" select="'3'" as="xs:string"/>
<xsl:value-of select="$x - 1"/>
saxon@work> Unsuitable operands for arithmetic operation (string, integer)
```

One solution is to typecast the value explicitly:

```
<xsl:value-of select="$x cast as xs:integer - 1"/>
saxon@work> 2
```

The latter aspect of the two, i.e., operation on typed input, is reserved to Schema-aware processors. This class of processors also support the whole set of built-in XML Schema types. In order to work with type-annotated input trees, the processor has to be advised to validate the input, which results in the aforementioned augmentation of the Infoset, i.e., the PSVI. This is not to be confused with the second type-related

capability of Schema-aware processors, which is utilization of user-defined Schema-types within the style sheet. After importing the required XML Schema documents, a Schema-aware processor allows one to use the simple types from an XML Schema for typed programming in the way described above. Imagine a user-defined type for TLAs:[8]

```
<xs:simpleType name="TLA">
 <xs:restriction base="xs:token"><xs:length value="3"/></xs:restriction>
</xs:simpleType>
```

After importing the defining Schema document, this simple type can be used in the following way:

```
<xsl:value-of select="if ('XML' castable as ex:TLA) then 'yes' else 'no'"/>
saxon@work> yes
```

Finally, XSLT 2.0 distinguishes different *node kinds*. These were called node "types" in XSLT 1.0, but in order to avoid confusion with the types of XML Schema, they have been renamed node "kinds". In fact, the category of node kinds is orthogonal to the notion of types, and combining the two is possible. Moreover, both are often used in a similar way, e.g., for indicating the type of a variable, or for asserting the type of a parameter. Node kinds are best explained by example: Possible node kinds are `item()`, `text()`, `node()`, `element()`, `attribute()`, and so on. The latter two kinds can take arguments; either an element/attribute name (which can be a wildcard), or a name and a type name. While the first case is available in all processors, the second one is only supported by Schema-aware processors. Schema-aware processors additionally provide the node kinds `schema-element()` and `schema-attribute()`. Both take a name as argument and they can be used to match typed elements and all members of their substitution group, if any. A final example demonstrates the use of node kinds. Assume an XML Schema that defines the following type hierarchy:

```
<xs:complexType name="baseType">
 <xs:sequence>
  <xs:element ref="ex:nested" minOccurs="0"/>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="resType">
 <xs:complexContent>
  <xs:restriction base="ex:baseType"/>
 </xs:complexContent>
</xs:complexType>
```

The XSLT 2.0 template rule below then matches all elements which are either of type `ex:baseType` or of type `ex:resType` (because the latter is derived from the former, which causes instances of the latter to be instances of the former as well).

---

[8]According to Peterson [83], TLA is an acronym for "Three Letter Acronyms".

```
<xsl:template match="element(*, ex:baseType)"/>
```

Although the transition to strongly typed programming and the introduction of user-definable types have considerably increased the power and robustness of XSLT 2.0, the limitation of XSLT 2.0's type concept have to be pointed out: The type information is only exposed as the qualified names of the types, and the type annotation of nodes can neither be queried nor navigated. Finally, the type operators (e.g., `instance of`) available do not discern whether an argument uses a given type, or a type derived therefrom.

These shortcomings are addressed by Chapter 7, where a path language for XML Schema is presented which provides much more sophisticated and powerful means for dealing with XML Schema types.

2.4.2 RELAX NG

RELAX NG [31] presumably is the most serious competitor of XML Schema. In contrast to XML Schema, it has been standardized[9] by *Organization for the Advancement of Structured Information Standards* (OASIS), a non-profit consortium of different companies engaged in e-business. RELAX NG was standardized in the same year as XML Schema, but contrary to XML Schema, it essentially is the work of two single persons and their proposals for a schema language, rather than being based on consensus of a large and heterogeneous working group.

The scope and objectives are clearly defined: RELAX NG aims to be "a simple schema language for XML." The below definition of a RELAX NG schema, which is taken from the abstract of the specification, reminds of the discussion of document grammars at the beginning of Section 2.

> A RELAX NG schema specifies a pattern for the structure and content of an XML document. A RELAX NG schema is itself an XML document.

This implicitly defines what RELAX NG purposely is not: It is not a data modeling language, and it is not a format for keeping external or additional data, eventually augmenting the instances which it constrains. This clearly distinguishes RELAX NG from its competitor, XML Schema. This initial strict confinement also assists the accomplishment of a second goal, which is the sound formal description of the schema language and its semantics. Further differences from XML Schema are the absence of an own type system or library, the availability of a compact non-XML syntax, and the pursuit of maximal flexibility. In fact, the categorization based on formal language theory in Section 4.2.4 will prove that in terms of expressiveness, the class of schema languages,

---

[9]RELAX NG is also standardized as ISO/IEC 19757-2:2003, as a part of the *Document Schema Definition Languages* (DSDL).

which RELAX NG is a representative of, is indeed more powerful than the class XML Schemas falls into.

In order to make the XML syntax amenable to a rigid formal description, RELAX NG defines a *full syntax*, a *simple syntax*, and precise conversion rules that govern conversion of schemas from the full into the simple syntax. While the full syntax has a richer set of language elements (making it more convenient to use), the simple syntax is very restricted in its language constructs, but rather cumbersome and lengthy. For instance, the full syntax provides means for including external content, whereas simplification requires all references to be resolved, and all external resources to be inserted in-place. Another example of a simplification step is the replacement of elements with mixed content by an explicit interleaving of the permitted child nodes and `<rng:text />` nodes. This reminds of the way in which DTDs express mixed content, and it is easily recognizable as being in direct correspondence to formal models like hedge grammars, which in fact is the formal foundation RELAX NG builds upon, and which we will discuss in Section 4.2.3.

Besides having a solid formal foundation, the specification of RELAX NG also formally defines the semantics of a correct RELAX NG schema. The semantics describe the simple syntax. Due to the simplification rules given, they also hold for every correct RELAX NG schema in the full syntax. The semantics are defined using axioms, inference rules, and well-defined variables, where necessary. The notation and interpretation are diligently described in the specification.

RELAX NG is a clearly defined alternative to XML Schema with distinctive characteristics, and it explicitly addresses a narrower range of applications, i.e., document validation in the sense of "checking whether the structure and contents of a given document instance conform to a certain document grammar". Because of its accurate and concise definition and because of its formal foundation, it is often considered to be superior to XML Schema, at least from an academic point of view. Other advantages are the presence of a compact syntax and the higher expressiveness. However, one has to bear in mind that typed processing as described in Section 2.4.1 above would not be possible with this restricted class of schema languages.

2.4.3   WSDL

In the context of *Web-based services*, which we discuss in Section 3.2, the *Web Services Description Language* (WSDL) [30] plays a key role. Basically intended to describe messages and operations of a network service, it is often (mis)understood, or even employed, as a way to describe the services itself, although it provides only a syntactical description of the elements of a service, rather than giving information about the semantics.

WSDL matters in the context of this report because it allows XML Schema to be used for describing the structures involved.[10] XML Schema definitions are embedded within

---

[10]In fact, the specification refers to XML Schema as WSDL's "canonical type system", and in practice,

WSDL documents at well-defined points (e.g., within `<wsdl:types />`). Recalling Section 2.3.3, this is yet another form of *composite schemas*, where Schemas are parts of third formats, rather than containing additional information within the Schema.

WSDL is a good example of how XML Schema is used as a basis for other W3C technologies. Under the (admittedly debatable) assumption that WSDL actually provides a *description* of a service (or of a *service interface*, at least), it also exemplifies a use case where XML Schema is employed as a modeling language.

2.5   Data Models

The XML Schema recommendation defines an abstract data model. Despite this, a variety of data models for XML Schema are currently available. They all differ with respect to each other as well as from the abstract data model, depending on their field of application. It appears that the presence of an abstract data model does not rule out the need for different *data model perspectives*. The differences in the data model may be caused by particularities of the implementation environment (e.g., the programming language chosen), or they may be directed by the anticipated way of utilization. XML Schema programming APIs which follow the paradigm of the *Document Object Model* (DOM) [57] might serve as an example of the former case. An object tree is a very natural way of representing structured data in object-oriented languages like Java, although it requires adjustments and simplifications to be made with respect to the abstract data model, which is a general graph rather than a tree. The perspective that the PSVI provides might serve as an example of the latter case. The PSVI only reveals parts and aspects of XML Schema's data model that are deemed to be important in the context of XML document processing.

Several DOM-like programming APIs for XML Schema are available today. Microsoft's XML Schema Object Model (SOM)[11], Eclipse's XML Schema Infoset Model (XSD)[12], the XML Schema Object Model (XSOM)[13] from java.net, and the Castor XML Schema Support[14] are the most prominent examples. The *Schema Component API* proposed by Litani [58, 59] augments the DOM API for XML instances with methods for accessing Schema information (i.e., the PSVI). Even though it aims at representing all Schema component properties of each component, it is unclear whether all components are exposed to the API (e.g., named groups or identity constraints), and whether full navigability is preserved. The *DOM Level 3* originally included an "Abstract Schemas Specification" [25] that ambitiously attempted to provide

---

usually no other schema is used.

[11]http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconXSDSchemaObjectModelSOM.asp

[12]http://www.eclipse.org/xsd/

[13]https://xsom.dev.java.net/

[14]http://www.castor.org/xmlschema.html

> [...] a representation for XML abstract schemas, e.g., DTDs and XML
> Schemas, together with operations on the abstract schemas, and how such
> information within the abstract schemas could be applied to XML documents
> used in both the document editing and abstract schema editing worlds.

This goal was very hard to achieve, and the project was abandoned later. Because of the
substantial structural differences, covering multiple schema languages is only possible at
the price of a very high level of abstraction. Reconciling the needs of areas of application
as diverse as instance editing and schema editing is a very difficult task as well, as
Section 5.1 shows.

Although striving to formally describe the abstract data model, the *Formal Description*
of XML Schema [19] also introduces a different data model, for example by omitting
identity constraints.

Section 2.4 of the *XQuery 1.0 and XPath 2.0 Formal Semantics* [36] describes the type
system of XML Schema, appendix D of the same document describes how import of
XML Schema shall be done. Both sections hence model essential parts of XML Schema.
In both cases, only the parts which are relevant in the context of XQuery and XPath
2.0 are described. This of course again leads to a data model different from the abstract
data model.

In conclusion, it can be said that in most of the cases a suitable data model, data model
interface, or API is available. And if it is not, new ones tend to be developed, or existing
ones are adapted. In [66], the author argues that a unified data model for XML Schema,
designed for accessibility and equipped with a well-defined set of accessor functions, is
desirable nevertheless. The following Section 3 demonstrates this fact by giving a list of
areas where no suitable data model is at hand yet, and by listing opportunities a unified
data model would generate.

2.6 Deficiencies

In practice, XML Schema is not as popular as one might think, despite its important role
in many of the W3C's standards. It has not replaced DTDs completely, and it still faces
competition from languages like RELAX NG. In particular, empirical studies by Lämmel
et al. [56] and Bex et al. [9] show that the more advanced, modeling-oriented features
are rarely used, and that a significant fraction of XML Schemas deployed at present is
not even standard-compliant. The main reason is that XML Schema is perceived as a
complex — or even overly complex — language. Both the XML syntax of XML Schema
and the recommendation are complex indeed. The wording in the recommendation is
unfortunate in some parts, and not always beyond doubt. Section 2.6.2 provides critical
remarks on the recommendation. Section 2.6.3 discusses an obvious insufficiency of the
recommendation of XML Schema.

2.6.1 Schema Editing and Visualization

The XML syntax makes writing and understanding of Schemas often difficult for humans. Surprisingly, a survey of current XML Schema editors, which has been done by the author [67], concludes that none of the editors provided comprehensive graphical support for the creation and modification of more advanced Schemas. Presumably, this is also due to the complexity of the recommendation, which makes the implementation of Schema technologies difficult. It may also be due to the lack of a clearly defined interface to the data model. The remainder of this report discusses this question in detail, and it will propose approaches and possible solutions to the problem of data model accessibility.

2.6.2 Critical Comments on the Recommendation

An essential cause of many problems when dealing with XML Schema is the size of XML Schema's recommendation, and the fact that the recommendation is only available in prose. Initially, a formal description was planned [19], but this endeavor was abandoned later. This is a severe disadvantage. A formal description would be extremely helpful for the understanding of the recommendation. A formal description rules out ambiguities and inconsistencies. Implementations thus would become more consistent and more uniform in behavior. Inconsistent behavior of Schema-validators and Schema-checkers is a major problem and is likely to discourage the use of XML Schema.

A point criticized often is that the recommendation is too *ad-hoc* in many parts. Frequently, the recommendation specifies an abundance of rules and multi-leveld sub-rules, where it would have been more concise to define the goal that a certain mechanism should achieve. Some constraints (e.g., UPA) are semantical in nature, where a syntactical definition would be less difficult to be understood by users;[15] and some constraints (e.g., type derivation) are defined intensionally, where an extensional definition would leave less space for ambiguities. If a rigid formal description was present, the utilization of ad-hoc solutions, corrections, and additions to the recommendation were probably less frequent.

The recommendation contains several sections which are marked as *non-normative*. Unfortunately, many of these non-normative sections are often used and referenced. For instance, the *Schema components diagram* from Appendix E of the recommendation is a very useful visualization, which can be encountered quite regularly. However, the diagram contains obvious errors and omissions, which is annoying. This is also true for the non-normative specification of the simple type definition component in Part 1, which is inconsistent with the normative specification in Part 2. Non-normative sections should not be written and edited with less diligence than normative sections.

---

[15]A fact criticized by Bex et al. [9].

The recommendation includes the specifications of the abstract data model, of the XML syntax, of the PSVI, and of the ways validation should be performed — all in one document. This compromises the conciseness and intelligibility of all four specification parts. A separation — similar to the separations done for the various XPath 2.0-related technologies — would be desirable.

This merging of specifications also affects the definition of the abstract data model, because the definition of abstract Schema components is not always sufficiently independent from the definition of the transfer syntax. Often, the exact semantics of a component property are only completely defined if the definition of the respective XML representation is taken into consideration as well. This must not be the case for an *abstract* data model.

The way in which the transfer syntax defines type derivation is overly complex and arguably inconsistent.[16] It is debatable whether or not rarely used features (of doubtable usefulness) like *nillability* and *notation declarations* should be part of XML Schema. Finally, there is consensus that some default mechanisms are poorly chosen. The attribute `elementFormDefault` should have the default value `true` rather than `false`, and the default attributes `final` and `block` are semantically ambiguous because they govern two semantically different properties of element declarations and type definitions.

2.6.3 Insufficiencies

In our opinion, the most grave insufficiency of XML Schema's abstract data model is that it has *no clear concept of component identity.*

The recommendation does not explicitly say how components can be identified, and it does not say when two components are identical, and when they are distinct. Two observations let us conclude that XML Schema's abstract data model has no consistent concept of component identity, and that this lack is a problem in practice.

1. It is not always clear whether two components are identical or not.

   For example, it is unclear whether inherited element declarations are identical to the element declarations in the base type.

2. There is no way to identify Schema components in a concise and unambiguous way.

   Identifiers which rely solely on component properties fail at distinguishing anonymous simple type definitions within constructions of union types.

In March 2005, the W3C has published a working draft for *XML Schema component designators* [50]. These designators have a path-like syntax in order to designate components — including anonymous ones. In our opinion, this syntax is too clumsy, and

---

[16]Different rules apply to the inheritance of content models, attributes, and attribute wildcards.

too much structure-oriented. Moreover, it does not address the fundamental problems of component identity.

The presence of the component properties {context} and {parent} in the type definition and element declaration Schema component, respectively, are presumably intended to distinguish local Schema components, although the recommendation does not state that explicitly. Assuming that these component properties are used for distinction (we actually cannot think of another use), type definitions, element declarations, and attribute declarations then would have recursive identifiers of the following form:

- A 3-tupel <{name}, {target namespace}, {context}> for type definitions

- A 3-tupel <{name}, {target namespace}, {parent}> for element and attribute declarations

Unfortunately, the component properties {context} and {parent} are defined in a rather cumbersome way, because the definitions are context-dependent. (For element declarations, e.g., {parent} is a complex type definition or a named group; for simple type definitions, five different cases have to be distinguished.) And finally, the concept fails completely for anonymous simple type definitions within union constructions. Such type definitions cannot be distinguished only by means of the 3-tupel above.

It is essential to be able to distinguish and identify Schema components, and component identity should be well-defined. Hopefully, forthcoming draft versions of XML Schema 1.1 will address this problem.

Chapter 3

The Case for Data Model Accessibility

In an earlier paper [66], we stated the necessity and usefulness of an accessible data model for XML Schema. In this section, we demonstrate this in more detail by describing fields of application that demand data model accessibility or will benefit therefrom.

We advocate a *general* concept of accessibility for XML Schema, as opposed to the specialized APIs listed in Section 2.5. We require such a concept to be comprised of three parts:

1. *A unified data model.* In contrast to the *abstract* data model, expected fields and ways of use should be taken into consideration for the design of such a data model, as well as constraints of representation in implementations. The data model should include well-defined rules for *canonicalization.*

2. *Unique component identifiers.* A coherent and concise concept of *component identity* is indispensable. Based on this, unique *identifiers* have to be defined for identifying and comparing components.

3. *Well-defined accessor functions.* Access of and operations on the data model should be backed by a set of well-defined accessor functions. These may comprise functions for navigating the component structure and their relationships, or functions for extracting component properties.

It becomes apparent that such a concept exists for XML instances. The XML Infoset fulfills the requirements of the first part. The concept of instance nodes (i.e., Infoset information items) together with the basic core functionality of XPath location paths meets the second. And the set of XPath functions is in accordance with our third postulation.

With the arrival of XPath 2.0, XQuery 1.0, and XSLT 2.0, the data model of XML in-

stances is defined very precisely as the *XQuery 1.0 and XPath 2.0 Data Model* (XDM) [42]. This recommendation meets the above reqirements even more closely. It precisely declares the context of application (i.e., XPath 2.0 and the technologies that build upon XPath 2.0: XSLT 2.0 and XQuery 1.0), it explicitly defines node identity (in Section 2.3 of [42]), and it defines a set of abstract accessor functions. Finally, *XQuery 1.0 and XPath 2.0 Functions and Operators* [61] describes functions and operators available in XPath 2.0 and in XQuery 1.0.

What we request is a comparable conceptual framework for XML Schema. However, in the case of XML Schema, this is not easily done, and the necessity to do so is less obvious. Therefore, the below sections demonstrate the need for an accessible, unified data model for XML Schema.

In the remainder of this section, we look at problems which are present today, or which will arise in the foreseeable future in important areas of application of XML (e.g., Web-based services). Subsequently, we demonstrate further opportunities that an accessible data model will generate. Obviously, solutions to current problems and future opportunities overlap in large parts.

3.1 Versioning and Extensibility

XML is an open and highly flexible data format, and it is thus readily amenable to versioning and extensibility. XML Schema was designed with these properties in mind from the very beginning. It intends to support versioning and extensibility by various features. XML namespaces, type derivation, wildcards, and substitution groups are the most notable among these. Despite the presence of this support for versioning, only little experience with Schema-versioning seems to be available today, and most probably only a small fraction of Schemas has truly been designed for versioning and extensibility. In addition, XML Schema 1.0 makes design for extensibility difficult in some cases. For instance, the *Unique Particle Attribution* constraint often conflicts with the demand for flexible element wildcards. The current draft of XML Schema 1.1 alleviates some of the issues. The next draft will presumably introduce even more advanced features for supporting extensibility. In fact, most of the relevant changes in XML Schema 1.1 are addressing schema versioning and extensibility.

Although the subject is of high importance for XML, XML Schema, and many of XML's primary areas of application, versioning and extensibility of XML vocabularies and its mechanisms have not yet been discussed to a great extent in scientific publications. There is a variety of theories, strategies, and principles which address the topic in various fields of software engineering. But the issues and requirements for XML vocabularies are specific ones, and the particularities of both the context of application scenarios and the XML Schema language have to be taken into account. In this specific area, influential work has been done by Orchard. Yet most of his work is only available as *online articles*

or *blog posts*. The same holds for many other contributions on that subject. For a commented list of online resources, we refer to a small version bibliography assembled by the author.[1]

3.1.1   Nomenclature

In this section, we briefly introduce the essential terms and principles related to versioning. For a detailed discussion of terminology, principles, and best practices in the specific context of XML, see Orchard and Walsh [80].

Versioning denotes the evolving of a software application or a data format or a protocol over time. It is usually governed by one instance, typically the company selling the application or the consortium defining the format or protocol. If we neglect versioning branches, versioning happens in a sequential and linear manner. A newer version supersedes (and thus replaces) an older one, which then is likely to disappear over time. Versioning often distinguishes *major* from *minor* changes. Minor version changes commonly imply compatibility with peers of the same major version. Typical examples of minor version changes are bug fixes or security updates in applications, and corrections and clarifications in specifications.

Extensibility is similar to versioning with respect to the resulting differences. But unlike versioning, extensions take place synchronously. While versioning occurs over time, extensions occur distributed in space. Extensions are typically not controlled by a single instance, but are applied by the user.[2] *Extensibility*, i.e., the planning and preparations for possible extensions, is in the responsibility of the creator of an application, format, or protocol. It is important to note that extensions co-exist, and thus potentially have to interact with each other. Finally, "extension" implies that there is a common core shared by all variants that have been created through extension.

Backward compatibility means that a more recent version can still cooperate with an older one. Usually this means that the newer version accepts and correctly processes output from older versions. Usually, backward compatibility is ensured between minor versions, whereas major versions often break backward compatibility.

Forward compatibility denotes the ability to interact with future versions. Design for forward compatibility strives to anticipate future changes and introduces a due amount of flexibility and tolerance. Forward compatibility resembles extensibility in many of its requirements and issues, and it leads to similar development strategies. It is obvious that forward compatibility is only possible up to a certain degree.

---

[1]Retrievable from http://people.ee.ethz.ch/~femichel/XML/version-bib.html or http://www.webcitation.org/5Mw2PCqID (archived).

[2]In UBL, extensions are called *customizations*, reflecting this aspect.

3.1.2 Versioning Support in XML Schema

As mentioned already, the predominant versioning features in XML Schema are XML namespaces, type derivation, wildcards, and substitution groups. Naturally, namespaces are not a suitable means for extensibility, but the remaining features can be employed for both versioning and extensibility.

Namespaces: Orchard [76, 77] advocates the following rules-of-thumb for deciding whether or not an XML namespace should be reused:

> *Re-use namespace names* Rule: If a backwards compatible change can be made to a specification, then the old namespace name SHOULD be used in conjunction with XML's extensibility model.

> *New namespaces to break* Rule: A new namespace name is used when backwards compatibility is not permitted, i.e. software MUST break if it does not understand the new language components.

However reasonable these rules seem, there are prominent examples which do not follow these principles. XSLT 2.0 uses the same namespace, although it is a major version change. Compared to XSLT 1.0, version 2.0 significantly extends the vocabulary and, in some cases, breaks backwards compatibility. The contrary example is the migration of UBL version 0.7 to version 1.0. Although the Schema did not undergo substantial changes, the namespace was changed in order to indicate its status as an official release.

In both cases the problem is that XML technologies (e.g., XSLT) only understand the Schema components in terms of their qualified names. Looking only at the qualified names, components are not discernible as long as they bear the same qualified name, regardless of any changes in the syntax or semantics. Vice versa, structurally and functionally identical components are regarded as distinct and independent as long as their qualified names differ.

The decision when to reuse a namespace is a design decision which is arbitrary to a certain extent. Collections of *Best Practices* for XML Schema like [33] provide some guidance. It will be interesting to see whether the XML Schema working group decides to reuse the namespace of XML Schema 1.0 for version 1.1.

The question whether or not XML namespaces should be utilized for designating version changes is related to the question of what the semantics of XML namespaces are in general. Although XML namespaces are a concept separate from XML Schema, XML Schemas are often said to be *structuring a certain namespace*. The XML Schema recommendation uses the expression of XML namespaces *identifying an XML vocabulary*. However, the original recommendation of XML namespaces — which was published before the advent of XML Schema — merely described namespaces as a way of avoiding collision of names [16]. It states that the possible collisions

> [...] require that document constructs should have universal names, whose scope extends beyond their containing document. This specification describes a mechanism, XML namespaces, which accomplishes this.

Consequently, XML namespaces are defined as "a collection of names, identified by a URI reference". The lack of a defined inner structure[3] or specified semantics complicates the discussion about the use of namespaces in the context of Schema versioning. The namespace URIs themselves are not suited for representing namespace structures. They are simply unique identifiers of entities, and abuse of fragment identifiers is no viable choice. Wilde [97, 98, 102] and Halpin [47] propose mechanisms for structuring namespaces, the latter explicitly addressing versioning of XML namespaces.

The Version Attribute: The transfer syntax of XML Schema offers a `version` attribute on the root element of Schema documents. Unfortunately, this attribute is simply of type `xs:token`, and it thus shares a deficiency of namespace URIs, i.e., the deficiency of not being able to express structured information. Furthermore, it has no equivalent in the abstract data model (i.e., in terms of Schema component properties),[4] and the mapping to Schema components is problematic because the attribute is defined on Schema *documents*. Thus, multiple different version attributes can potentially be attached to one *assembled* Schema through XML Schema's include mechanisms. All in all, the version attribute of the transfer syntax is not very useful for versioning purposes, and it is rarely used in practice.

Type Derivation: Besides supporting code reuse and modeling of meaningful relationships, type derivation in XML Schema is an essential feature for versioning and extensibility. The effects of type derivation are not only confined to the design and the maintenance of Schemas. Through type substitution and substitution groups, type derivation also affects instance documents directly. For this reason, great diligence was devoted to ensuring compatibility among derived types.

The first of possible mechanisms of derivation, i.e., *derivation by restriction*, is designed in such a way that every instance of a restricted type is a valid instance of the base type. This is the utmost level of compatibility. Essentially, the base type would not have to know about the additional restrictions, and validation could still take place as usual. The brief example in Figure 3.1 illustrates the concept of restriction. Only parts which are optional in the base type are allowed to be restricted away in the derived type. Thus, the content model allowed by the derived type (in the example, the empty content model) is always a valid content model with respect to the base type.

---

[3]While the first edition [16] contained a non-normative appendix entitled "The Internal Structure of XML Namespaces", the second edition [17] omits this section.

[4]Even worse, the XML Schema recommendation explicitly states: "The other attributes (id and version) are for user convenience, and this specification defines no semantics for them." Given the attribute's name, it is certainly misleading to claim the attribute not to have any semantics.

```
<xs:complexType name="base">
 <xs:sequence>
  <xs:element name="nested" minOccurs="0"/>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="restricted">
 <xs:restriction base="ex:base">
  <xs:sequence/>
 </xs:restriction>
</xs:complexType>
```

Figure 3.1: Complex type restriction

Derivation by extension ensures that the inherited content model remains unchanged. Extensions are always added to the end of the model group. The addition is defined to be done as a sequence. Consequently, the additional parts always appear *after* all other parts. The set of instances of an extended types is a *superset* of the instances of the base type rather than a *subset*. However, the fixed position of the extension elements provides for resilient processing. An application that does not understand the extension can always omit the additional parts and only validate the inherited part of the content model. However, we will see in a moment that this is not always trivial.

The above *set-theoretic considerations* are a powerful aid for describing type derivation. Yet version 1.0 of the recommendation specifies type inheritance in an intensional way, which apparently leads to some (presumably unintended, and therefore) undesired corner-cases. Version 1.1 of the recommendation is expected to employ the extensional definition of type derivation. This will make the recommendation more clear, and hence, implementations will become more consistent.

Substitution groups build upon type derivation, and wildcards essentially contribute to the power of type derivation. Both are described below.

It is important to recall that type derivation is the only way to express semantic relationships between components from different versions, and this only holds true if the related components are accessible. I.e., the components have to be in the same (assembled) Schema. This is possible either by using one namespace for all components, or by importing the namespaces of all previous versions. The *Universal Business Language* (UBL), the most comprehensive open XML vocabulary for global e-business, and probably one of the largest and most maturely designed XML Schema libraries, intensively utilizes the more advanced features of XML Schema. In addition to the constraints that XML Schema imposes regarding the derivation of types, UBL constrains versioning of vocabularies in a very concise manner. Minor versions are only allowed to be created by applying type derivation, every minor version is *packaged* within an own namespace, and each minor version must import the namespace of the previous minor version. Using these three principles, UBL achieves a cascade of XML Schemas, resulting in one assem-

bled Schema which comprises all minor version components. As a result, the version history of each component is entirely described by XML Schema, and applications can employ sophisticated fallback strategies.

Gregory and Gutentag [45, 46] demonstrate how this strategy, together with type-aware processing, for example in XSLT 2.0, enables polymorphic processing (i.e., applications can discern different versions and act appropriately — this includes fallback to previous versions), and they conclude:

> This subject, although a bit complex, is worth exploring - it provides perhaps the strongest argument for using namespaces as the package for versioning, although in a non-obvious fashion. The basic proposition is this: by allowing, between minor versions, only changes permitted by XSD extension and restriction, we can ensure that all minor versions of an element are backward-compatible.

The above quotation clarifies how XML Schema is the key to versioning of XML vocabularies. XML Schema is capable of expressing relationships between components, both synchronic and diachronic ones, if we may employ a term from linguistics. This insight, together with the awareness that polymorphic processing and fallback strategies are the keys to versioning-resilient processing, makes the need for an accessible data model for XML Schema obvious. Only an accessible data model allows the Schema information to be exploited.

We emphasize this by giving an example: Orchard [78] proposes the introduction of a new attribute `xsi:basetype`. The idea is to indicate the base type of a type *in the instance* to processors that do not understand a derived type, and therefore may want to validate the instance node against the base type of its actual type. Knowing that the complex type definition Schema component contains a property {base type definition}, it becomes obvious that the presence of accessible and navigable schema information eliminates the need for such an attribute.[5]

Type Redefinition: The transfer syntax of XML Schema allows types to be *redefined*, either in a new or in the same namespace. Even though this functionality is sometimes recommended for versioning (in fact, the recommendation lists "evolution and versioning" as reasons for its introduction), redefining types in our opinion is not a good versioning strategy. This is mostly due to the fact that the modifications caused by `xs:redefine` are *pervasive*. In consequence, the original definitions of redefined types are no longer present in the data model, and neither introspection of the version history nor Schema-aware fallback strategies are possible. Redefinition is only a construct of the transfer syntax and has no representation in the abstract data model.

---

[5]One could argue that Orchard addresses the case where the most recent Schema is not available at all. However, the required Schema typically is retrievable, but applications written against the original Schema cannot access, and therefore utilize the schema information.

Substitution Groups: Substitution groups are based on type derivation. The members of a substitution groups are always required to use types that are derived from the type of the substitution group head. Substitution groups have been designed with extensibility in mind. Whereas type extension can be regarded as a way of extending *sequence* model groups, substitution groups can be thought of as a kind of extensible *choice* model group or "late-binding choice".[6]

```
<xs:element name="realname"
    type="xs:string"/>
<xs:element name="avatar"
    type="xs:token"/>


<xs:element name="person">
 <xs:complexType>
  <xs:choice>
   <xs:element ref="ex:realname"/>
   <xs:element ref="ex:avatar"/>
  </xs:choice>
 </xs:complexType>
</xs:element>
```

```
<xs:element name="realname"
    type="xs:string"/>
<xs:element name="avatar"
    type="xs:token"
    substitutionGroup="ex:realname"/>

<xs:element name="person">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="ex:realname"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

Figure 3.2: Choice model group compared to a substitution group

From a structural point of view, the results of a choice model group and an element which may be replaced by members of its substitution group can be regarded as equivalent, because the set of valid documents is identical. Figure 3.2 displays two alternative definitions of the model group of the element `ex:person`. It becomes apparent that the two document structures shown in Figure 3.3 represent the complete set of allowed instance structures for both variants.

```
<ex:person><ex:name>Erik Wilde</ex:name><ex:person>
```
```
<ex:person><ex:avatar>dret</ex:avatar><ex:person>
```

Figure 3.3: Valid document structures

But contrary to the choice model group in Figure 3.2, the substitution group is extensible, both in customizations and in future versions, and even from other namespaces, as exemplified by Figure 3.4.

```
<!-- import xmlns:ex -->
<xs:element name="email" type="xs:token" substitutionGroup="ex:realname"/>
```

Figure 3.4: An external element that extends the substitution group

---

[6]Obasanjo [75] compares substitution groups to the concept of *subtype polymorphism* in object-oriented programming.

While this is a very powerful feature for extensibility, it causes the same kind of problem as does type extension. Siméon and Wadler [88] point out that type derivation by extension is a kind of type derivation fundamentally different from type derivation in most object-oriented programming languages.

> In languages such as Java, one can typecheck code for a class without knowing all subclasses of that class (this supports separate compilation). But in XML Schema, one cannot validate against a type without knowing all types that derive by extension from that type (and hence separate compilation is problematic).

This also holds for content models that can be extended by adding elements to a substitution group. It is more difficult to come up with consistent fallback strategies for these cases than for type restriction, where the base type always is a valid description of every instance of all derived types, although probably too tolerant.

The approach of *validation by projection* can serve as a workaround. We will describe this approach in Section 3.4. The possibility to access, navigate, and explore the Schema certainly is beneficial in either case because it allows an application to determine (at runtime) which legal extensions the Schema describes — be it through type extension or through substitution groups.

3.1.3 Changes in XML Schema 1.1

Wildcards: Wildcards are available in XML Schema 1.0 as well, but they have been redesigned substantially. For this reason, we discuss wildcards as an example of the changes applied by version 1.1 of XML Schema.

Element wildcards permit the appearance of arbitrary top-level elements (i.e., elements which are declared globally) in the instance. The set of allowed elements can be limited by the attribute `namespace`. The possible values in XML Schema 1.0 are roughly either a list of allowed namespaces, the current target namespace of the schema, or `##other`, a special token which indicates that elements from all namespaces, except the current target namespace, are permitted. XML Schema 1.1 increases the flexibility by providing an attribute `notNamespace`, which allows to exclude namespaces explicitly.

Content models in XML Schema must comply with a restrictive kind of unambiguity called *Unique Particle Attribution* (UPA). It requires that while parsing an instance each node can be related to at the most one particle in the content model, without looking ahead in the instance. For example, $(a?, a)$ violates UPA because an element a in an instance cannot unambiguously be mapped to a particle. This constraint limits the use of wildcards very often. Wildcards that allow elements from the current target namespace are not allowed to appear directly after optional elements, or directly in front of elements, if the wildcard itself is declared optional. The following content model is illegal:

```
<xs:sequence>
 <xs:element ref="ex:a" minOccurs="0"/>
 <xs:any namespace="##any"/>
</xs:sequence>
```

The reason is that the element `ex:a` is contained in the set of possible matches of the adjacent wildcard. Therefore, the above example violation $(a?, a)$ is contained as well. The current draft of XML Schema 1.1 addresses this problem in two ways. First, it provides a new attribute `notQName`, which can contain a list of qualified names denoting elements that should be excluded from the set of elements allowed by the wildcard. This already solves the above problem. By excluding `ex:a` from the wildcard, the content model satisfies UPA. The second modification in XML Schema 1.1 is that UPA is relaxed insofar as wildcards are allowed to be competing with particles. (Competing wildcards and competing particles are still forbidden, nevertheless.) Content models as the following are legal in XML Schema 1.1:

```
<xs:choice>
 <xs:element ref="ex:a"/>
 <xs:any namespace="##any"/>
</xs:choice>
```

Forthcoming draft versions maybe may introduce even more radical changes. One idea is to allow to declare content models as open. This is related to the idea of *default wildcards*, another possible new feature in future draft versions. No matter which features will be included in future draft versions, a trend towards extensibility in the XML Schema working group can be clearly observed.

3.1.4 Conclusions

Versioning and extensibility of XML vocabularies is an important field. It will increasingly move into the focus as more vocabularies are deployed and as the first major version changes become urgent. XML Schema is designed for — and suited for — versioning and extensibility. The awareness of the XML Schema working group with respect to this fact is reflected by the changes in version 1.1 and by the efforts of publishing a *Guide to Versioning XML Languages using XML Schema 1.1* [79]. Many of the present problems with versioning are due to the fact that useful information captured by Schemas is not accessible to applications, or at least not in a way that is flexible, adaptable, and fine-grained enough.

3.2 Web-Based Services

One of the driving forces of what is perceived by many as the *second wave of the Internet revolution* can be summarized under the term "Web-based services". The abundance of standards for *Web Services* published by the OASIS consortium as well as the strongly

advertised paradigm of *Service Oriented Architectures* (SOA) are the best-visible manifestations of this trend (or *hype*). While it is beyond the scope of this report to discuss Web-based services in detail, we want to briefly investigate the consequences for XML vocabularies.

Many of the problems that arise in the context of Web-based services are closely related to versioning and extensibility. In the specific context of message exchange, the definitions of forward and backward compatibility can be reformulated as:

- *Backward compatibility* means that new versions of receivers, or consumers, still accept and understand messages from old producers, or senders.
- *Forward compatibility* means that new producers are introduced in such way that old consumers can still process the new messages without failing.

The basic paradigm of *loose coupling*, which is an essential trait of Web-based services of all kind, causes aggravated forms of the aforementioned problems of versioning and extensibility. In a distributed and decentralized — i.e., loosely coupled — setting, one has to face "asynchronous versioning" and unpredictable extensions. Strict versioning strategies are likely to be replaced by more generic versioning policies, and lack of control and foreseeability needs to be compensated by more resilient processing.

As described in Section 2.4.3, WSDL usually utilizes XML Schema for describing message structures. Therefore we conclude that XML Schema will be a key for coping with the problems of XML vocabularies in Web-based services.

We expect *run-time schemas* to become a common scenario. By *run-time schemas* we mean that the schemas which are effectively describing the received instances are different from the schema against which the applications in charge of processing these instances have been written. Run-time schemas are likely to require applications to be able to perform *type introspection* (or even *type reflection*), and to have resilient fallback strategies.

The design of network protocols has dealt with issues of this kind for several decades. Adaptation of principles from that realm thus suggests itself. The design of the *Internet Protocol* (IP) is guided by a *robustness principle* defined in RFC 1122 [13];

1.2.2 Robustness Principle

At every layer of the protocols, there is a general rule whose application can lead to enormous benefits in robustness and interoperability [IP:1]:

"Be liberal in what you accept, and conservative in what you send"

Software should be written to deal with every conceivable error, no matter how unlikely; sooner or later a packet will come in with that particular combination of errors and attributes, and unless the software is prepared, chaos can ensue. In general, it is best to assume that the network is filled with malevolent entities that will send in packets designed to have the worst pos-

> sible effect. This assumption will lead to suitable protective design, although the most serious problems in the Internet have been caused by unenvisaged mechanisms triggered by low-probability events; mere human malice would never have taken so devious a course!

This still very well describes the problems encountered in Web-based services. However, liberal consumers are not trivial to design. The specification of HTTP 1.0 [6] defines the following policy for the extension of HTTP headers.

> The extension-header mechanism allows additional Entity-Header fields to be defined without changing the protocol, but these fields cannot be assumed to be recognizable by the recipient. Unrecognized header fields should be ignored by the recipient and forwarded by proxies.

The two general rules defined by this policy can be applied to Web-based services as well. The first rule, i.e., "do not assume extensions to be recognized", is a very general rule, and may not always be applicable (if extensions *must* be recognized). The second rule is the well-known *must-ignore* paradigm. It is also used in the *Simple Object Access Protocol* (SOAP) [12]. Furthermore, it is one of the basic principles of HTML. The must-ignore rule can be interpreted in different ways. For example, it has to be defined whether only the *element* which is unrecognizable should be neglected ("ignore container"), or whether the complete sub-tree which depends on that element should be ignored as well ("ignore all"). HTML chooses the first alternative, but security-sensitive applications are likely to favor the second one.

There is also the opposite rule, which is *must-understand*, and popular processing models (e.g., SOAP) often combine the two rules. Obasanjo [75] and Orchard [77] advocate the use of *must-understand* flags for mandatory extensions. The latter even votes for inclusion of a general `xml:mustUnderstand` attribute into the XML standard.

These questions are related to general questions of *compatibility* between XML vocabularies. Dui and Emmerich [37] discuss this question[7]. They use *extents* of schemas (i.e., the set of valid documents) in order to compare schemas extensionally. They point out that "testing compatibility between XML languages typically involves an unknown and potentially infinite set of instances of that language" and that compatibility is undecidable for schema languages in general, but decidable in restricted cases, e.g., if schemas are connected by a version history. They recommend the extent of new versions to be proper supersets of the extent of previous versions.

Bau [3] counters this recommendation by recalling that it does not prevent compatibility from being broken because in practice, a *missing* part may cause a message to be rejected, e.g., if the part is required for security reasons. He advocates the use of *contracts* for ensuring compatibility.

---

[7]The study includes rule-based schema languages, and *syntactic* compatibility is differentiated from *static semantic* compatibility (the latter additionally considering co-constraints).

Wilde proposes "Semantically Extensible Schemas for Web Service Evolution" [96] and distinguishes initial semantics from application semantics, declarative semantics from non-declarative ones, and intensional (declared within the Schema) form extensional (included into the instances) ones.

3.3   Information Retrieval

As XML Schemas increasingly become a format for capturing metadata of XML documents, accessibility of Schemas promises interesting applications. A trivial example is Schema-aware data mining of instance documents. Assume the hierarchy of types representing persons in different specificity that we briefly sketched in Section 2.3.3. If an application is aware of this type hierarchy, it can semantically correlate elements in an instance document that — when looking at the instance alone — are seemingly unrelated. However, this functionality requires but a basic amount of Schema-awareness. The language elements of XSLT 2.0 which are available in Schema-aware processors are sufficient to perform this kind of data retrieval. If `ex:personType` is the base type of the example type hierarchy, then the XPath expression

```
//element(*,ex:baseType)
```

returns all elements which have types identical to or derived from `ex:personType`. If we want to further differentiate our selection, say, if we only want to select elements with types derived *by restriction*, then the basic functionality of XPath 2.0 is not sufficient. An accessible and navigable representation of the Schema would provide this functionality.

The need for more sophisticated exploring of Schemas becomes apparent in the second example from Section 2.3.3. The structure in the instance in Figure 2.2 cannot be traversed without knowledge of the Schema,[8] although the XML format describes directed graphs — which is a navigable structure *par excellence*. If the information from the Schema in Figure 2.1 were accessible, the structure of the instances becomes intelligible to the application which processes the instance. Imagine an accessor function that, for a given element, returns the set of all elements that are related to this element by means of *identity constraints*. With such an accessor function, complex non-tree structures become readily navigable, provided that the Schema adequately describes the structures.

Finally, the aforementioned *points of extensibility* in XML Schema (e.g., the `xs:appinfo` element) potentially create even more powerful facilities for exploring and exploiting metadata. For instance, if the Schema is linked to an ontology, more sophisticated reasoning may become possible while processing instance documents.

---

[8]This is not completely true, as the use of the `xml:id` attribute allows to express at least one part of the most basic kind of non-tree relationships, i.e., that something is a *key*. However, in general, our statement holds, and moreover, `xml:id` was a later addition to XML.

3.4   Validation

The novel areas of application and the problems arising in this context require us to rethink validation as well. The two main reasons are:

1. In order to realize applications that adhere to the principles of *compatibility* and the paradigm of *liberal receivers* as described in Section 3.2, validation needs to be more tolerant and adaptable.

2. Many type-aware or Schema-aware applications emphasize the second of the two aspects of Schema-validation described in Section 2.3.1, that is, *type-annotation* becomes more important than mere *validation*.

In both cases, applications require validation to become more versatile and customizable. An accessible data model allows applications to retrieve Schema information and to utilize parts thereof in order to implement customized functionality for specialized partial validation.

An intuitive approach to coping with the problems of compatibility is what Bau [3] and Orchard [78] call *validation by projection*. The basic idea is to apply the *must-ignore* rule to XML documents in a way that subsequent validation against a Schema is possible, even if the instance complies to a newer version of the Schema. This is basically done by omitting all elements that are not understood by the consuming application. In order to identify the non-recognizable parts, inspection of the Schema is essential. For this reason, the principle is sometimes known as *Schema-aware XML projection*.

The consequences of *validation by projection* with respect to the set of accepted instances is similar to the effect of inserting wildcards between each two element declarations, and it is similar to the idea of *open content models* proposed by Orchard (and maybe embraced by upcoming draft versions of XML Schema 1.1). The difference between the approaches is that in the latter cases the extensibility is managed on the producer's side, whereas in the first case the consumer handles possible extensions. It is arguable which approach is preferable. As long as the transfer syntax does not provide a convenient notation for open content models or default wildcards, projection by validation may be a valuable strategy for dealing with extensibility.

Gregory and Gutentag [46] recall that projection must be carried out carefully if the documents processed are legal documents. In the case of transaction documents which must be kept due to legal regulations, the raw data should be archived *before* applying projection.

An interesting perspective on the second aspect of Schema-validation (i.e., type annotation) is found in a blog post by Ewald entitled "Making everything optional"[9] and in the replies to the initial entry. The title already summarizes the basic concept. Ewald proposes to think about a "schema not as the definition of what this system needs right

---

[9]http://pluralsight.com/blogs/tewald/archive/2006/04/20/22187.aspx

now but as the definition of what the data should look like if it's present instead." This is a radical re-interpretation of a schema, and it is an interpretation that is only possible in the case of XML Schema and its type annotation facilities.

In its most radical form, such a Schema simply consists of a set of global definitions and a declaration of a root element which has a content model consisting solely of a wildcard with arbitrary cardinality. If one thinks of this as "sloppy validation", it might seem pointless, but looking at it as a kind of "best effort annotation", the potential for type-aware technologies becomes evident. An XSLT 2.0 stylesheet may — at least for some parts of a document — not be too restrictive in terms of grammar-compliance, but it may strongly depend on the presence of type annotations — e.g., in order to apply type introspection, run-time decisions, and Schema-aware processing.

The recent introduction of type-aware technologies have changed the way schema-validation can be employed. All novel ways of utilization will benefit from access to a unified representation of XML Schema information.

Chapter 4

XML Schema from a Formal Perspective

A key difference between XML Schema and RELAX NG is the former's lack of a formal description. Despite several attempts to provide a *formal description* or *formal semantics* for XML Schema — most notably by Brown et al., on behalf of the XML Schema working group [19] — neither is available for XML Schema today. A formal description is a highly desirable feature nevertheless, especially as the related technologies become more powerful and mature.

Section 4.1 emphasizes the need for a formal foundation of XML Schema, and Section 4.2 gives a brief overview of the parts of the theory of formal languages which are applicable to schema languages. Section 4.3 summarizes different schema-specific approaches, while Section 4.4 focuses on a few interesting properties of formal language theory which promises to have useful applications.

Within the limits of this chapter, the subject can only be discussed superficially. For more detailed and more authoritative information, the respective references should be consulted, or the excellent book by Hopcroft, Motwani, and Ullman [51].

4.1   Use of Formal Foundations

The ensemble of type-aware XML technologies based on XPath 2.0 (i.e., XPath 2.0, XSLT 2.0, and XQuery 1.0) is the first group of technologies among the ones controlled by the W3C to have a solid formal foundation. The *XQuery 1.0 and XPath 2.0 Formal Semantics* [36] states:

> A rigorous formal semantics clarifies the intended meaning of the English specification, ensures that no corner cases are left out, and provides a reference for implementation.

Conciseness of specification is one reason, others include the applicability of calculus, for instance comparability, analysis and prognosis (e.g., of computational expressiveness and complexity), and verifiability. From a software engineering perspective, code optimization or even code generation is of great interest.

The first time formal languages became important in engineering was in the last century, when formal language theory enabled the effective implementation of lexical analyzers, parsers, and compilers. In fact, the theory of formal languages played the key role for the success of compilers and the introduction of higher-level programming languages.

In analogy, the investigation of formal descriptions of XML and the technologies that work with XML is motivated by the hope of increasing the potential for optimization, robustness, verifiability, code generation, and compilation from higher-level, more descriptive languages (e.g., for XML pipelines). XQuery already employs a fair amount of formal theory — XML Schema could benefit from formal foundations similarly.

4.2   Formal Languages

The theory of formal languages has its roots in different scientific fields as diverse as set algebra, linguistics, theoretical informatics, coding theory, and circuit design. In its most general definition, a formal language L is a set of *strings* chosen from a set Σ^*, the set of all strings[1] over a finite alphabet Σ.[2] Such an $L \subseteq \Sigma^*$ is said to be a *language over* Σ.

The *empty language* is commonly denoted by $\emptyset$, and the *empty string* or *nullstring* by ϵ. Note the difference between $\emptyset$ and ϵ: the former is a set (i.e., a language), the latter a string. The difference becomes obvious when looking at the cardinalities: $|\emptyset| = 0$, whereas $|\{\epsilon\}| = 1$. In consequence, *no* string is part of the empty language $L_\emptyset := \emptyset$, while $L_\epsilon := \{\epsilon\}$ contains exactly one string, i.e., the empty string. Both languages can be defined over an arbitrary alphabet.

A commonly used classification of formal languages is known as the *Chomsky* or *Chomsky-Schützenberger hierarchy*, which is shown in Table 4.1.[3] Although additional hierarchical tiers can be identified and distinguished, the concept of of such a hierarchy remains unchanged. The idea is to separate classes of languages by comparing the expressiveness of the corresponding grammars that produce or recognize the respective language. Each grammar, and hence its extent, (i.e., the language produced or recognized by the gram-

---

[1]More precisely, Σ^* is the union of the empty string and the transitive closure of the powers of an alphabet Σ, i.e., $\Sigma^* = \{\epsilon\} \cup \Sigma^+$. Put differently, $\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i$, where $\Sigma^0 = \{\epsilon\}$ and $\Sigma^k = \Sigma^{k-1} \times \Sigma$.

[2]In the following, alphabets and sets of nonterminals, terminals, variables, and symbols are always implicitly defined to be *finite*.

[3]In the original article [29], Chomsky uses a different terminology; he identifies no explicit computational models for grammar types 0 and 1, and the proper-subset relationships between the grammar types were proved by others.

mar), in the hierarchy is a *proper superset* of the next lower tier. Consequently, the hierarchy can be used for an objective *classification* of languages.

Type	Grammar	Model of Computation	Chomsky's Terminology
0	Unrestricted	Turing machine	Natural language (implied)
1	Context-sensitive	Non-deterministic linear-bounded Turing machine	Transformational grammar
2	Context-free	Non-deterministic push-down automaton	Phrase-structure grammar
3	Regular	Finite state automaton	Finite-state grammar

Table 4.1: Chomsky hierarchy

This classification is related to the theory of *complexity* and *computability*, which goes back to Post, Church, and Turing. Each class of languages corresponds to a *model of computation*. The model of computation of each hierarchy tier is strictly more powerful in terms of its computability than the next lower tier.

This leads to important results for applications. On the one hand, languages from higher tiers are effectively more powerful in terms of their *expressiveness*. On the other hand, they require more complex models of computation. For instance, context-free grammars can describe correctly parenthesized expressions, whereas no regular language is capable of describing this class of languages. However, the model of computation of context-free grammars requires a pushdown-stack, while regular languages are equivalent to plain finite state automata.

4.2.1 Regular Expressions

Definition: Regular expressions define exactly the set of regular languages. The regular language described by a regular expression E is denoted $L(E)$.

The three operations defined for regular expressions are *concatenation*, *union*, and *closure*. The respective operator symbols are $\cdot$ or comma (often omitted), $+$ (or a vertical bar), and *Kleene star* $*$. Often, the additional Kleene operators ? and $^+$ are added, but they can be expressed by the three basic operators: $E? = (E + \epsilon)$, and $E^+ = (E \cdot E^*)$.[4] Although the operator precedence is well-defined, parentheses are usually inserted for legibility.

The class of regular expressions is usually defined recursively as the set of expressions that can be generated by applying only these two rules (modified after McNaughton and Yamada [64]):

1. All symbols are regular expressions, and so are ϵ and $\emptyset$.

---

[4]Note that the reverse holds as well: E^* can be rewritten as $(E^+ + \epsilon)$. We will use this in Section 8.2.

2. If F and G are regular expressions, then $(F \cdot G)$, $(F + G)$, and (F^*) are regular expressions as well

Regular languages are closed under the above three regular operations, and under intersection and complement. Thus, regular languages form a boolean algebra over a given alphabet.

Example: A regular expression $E = ((a \cdot b?) + c^*)$ produces the language $L(E) = \{\epsilon, a, ab, c, cc, ccc, \dots\}$.

Model of Computation: There exist three different models of computation which all have their applications: *deterministic finite automata*, *non-deterministic finite automata*, and *epsilon-free non-deterministic automata*. However, it can be shown that all three models, and regular expressions *per se*, have equivalent computational power.[5] The proof is done by cyclic reduction, that is, it is shown pairwise that the set of languages describable by one model is completely contained in the set of describable languages of another. Due to the transitivity of the inclusion relation, all models of computation must be equivalent to each other, and to regular expressions. Finite state automata are thus said to be the *natural representation* of regular expressions.

Limitations: A famous property of regular languages is the so-called *pumping lemma for regular languages*. It can be used in order to prove that certain languages are not regular. The lemma states:

> For each regular language L, there exists a constant n such that for every string xyz which fulfills the following conditions
>
> 1. $xyz \in L$
> 2. $|xyz| \geq n$
> 3. $|yz| \leq n$
> 4. $y \neq \epsilon$
>
> the string $xy^k z$ also is in L for every $k \geq 0$.

This can be understood quite intuitively if we consider the corresponding finite state automaton. Let n be ($\#_{states} + 1$). Then, due to the *pigeon hole principle*, the automaton must, upon producing a string with length $\geq n$, visit at least one state more than once. This is, the automaton contains at least one cyclic transition. Let y be the symbol connected to this transition. As the automaton has no possibility to limit the number of visits of this transition, it also produces the set of strings $xy^k z$, as defined above. (This last step is sometimes summarized as "finite automata can't count".)

---

[5]This does not necessarily mean that their *computing power* is equivalent as well. In fact, non-determinism, which can be thought of as arbitrary parallelism, exponentially increases computing power (the downside being the problem of realizing non-deterministic automata).

An interesting example of a non-regular language is the language of *parenthesized expressions* (sc. *correctly* parenthesized expressions) of arbitrary length. This can easily be proven by applying the pumping lemma. Assume a string w in $L_{parenth}$ with $|w| \geq n$ and let β, β' be the parenthesis symbols. Then, due to the pumping lemma, an n can be found such that a decomposition $x\beta y\beta' z$ exists for w, such that $x\beta^k y\beta' z \in L$, which is a contradiction.

This example makes obvious that regular expressions are not capable of describing well-formed XML documents, since XML is precisely one of these kinds of parenthesized languages of the above form. On the other hand, the example provides the motivation to investigate another class of languages, the context-free languages presented in the next section.

Applications: Typical examples of applications of regular languages include *lexical analyzers*. Lexical analyzers use finite state automata for recognizing *tokens* (e.g., keywords) in compilers. Vice versa, regular expressions are typically used for prescribing the structure of tokens (e.g., legal variable names). XML Schema uses regular expressions as well, as the following excerpt from the XML Schema for Schema exemplifies:

```
<xs:simpleType name="NCName" id="NCName">
 <xs:restriction base="xs:Name">
  <xs:pattern value="[\i-[:]][\c-[:]]*" id="NCName.pattern"/>
 </xs:restriction>
</xs:simpleType>
```

Another example from the universe of XML is the content models of DTDs, which are regular expressions. However, general content models in SGML are not strict regular expressions, because they allow the use of the interleave operator &. Regular expressions extended by this operator sometimes are called *extended regular expressions*. The interleave operator causes several problems. Firstly, extended regular expressions are no longer *local*. This increases the complexity of such expressions, as we will see in Section 8.2. Secondly, different interpretations of the semantics of the interleave operator are possible. It is a priori unclear whether an expression $((ab) \& (xy))$ only allows the strings $\{abxy, xyab\}$, or also $\{axby, xaby\}$.

Although XML Schema knows a comparable construct, the restrictions of XML Schema's `xs:all` group avoid the semantic problems because such a group must not contain other model groups, and because it allows at most single occurrences of each element.

4.2.2 Context-Free Grammars

Context-free grammars, established by Chomsky in the aforementioned article [29], were initially intended to provide a description of natural language. Despite missing this goal, context-free grammars proved very powerful in the field of computer science.

Informally, context-free grammars can be viewed as an extension of regular expressions into the recursive domain.

Definition: Formally, a context-free grammar, or *CFG G*, is defined as $G = (V, T, P, S)$, where

1. V a set of *non-terminals*, or *variables* (in linguistics, these are called *syntactic categories*)
2. T a set of *terminals*
3. P a set of *productions* of the form $v \to x$ with $v \in V, x \in \{V \times T\}$
4. S a *start symbol* with $P \in V$

The context-free language generated by G is $L(G) = \{w \in T^* \mid S \xrightarrow{*} w\}$ with $\xrightarrow{*}$ being the transitive and reflexive closure of the above rewriting operator $\to$. That is, $L(G)$ is comprised of all those strings w that can be legally derived from the start symbol S by recursively applying the rewriting rules of the productions P.

Context-free languages are closed under the regular operations (i.e., concatenation, union, and Kleene star), but contrary to regular languages, CFGs are not closed under intersection and complement.

There is a tree representation of the derivation steps called *parse trees* which is used very often and proves highly useful. In a parse tree, the inner nodes (i.e., nodes with out-degree ≥ 1) are labeled with nonterminals $\in V$, the root node (i.e., the one with in-degree $= 0$) is labeled by the start symbol S, and the leaf nodes (i.e., the ones with out-degree $= 0$) are labeled with terminals $\in T$. The concatenation of the terminals, read from left to right, is called the *yield* of the parse tree, and it is exactly the string w produced by the derivation.

If more than one parse tree can be determined for a string, the grammar is called *ambiguous*. Often, ambiguity of a grammar is due to a poor design of the production rules, and can be avoided by transforming the grammar. But there is a class of *inherently ambiguous* context-free languages for which no non-ambiguous grammar exists. Furthermore, it is undecidable whether a given context-free grammar is ambiguous or not.

A commonly used notation for context-free grammars is the *Backus-Naur Form* (BNF), or *Extended BNF* (EBNF). We will see below that the `<!ELEMENT ...>` element in DTDs also is a notation for CFGs; and complex types in XML Schema can be seen as the right-hand sides of production rules in a CFG.

Example   A context-free grammar $G = (V, T, P, S)$ with
1. $V = \{P\}$
2. $T = \{(,)\}$
3. $P = \{P \to (P),\ P \to P \cdot P,\ P \to \epsilon\}$
4. $S = P$

produces the context-free language of parenthesized strings $L(G) = \{\epsilon, (), (()), ()(), (()()), \ldots\}$ The figure on the right hand displays the parse tree for the string $(()())$.
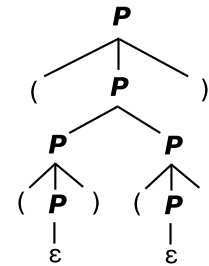
Model of Computation: The model of computation of context-free grammars and languages is the *push-down automaton* (PDA) or *stack automaton*. It adds a stack to the finite state automaton. It can be shown that PDAs are strictly more powerful than finite automata.

Like for finite state automata, there are *deterministic* and *non-deterministic* push-down automata as well. But in contrast to finite automata, these two classes are not equivalent for push-down automata. In consequence, the corresponding classes of grammars (i.e., deterministic and non-deterministic context-free grammars) are not equivalent either. For this reason, the Chomsky hierarchy in Table 4.1 is sometimes refined by inserting an additional tier between type-3 and type-2 grammars: the class of deterministic context-free grammars. It can be shown that the expressiveness of this class is strictly between regular languages and non-deterministic context-free languages, and the set of languages which can be described by deterministic CFGs is a proper subset of the set of languages of the former.

A simple example shall illustrate the difference. Consider the language of *palindromes* (of even length, for the sake of simplicity: $L_{pal} = \{ww^r \mid w \in T\}$), and the language of *palindromes with center-marker* ($L_{cmp} = \{w\#w^r \mid w \in \{T \setminus \#\}\}$). It is intuitively clear how PDAs recognize strings of these languages. For every input symbol they consume, a corresponding stack symbol is pushed onto the stack. As soon as half of the string is consumed, stack symbols are popped from the stack and compared to the input symbols consumed. If they are all equal, the input string is recognized.

The important question is of course, how does the PDA know when it is supposed to switch from pushing to popping mode? And here, the difference in computational power becomes evident. Deterministic PDAs are perfectly able to recognize L_{cmp}, the palindromes with center-marker, because the center-marker triggers the PDA to enter the popping mode. But as they do not have prophetic capabilities, they fail to recognize L_{pal}, for nothing indicates that the center of the string has been reached.

Non-deterministic PDAs recognize both languages. Recognizing L_{cmp} is trivial, and the recognition of L_{pal} is intuitively understandable when recalling the interpretation of non-determinism as parallelism. Imagine an additional instance of the PDA in popping mode to be forked for every input symbol consumed. If one of those forked instances succeeds in recognizing the input string, then the string is in L_{pal}.

The non-equivalence of deterministic and non-deterministic context-free grammars is important in the context of XML, as both non-deterministic and deterministic grammars are available.

Limitations: There is a *pumping lemma* for context-free languages as well. Its basic idea is similar, and it is used for the same purpose as the pumping lemma for regular languages, i.e., in order to prove that certain languages are not context-free languages. While the proof of the pumping lemma for regular languages is carried out by induction over the length of strings, the proof for the pumping lemma for CFLs is by induction over the depth of parse trees. The idea is to show that if the depth of the parse tree is $> |V|$, at least on nonterminal has productions that (possibly indirectly) contain this nonterminal, which has effects similar to the effect of cycles in finite state automata. Detailed proof can be found in the literature.

The pumping lemma for context-free languages states:

> For each context-free language L, there exists a constant n such that for every string $vwxyz$ which fulfills the following conditions
>
> 1. $vwxyz \in L$
> 2. $|vwxyz| \geq n$
> 3. $|wxy| \leq n$
> 4. $w, y \neq \epsilon$
>
> the string $vw^k xy^k z$ also is in L for every $k \geq 0$.

As a consequence, languages can be proven not to be context-free, for example $L_3 = \{a^k b^k c^k \mid k \geq 0\}$ or $L_{ww} = \{ww \mid w \in T\}$. The latter one is of interest in our context because it contains a kind of co-constraint that we might imagine to be applied to a document type as well. Bearing in mind the model of computation (i.e., the stack automaton), it is intuitively clear that this language cannot be recognized, and thus cannot be described by a context-free grammar. Assume the PDA to have consumed the first half of a word in L_{ww} and to have put corresponding symbols onto the stack. If the PDA now tries to compare the following input symbols to the symbols on the stack, it cannot do so. The first symbol of the second half of the input string would have to be compared to the symbol lying at the bottom of the stack, which cannot be accessed.

If we allow access of the stack at random positions, the model of computation becomes equivalent to a linear-bounded Turing machine. From Table 4.1, we see that L_{ww} therefore is part of the class of *context-sensitive languages*.

Applications: The main application of context-free grammars is *parsers*. This is the field where context-free grammars have been extremely influential for the last four

decades, and in turn, the popularity of context-free grammars is due to the impact they have had on the construction of parsers and compilers.

Different classes of parsers can be distinguished, which are of different computational power, i.e., they can only be employed in order to parse a certain subclass of context-free languages. These subclasses are usually named after the parser type. The most prominent examples are:

- *LL(k)* parsers read the input from <u>L</u>eft to right, build a <u>L</u>eft-most derivation, and use <u>k</u> symbols of look-ahead. "Building the left-most derivation" means that the parser works in a *top-down* manner. Because the worst-case complexity of parse tables for LL(k) parsers is exponential in k, usually only LL(1) parsers are of interest. This in turn limits the set of parseable languages to non-ambiguous languages without left-recursion (i.e., production rules must not contain a nonterminal both in its left-hand and right-hand side or derivatives therefrom).

- *LR(k)* parsers read the input from <u>L</u>eft to right, build a <u>R</u>ight-most derivation, and use <u>k</u> symbols of look-ahead. "Building the right-most derivation" means that the parser works in a *bottom-up* manner. LR(k) parsers are more powerful than LL(k) parser; each LL(k) grammar is a LR(k) grammar as well.

- *SLR* parsers, or <u>Simple</u> <u>LR</u> parsers, are a subclass of LR(k) parsers. They are LR(0) parsers with one symbol of look-ahead added, and they have restricted rules for how they manage their stack in order to avoid some problems encountered with LR(0) parsers.

- *LALR*(1) parsers are <u>L</u>ook-<u>A</u>head LR parsers. Although less powerful than LR(1) parsers, most of today's parser generators (e.g., *yacc*) produce this kind of parsers.

XML parsers are top-down parsers, and due to the requirement that content models must be deterministic, XML parsers work with one-symbol lookahead. This lets us conclude that XML parser are in the category of LL(1) parsers. Although XML grammars may contain left-recursion, other restrictions like determinism in DTDs and XML Schemas compensate for this. Brüggemann-Klein state that all SGML parsers (which are a more general class than XML parsers) are LL(1) parsers [23].

Extended Context-Free Grammars: A special case of context-free grammars is called *extended CFGs*. It implies that the right-hand side of the productions can be a regular expression. This is merely a change in notation and does not affect the effective computational power, because every extended CFG can be rewritten as an ordinary CFG. For instance, the production $P \to (E + F)$ can be rewritten as $P \to E$, $P \to F$.

DTDs are extended context-free grammars. The description of the content models of elements allows the usage of more than the three basic regular operators, but we have shown in Section 4.2.1 how these operators can be reduced to the three regular operators.

Let us briefly demonstrate how DTDs can be mapped to the above definition of context-free grammars as $L = (V, T, P, S)$:

- The set of nonterminals V is the set of element declarations with non-empty content model; i.e., element declarations of the form `<!ELEMENT` *name* `(...)>`.

- The set of terminals is the union of all empty element declarations and the special symbol `#PCDATA`. The latter is of special interest, as it is actually the nonterminal of a production rule implied by the specification [18]. Its right-hand side is the set of legal characters in XML.

- The production rules P are the element declarations. The second production rule from our introductory above would read in a DTD as `<!ELEMENT P (P,P)>`.

- Finally, the start symbol S is defined in the *document type declaration*. Using the above example again, this is `<!DOCTYPE P [` *the element declarations* `]>`.

Looking at the interpretation of DTDs as extended context-free grammars, one realizes that many of the essential features of DTD, like attributes, IDREFS, and parameter entities, are not present. In fact, the practical expressiveness — more pragmatically understood as usefulness — is not covered by the interpretation as context-free grammar. However, this does not mean that DTDs are more expressive than CFGs in the formal sense. They just include a set of features which do not change the expressiveness in the formal sense of "which classes of languages are they able to describe", but which essentially add to the practical usefulness of DTDs as an actual schema language for XML. This discrepancy is even more pronounced in the case of XML Schema.

This has two consequences: Firstly, when we want to compare schemas in terms of their expressiveness in the strict sense, or when we want to make statements about computational complexity, we might want to refer to a *grammar core* of a schema language. For simple, grammar-oriented schema languages like RELAX NG, this grammar core may embrace almost the whole language. For more complex modeling-oriented languages like XML Schema, this may be a small subset.

Secondly, several attempts have been made to describe schema languages *entirely in a formal way*. Section 4.3 offers an overview in the case of XML Schema, and Section 4.5 discusses whether this is a reasonable way of addressing the above discrepancy, or whether other possibilities are conceivable.

4.2.3 Hedge Grammars

Hedge grammars have been introduced by Murata [71] as "a simple but powerful model for XML schemata". Hedge grammars build upon *regular tree languages*, which was first studied systematically by Brainerd [14] as a generalization of regular languages over strings. Murata is one of the two authors of the XML schema language RELAX NG (see Section 2.4.2), which is defined as a hedge grammar.

Hedge grammars are not more powerful than context-free languages (in fact, they can be shown to be context-free), but hedge grammars are especially suited for describing XML grammars because they reflect the structures of XML in a semantically intuitive way.[6]

Definition: According to [71], a *regular hedge grammar* (RHG) is defined as $G = (\Sigma, X, N, P, r_f)$ whith

1. Σ a set of *symbols*
2. X a set of *variables*
3. N a set of *nonterminals*
4. P a set of *production rules* of either the form $n \to x$ (where $n \in N, x \in X$) or $n \to a\langle r \rangle$ (where $n \in N, a \in \Sigma$ and r a regular expression comprising nonterminals).

When interpreted as *sequences of trees*, symbols identify interior tree nodes, where variables label leaf-nodes.

Example: A hedge grammar $G = (V, T, P, S)$ with

1. $\Sigma = \{a\}$
2. $X = \{x\}$
3. $N = \{n_1, n_2\}$
4. $P = \{n_1 \to a\langle (n_1 + n_2)^+ \rangle, \ n_2 \to x\}$
5. $n_r = (n_1 + \epsilon)$

produces the context-free language of parenthesized strings $L(G) = \{\epsilon, a\langle x \rangle, a\langle xa\langle x \rangle\rangle, a\langle xa\langle x \rangle x \rangle, a\langle xxa\langle xa\langle x \rangle xx \rangle x \rangle, \ldots\}$ The figure on the right hand displays the tree interpretation of the expression $a\langle xa\langle a\langle xxx \rangle\rangle a\langle xx \rangle\rangle$.
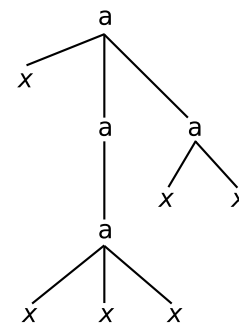
Note that the interior nodes are labeled by *symbols*, rather than nonterminals as in the parse trees of CFLs, for the structure we see here is the *yield*, or outcome of the derivation.

Tree Interpretations: An *interpretation* of a tree (or hedge) is a mapping that maps every node in a tree to a nonterminal. Interpretations can be used in order to define when a tree is generated by a given tree grammar: a tree is generated by a tree grammar if an interpretation against this grammar exists, i.e., if every node of the tree can be mapped to a nonterminal of this grammar.

We mentioned before that tree grammars are context-free grammars. The interpretation of a tree thus is the parse tree of the context-free grammar which generates this tree.[7]

---

[6]In particular, *ordered* sequences of nodes and *text nodes* (called variables).

[7]In fact, the set of parse trees of context-free languages form a regular tree language.

Note however that more than one interpretation of a tree may exist (see the discussion of ambiguity of context-free grammars in Section 4.2.2).

It is evident that interpretations can be thought of as *type annotations*. XML schema languages therefore in general have restricted grammars in order to prevent ambiguous interpretations. The most restricted class of XML grammars, *local tree grammars*, of which DTD is an example, require one-to-one correspondence between nonterminals and element names (i.e., between the n and the a in hedge productions $n \rightarrow a\langle r \rangle$). This restriction obviously makes the interpretation trivial. The limited expressiveness, on the other hand, is a disadvantage. Section 4.2.4 presents a taxonomy of XML grammars and discusses their respective limitations.

Model of Computation: Together with the notion of hedge grammars, Murata introduces deterministic and non-deterministic hedge automata. Yet the operation of hedge automata is no longer as easily imaginable as the operation of push-down automata. We refer to [71] for a detailed description. Furthermore, due to the correspondence to *Dyck strings*, one can conclude that push-down automata are equivalent in computational power to hedge automata, and that push-down automata can be used as a model of computation as well.

Dyck Strings: Berstel and Boasson [8] use *Dyck strings* for describing XML languages. Dyck strings are a special subclass of *balanced context-free languages*. Without going into detail of Dyck strings and their properties, we note the principal idea that T, the alphabet of terminals of Dyck strings, is composed by two complementary alphabets (i.e., $T = A \cup \overline{A}$), and that the language is generated by a grammar $X \rightarrow aX^*\overline{a}$.[8]

Obviously, the strings produced by such a grammar can be used for describing well-formed XML documents, and the grammar thus is an abstract XML grammar. While hedges seem to reflect the hierarchical structure of XML documents very well, Dyck strings describe their *serialization*. If we think of it in analogy to the common processing models of XML, we might say that hedges are similar to the DOM model, whereas Dyck strings resemble the SAX model.

We know that the two well-known processing models are equivalent in power — although of varying appropriateness in use. And in fact, Brüggemann-Klein and Wood [22] prove that Dyck languages (and balanced context-free languages in general) and hedge languages are equivalent. However, hedge grammars can be considered to be representing XML grammars on a higher level of abstraction, preserving the semantics of the hierarchical relationships. Brüggemann-Klein and Wood call the hedge the *generic-derivation hedge*, and its corresponding Dyck string the hedge's *sequential form*.

---

[8]More precisely, this is the language of Dyck primes. Dyck strings additionally allow concatenation of Dyck strings, but the formula gives the basic idea.

From this equivalence we conclude that the limitations of regular hedge grammars are the same as for context-free grammars. In particular, the language L_{ww} is still not describable. A consequence for XML grammars is that it is not possible to express constraints like "subsequent paragraphs must have the same inner structure, no matter *how* this (arbitrary) structure is".

To make this clear, assume the following DTD:

```
<!ELEMENT doc (para+)>
<!ELEMENT para ((h1 | h2 | h3), #PCDATA)>
<!ELEMENT h1 (#PCDATA)> ...
```

Then we cannot use the grammar in order to express the constraint "if one `para` starts with `h2`, then every `para` must start with `h2`." This is exactly a constraint of the form encountered in L_{ww}. As we discussed in the example of Section 4.2.2, there are languages capable of expressing such constraints, the *context-sensitive languages*.

Applications: Today, the most important application of hedge grammars is in the schema language RELAX NG, both as a formal foundation of the specification and for the implementation of RELAX NG parsers.

4.2.4 Formal Categorization of Document Grammars

One important application of formal descriptions is the *categorization*, or the *classification*, of grammars and languages. Once a formal description is at hand, we can compare languages and grammars using objective criteria.

Murata, Lee, and Mani provide such a "Taxonomy of XML Languages Using Formal Language Theory" [72]. They identify subclasses of *regular tree languages*, discuss the allowed operations, and compare these classes in terms of expressiveness. Furthermore, they study different algorithms for the validation of document structures for each class, and they analyze existing XML schema languages using their taxonomy.

Regular tree grammars (RTG) are more general and abstract than regular hedge grammars, and their relation to context-free grammars is the same. Formally, regular tree grammars are $G = (V, T, P, S)$ with the variables defined as above. The productions of regular tree grammars are of the form $X \rightarrow ar$ with $X \in V$, $a \in T$, and r a regular expressions over nonterminals $\in V$. The right-hand side of productions is called the *content model*.

Murata's taxonomy is summarized in Table 4.2, which needs further explanation. Firstly, note that the hierarchy exhibits the same property as the Chomsky hierarchy, i.e., each class in the table is strictly more powerful than the class in the next lower row. In consequence, the languages generated by the respective grammars are proper supersets of each other. Secondly, the column labeled "Closed" indicates under which operations

the respective languages are closed. The symbols are used in their usual sense. Finally, the third column requires the definition of *competing nonterminals*.

> *Definition:* Two nonterminals are *competing* if two nonterminals share the same terminal a in their respective productions.

Put differently, if two nonterminals are *not* competing, a direct correspondence exists between terminal a and the nonterminal X. Assuming the productions to be in a reduced form, where each nonterminal only appears once in the left-hand side of productions,[9] a direct correspondence between terminals a and content models r can be established.

Class	Closed	Restriction of Competing	Example
Regular Tree	$\cup \cap \setminus$	Unrestricted	
Restrained-Competition	$\cap$	No content model produces competing nonterminals with a common prefix of nonterminals	RELAX Core
Single-Type Tree	$\cap$	No content model contains competing nonterminals	XML Schema
Local Tree	$\cap$	There are no competing nonterminals	DTD

Table 4.2: Taxonomy of XML grammars according to Murata [72]

However, the expressiveness of the grammar part of a schema language does not necessarily describe a language's actual usefulness, which usually depends on engineering requirements as much as on formal properties.

In the particular case of XML Schema, Bex et al. [9] argue that the *element declaration consistent* (EDC) and *unique particle attribution* (UPA) constraints are too restrictive, and that they do not meet their intended goal. They propose to replace EDC and UPA by a more precise yet less restrictive constraint, which they call *one-pass preorder typability* (1-PPT). The idea is to prohibit only those cases where type annotation for a certain instance node cannot be done during a post-order (or *depth-first*) parsing of the instance. This is very similar to the notion of restrained-competition grammars, although the definition is slightly different. Both approaches follow the principle of excluding unfavorable cases by formal rules, rather than applying various constraints in an *ad-hoc* manner, which eventually results in a too restrictive class of grammars.

---

[9]This is always possible because productions contain regular expressions. Multiple productions can be rewritten as union of the respective content models.

4.3 XML-Specific Formalisms

As mentioned above, a "Formal Description" for XML Schema [19] was attempted, but abandoned later. RELAX NG [31] defines formal semantics, as described in Section 2.4.2. DTDs are a subset of SGML document grammars, which has been subject of various formal studies. The reader may want to refer to Brüggemann-Klein [20].

Thompson [90] proposes to define a dedicated *logic*, i.e., a logical framework, for describing XML Schema. In a working paper,[10] Sperberg-McQueen outlines how *definitieve-clause grammars* might be used for a description of XML Schema. Both approaches are explicitly Schema-related, and it is arguable whether a formal description should describe XML Schema with all its features *at once*. A limitation to the grammar-related parts of XML Schema, or a layered approach similar to the approach chosen in RELAX NG, is perhaps more realistic and reasonable.

Another motivation for formally describing XML documents and XML grammars is the goal of developing a *conceptual modeling formalism* for XML. A general discussion of conceptual modeling can be found in Hoppenbrouwers [52]. An overview of conceptual modeling formalisms for XML can be found in Nečaský [74] and in Mohan and Sengupta [69, 86]. Sengupta and Wilde [87] provide an overview along with a set of requirements.

Most of the formalisms proposed for XML are extensions or adaptations of well-known modeling formalisms, either the *Unified Modeling Language* (UML) or the *Entity Relationship Model* introduced by Chen [26].

In spite of an abundance of proposals, there is no formalism for conceptual modeling established at present. There are various reasons for this. Since XML is capable of representing a wide range of different data structures, the required models are considerably more complex than in the case of the more limited relational data structures. Furthermore, the trade-off between minute correspondence to underlying schema languages versus simplicity and power of the modeling formalism, i.e., the choice of the level of *granularity*, is crucial. Some formalisms fail because of an overly simplistic view, which omits too many of a schema language's essential features, other formalisms are hardly useable in practice because of their complexity.

Proposals include *Heterogeneous Nested Structures* (HNS) [86] and the *eXtensible Entity Relationship model* (XER) [85] by Sengupta, *XGrammar* [63] and *EReX* [62], which builds upon XGrammar, by Mani, *ERX* [84] by Psaila, and *Conceptual XML* (C-XML) [38] by Embley et al.

XGrammar provides a formal grammar that attempts to adequately capture XML Schemas. XGrammar assumes the existence of a set of nonterminal names $\widehat{N}$, terminal names $\widehat{T}$, and atomic simple types $\hat{\tau}$. An XGrammar G is then defined as the *7-tuple* $G =$

---

[10]Available at http://www.w3.org/People/cmsmcq/2004/podcg.html.

$(N_T, N_H, T, S, E, H, A)$ with

1. $N_T \subseteq \widehat{N}$ the set of nonterminals of tree type
2. $N_H \subseteq \widehat{N}$ the set of nonterminals of hedge type
3. $T \subseteq \widehat{T}$ the set of terminals
4. $S \subseteq N_T \cup N_H$ the set of start symbols
5. E the set of element productions $X \rightarrow aRE \mid a \in T, X \in N_T$
6. H the set of hedge productions $X \rightarrow RE \mid X \in N_H$
7. A the set of attribute productions $X \rightarrow aRE \mid a \in T, X \in N_T \cup N_H$

RE then is further defined, and the attribute production rule is refined in order to comply with the ID/IDREF concept. One immediately sees that such a grammar can describe XML Schema quite precisely, but at the cost of a lengthier and less concise grammar notation. A formal grammar notation of XML Schema can prove very useful nevertheless. The presence of a formal grammar facilitates specification, implementation, and reasoning about XML grammars.

Finally, the separation made between an formal XML grammar (XGrammar) and conceptual modeling formalism built upon the formal grammar (EReX) is more likely to be successful in dealing with the problems of granularity and complexity than monolithic approaches.

4.4 Interesting Properties

Since formal languages have been studied extensively over the last decades, a wide range of properties has been investigated. We discuss a small fraction of those properties that appear to be useful in our context. Some of them are directly applicable, others may serve as an inspiration for new algorithms or new perspectives and presentations of XML schema languages.

4.4.1 Marked Expressions

McNaughton and Yamada [64] first made symbols of a regular expression distinct by adding indices to the symbols. In a *marked expression*, all symbols are unique. The marking is usually expressed by adding subscripts to the symbols. For instance, the regular expression $E = (c(a+b)^*a)$ becomes $E' = (c_1(a_1+b_1)^*a_2)$ when being marked.

For marked regular expressions, a deterministic automaton can always be derived. However, *unmarking* the symbols makes the automaton nondeterministic in the general case. Brüggemann-Klein [20, 21] utilizes this in order to concisely define *unambiguous* SGML content models. According to Brüggemann-Klein, the one-symbol-lookahead version of unambiguity in the SGML specification should be called more precisely *1-unambiguity*.

Let us first introduce the complementary operation of marking, i.e., the *unmarking* of marked expressions. The operation simply discards the subscripts, and it is sometimes written as a function $\chi(x)$, sometimes as an operator $x^\natural$. We use the latter notation. The definition of 1-unambiguous content models then is:

> A regular expression E is *1-unambiguous* if its marked expression E' denotes no two strings uxv and uyw where $x \neq y$ and $x^\natural = y^\natural$.

If we resume the example from above, we can now prove that the regular expression $E = (c(a + b)^*a)$ is not 1-unambiguous. We consider the marked expression $E' = (c_1(a_1 + b_1)^*a_2)$ and two input strings $c_1\mathbf{a_1}a_2$ and $c_1\mathbf{a_2}$ (the marked symbols which correspond to the variables x and y, respectively, are printed in boldface). We recognize that the marked symbols are distinct, but after unmarking the strings, the symbols are equal: the first string becomes caa, the second becomes ca. Therefore, we have $a_1 \neq a_2$ yet $a_1^\natural = a_2^\natural = a$, and we conclude that E is not 1-unambiguous.

One of the applications of marked expressions thus is checking of ambiguity, and providing the formal foundation for the definition of 1-unambiguity, which is important and helpful. In Section 8.2, we will show how marked expressions can be used for an efficient construction of automata, and how this can be utilized for XML Schema applications.

In the same section, we also extend marked expressions to symbols with *numeric exponents*, i.e., symbols for which the range of permissible occurrences is given by integers in $0, \infty$. However, the definition of 1-unambiguity then must be adapted, too, for instance by using marked symbols from $\Sigma \times \mathbb{N}^2$, rather than from $\Sigma \times \mathbb{N}$, and by adapting the $\natural$ operator as well.

4.4.2 Derivatives

Brzozowski [24] introduced *derivatives* of regular expressions in order to develop an elegant algorithm for constructing finite automata from regular expressions. A derivative of a regular expression with respect to a certain marked symbol is the regular expression which describes the set of strings that can follow this symbol. Given a marked regular expression $E = ((a_1 + b_1)c_1a_2^*)$, the derivation with respect to a_1 is $D_{a_1}(E) = (c_1a_2^*)$. Consecutive derivations $D_{c_1}(D_{a_1}(E))$ can be rewritten $D_{a_1c_1}(E)$.

For the formal definition of derivatives, Brzozowski introduces a function $\delta(E)$, which sometimes is said to test whether a regular expression is *nullable* or *transparent*.

$$\delta(E) = \begin{cases} \epsilon & \text{if } \epsilon \in E \\ \emptyset & \text{if } \epsilon \notin E \end{cases}$$

Although a regular expression may have infinitely many derivations, the number of *characteristic derivations* (i.e., the set of derivatives modulo associativity, commutativity,

and idempotence of the + operator) is always finite. Therefore, every regular expression can be decomposed into a sum of disjoint terms and δ:

$$E = \delta(E) + \sum_{a_i \in \Sigma} a_i D_{a_i}(E)$$

From this decomposition it is clear that an automaton can directly be constructed, where the symbols a_i denote the arcs (i.e., the input symbols which cause the transition) and the derivatives D_{a_i} the vertices (i.e., the states of the automaton).

It is not hard to see that this decomposition of regular expressions using derivatives can be applied in order to test whether or not a given sequence of nodes subsumes a content model in an XML grammar. Sperberg-McQueen discusses the "applications of Brzozowkski derivatives to XML Schema processing" [89].

4.4.3 Follow Sets

Berry and Sethi [7] introduced the notion of *follow sets* of positions (i.e., marked symbols). Follow sets contain all positions that can legally follow after a given position in a string. A special set is the *first set*, which contains all positions with which a string can be legally started. The definitions are as follows:

$$first(E) = \{a \mid av \in L(E)\}, \qquad follow(E, a) = \{b \mid uabv \in L(E)\}$$

Additionally, either a last set $last(E) = \{b \mid ab \in L(E)\}$ or a special *endmarker* symbol (usually denoted by !) is required. For this report, we choose the latter alternative, and thus always implicitly add ! to Σ where follow sets are used.

Obviously, follow sets are closely related to derivatives of regular expressions. Follow sets can be employed for similar purposes (most notably, for the construction of deterministic finite automata from regular expressions), while yielding more efficient algorithms. Section 8.2.2 demonstrates how follow sets can be utilized for checking content models in XML grammars.

A second application of follow sets is that they permit concisely defining when two positions are *competing*. The notion of competing positions is used for defining ambiguity, and in decision algorithms for ambiguity. Brüggemann-Klein [20] defines competition of two positions in regular expression using follow sets:

Two positions x and y in a marked expression E' *compete* if and only if

1. $x, y \in first(E)$ or

2. $x, y \in follow(E, z)$ for some $z \in E$

Brüggemann-Klein [20, 21] also gives a definition of competing positions for *extended expressions*, i.e., expressions which contain the interleave operator & as well. The definition uses an adapted version follow sets, $follow^-$, which has been proposed by Clark. However, this workaround defines those adapted follow sets recursively for expressions $E = F$ & G. Neumann [73] argues that this is not justifiable because the & operator is not associative. He then gives a different ambiguity test for SGML content models.

4.5 Canonicalization and Normalization

Canonicalization and normalization are important for different reasons. Both can be used in order to make languages, grammars, or single instances *comparable*. In the context of document grammars, canonicalization is usually a prerequisite for making document grammars amenable to a formal description, and normalization can be used in order to guarantee certain quality criteria.

We distinguish canonicalization from normalization. Normalization is a stronger concept, which also includes structural transformations, and which is connected to a *measure* (or to some *quality criteria*) that is used for defining a kind of *minimal* or *best* form. Canonicalization primarily defines how certain degrees of freedom shall be limited, it defines rules for the unification of encodings, and it defines which properties should be discarded in a canonical form.

A common example of canonicalization in grammars is the omitting of unused symbols, nonterminals, or production rules. For instance, Berstel and Boasson [8] call a grammar *reduced* if "every non-terminal is accessible from the axiom, and every non-terminal produces at least one terminal word." The well-known example of canonicalization of XML documents is the Infoset, and in Section 6.3, we discuss a possible approach to obtain canonicalization for XML Schemas as well.

A normalization of extended context-free languages has been proposed by Albert et al. [1]. There are also proposals for normal forms for XML, for instance by Embley and Mok [39] and by Arenas and Libkin [2]. Unfortunately, both chose the name *XML Normal Form* (XNF) for their proposals, although they are not related.

Normalization and canonicalization in XML Schemas have not been discussed extensively yet. We list a few reductions that normalization of XML Schemas — in contrast to canonicalization — could include:

Unification of Attributes and Elements: In its simplest interpretation, attribute declarations are mapped to element declarations, such that the documents described by the grammar all are in the so-called *Element Normal Form* (ENF). For instance, Sengupta's XER [85] assumes documents to be in ENF.

Resolution of Substitution Groups: From a semantic point of view, the effect of *substitution groups* and *choice model groups* is equivalent. One might therefore

consider to transform substitution groups into choice model groups.

Restructuring Model Groups: For regular expressions, normal forms can be defined via the corresponding minimal automata. As model groups are extended regular expressions, normalization can be applied to them.

While it is debatable to which extent normalization is desirable for XML Schemas, canonicalization is an important prerequisite for formal descriptions of XML Schemas. Both mechanisms can be used in order to separate different *layers*, eventually singling out a purely syntactical subset, or *grammar core*, of XML Schema. One example of such a layering can be found in RELAX NG, where a well-defined *simplification* of a convenient (but complex) syntax to a simple (but formally describable) one takes place.

However, XML Schema is significantly more complex than RELAX NG, because it contains modeling-oriented features as well. It will be advantageous to gradually strip off these more abstract properties, eventually applying normalization transformations like the ones listed above. A layered approach that achieves a separation of concerns could substantially alleviate a later formal description of XML Schema. We believe that a combination of separation of concerns and a simple yet robust grammar formalism is a more promising approach than the approaches listed in Section 4.3 which attempt to formally describe XML Schema with all its features as a whole.

Chapter 5

Data Model Access

In Section 3, we stated the need for an accessible data model for XML Schema. We pointed out that *accessibility* should include three parts:

1. a *unified* data model,
2. a coherent concept of identity with the possibility to uniquely identify the components of the data model, and
3. a set of well-defined accessor functions.

However, we want to clarify that XML Schema information is accessible already at present, and in various ways. The Schema-aware additions in XSLT 2.0, which we described in Section 2.4.1, are one example; other examples include the APIs of validating parsers, most notably the *Xerces Native Interface* (XNI) of the Xerces parser family. Yet these kinds of accessibility are not sufficient to satisfy our above requirements. XSLT 2.0 does not expose the *structure* of an XML Schema (e.g., in a navigable way), and XNI uses a proprietary representation of the data model, and the interface is only accessible on a low level of programming which requires an amount of expertise which we think is inappropriate.

The problems are mostly due to the way in which XML and XML Schema evolved. XML progressed from an untyped, but reasonably self-contained, data format to a typed format which potentially requires the Schema to be present in order to interpret the data contained. This development required adaptations and additions to be made. For instance, the Infoset became augmented by the PSVI,[1] and the Infoset became refined (or reduced) to the XPath data model when XSLT entered the scene.

As XML technologies become more mature, and as the use of XML is increasingly shifted from being a simple interchange data format towards becoming a central data format for

---

[1]Unfortunately, the Infoset was not very well prepared to integrate such additions. Despite vivid discussions which demanded the incorporation of extension facilities into the Infoset, the final specification contained none of these.

global-scale information systems, we expect Schema-awareness to become an essential feature. In order to fully utilize Schema data, its data model must be designed to be accessible.

It is not our claim to have the solution to the question how such an accessible data model should look in the end. We rather present two approaches which we are experimenting with. We think that the presence of prototype implementations is essential for identifying the most important scenarios of use, for exploring problems and opportunities, and in order to finally design a framework which meets the three goals stated above.

5.1   Applications

There are a great many of applications which potentially will utilize, and benefit from, accessible XML Schema information. The kinds of these applications can be very different, which results in manifold demands and needs. This complicates the design of an accessible data model. There is no single optimal solution. The design of an accessible data model must be guided by the class of applications it is intended to be utilized by. Many of the needs of different use cases are not easily reconciled, and a great amount of flexibility (or arbitrariness) is inherent to many of the design decisions to be taken. It is therefore even more important to have prototype implementations available, in order to experimentally explore these flexibilities or trade-offs.

The space of different classes of applications which will presumably utilize Schema data can be roughly divided into two dimensions:

1. Applications which require *read-write* access to the Schema, and applications for which *read-only* access is sufficient.

2. Applications which work with Schemas on a *stand-alone* basis, and applications which use Schemas for processing *instance documents*.

In the first of these two dimensions, we constrain our studies to *read-only* scenarios. This is in accordance with the corresponding models and technologies for XML documents, which also focus on read-only access (i.e., the Infoset and XPath). The second dimension is further illustrated by examples in the following sections. The following chapters then present our two approaches, *SCX* (Chapter 6) and *SPath* (Chapter 7).

5.1.1   Stand-Alone Applications

By *stand-alone* applications we mean applications that work on the XML Schema as primary input. Examples of applications from this class include:

UI Generation: Because XML Schema describes the syntactical structure of a class of documents (and potentially a subset of the semantics, too), the generation of user

interfaces from Schemas is desirable. Garvey and French propose generation of user interfaces from composite schemas [44]. Many of the challenges they encountered are related to data model accessibility and presentation.

Schema Mapping: Schema mapping, i.e., the identification of structural (and possibly semantic) similarities, and the generation of a mapping between the Schemas (i.e., by generating an XSLT stylesheet), is an important process transferring XML documents in loosely coupled systems. Schema-mapping frameworks like Cupid [60] could be substantially improved by the availability of more complex Schema information.

The recent popularity of *mash-ups* adds to the importance of mapping applications, because re-purposing of information often requires structural transformations.

Documentation: XML Schemas are not suitable as a documentation format. In Section 8.3, we demonstrate how documentation can be generated from XML Schemas in a versatile and extensible fashion. Documentation of XML Schema will be crucial in the near future, as XML Schema is increasingly used as a format for describing service interfaces, for example in WSDL (see Section 2.4.3).

5.1.2 Instance-Driven Applications

By *instance-driven* applications we mean applications that work on XML instances as primary input, utilizing XML Schema information while processing. Generally, examples of this class of applications primarily include *Schema-aware transformations*, for example:

Resilient processing: Utilizing Schema information, applications can be designed to be more resilient and tolerant (e.g., in the face of unknown versions or extensions). This may include *type reflection* (i.e., the run-time generation of stylesheets from a template meta-stylesheet).[2]

Compatibility Preprocessing: Through the ability of *type introspection*, input data may be preprocessed in order to be compliant with a processing application. A particular case of compatibility preprocessing is *Schema-aware validation by projection*, as described in Section 3.4.

Data Mining: Recalling our reflection of XML Schema as a metadata format in Section 3.3, Schema information may be utilized for interpreting instance data.

---

[2]At present, we expect this class of application to become the most important scenario of use for data model accessibility.

Chapter 6

SCX: Schema Component XML Syntax

In contrast to DTDs, the transfer syntax for XML Schema is not very convenient for human readers to deal with. This is mostly because of the verbosity which is introduced by the XML syntax. From this one might be tempted to conclude that the transfer syntax is better suited for machine-processing, but on closer inspection it becomes apparent that this is not the case either. While the XML syntax enables machines to easily parse XML documents, it does not facilitate the assembling of the Schema's *abstract data model* from the (possibly multiple) Schema *documents*.

One of the reasons is that XML Schema documents are not fully self-contained. The right-hand side of Figure 6.1 on Page 65 illustrates how an XML Schema document depends on various external data sources. Firstly, it may *include* further Schema documents (not shown in the figure), and it may *import* Schema documents with a different namespace (the Schema document `personLib.xsd` in the figure). Secondly, every XML Schema document implicitly imports all *built-in types* from the recommendation. And thirdly, every Schema obeys additional rules and constraints which are defined by the recommendation. While *includes* and *imports* are usually present as XML documents, the other contributions are typically built into the processor or Schema-validator. An application which desires to retrieve the actual (i.e., assembled) Schema must have this information available as well. This is not easy, because the *XML Schema for Schemas* covers only a fraction of this information, whereas the rest is only available in prose — inaccessible to machines.

Another reason why the retrieval of the Schema information from Schema documents is hard is the misalignments between the elements of the transfer syntax and the units of the abstract data model, which are the *Schema components*. Consider for instance the *complex type definition* Schema component. From the recommendation, we see that this component has the following properties:

1. {name} (optional)
2. {target namespace} (possibly absent)
3. {base type definition}
4. {derivation method}
5. {final}
6. {abstract}
7. {attribute uses}
8. {attribute wildcard} (optional)
9. {content type}
10. {prohibited substitutions}
11. {annotations}

Property 1 is easy to determine. Property 2 has to be looked up from the document's top-level `xs:schema` element. 5 and 10 have to be computed from document-wide defaults and local definitions, whereas 3, 4, and 6 have to be computed from local definitions and default values obtained by interpreting the recommendation. 11 requires the collection of all `xs:appinfo` and `xs:documentation` elements and of all attributes with non-Schema namespaces from the type definition element and all of its dependents. 7 and 9 are only determinable by resolving all references to named groups and by investigating the complete set of ancestor types — always correctly computing the resulting effective model group, which depends on the type of derivation. The construction of the Schema property 9 is particularly intricate because the recommendation requires it to be "one of empty, a simple type definition or a pair consisting of a content model [...] and one of mixed, element-only." The {simple type definition} of course is yet another property to be calculated. And finally, the computation of 8 involves the resolution of all attribute groups, consideration of all ancestor types, and thorough study of the recommendation.

Obviously, the transfer syntax is *not* a good representation of Schema information for machine-readable access either. It turns out that the transfer syntax is a hybrid of a machine-readable format (i.e., XML) and a human-editable syntax (i.e., notational shorthands like document-wide default values).

In Section 2.1, we quoted an excerpt from the recommendation [91] of XML Schema that stated:

> The abstract model for schemas is conceptual only, and does not mandate any particular implementation or representation of this information.

This inspired us to design an alternative XML syntax for XML Schemas with explicit objectives and clear design rationales. We call our XML syntax the *Schema Component XML Syntax* (SCX) [101].

6.1 Design Rationale

The design of SCX is guided by clear objectives and the following general strategy.

1. The syntax represents the *Schema components*, from which the abstract data model is built, as faithfully as possible.

2. The format is *self-contained* as far as it is possible in an XML format.

3. The syntax is explicitly intended to be read by *machines*.

Note that the third principle implies that SCX is explicitly *not* intended for human use. XQuery also makes a clear split between human-readable and machine-readable formats. The standard syntax is a non-XML syntax, but an XML syntax called *XQueryX* [65] is available as well. The former syntax suits humans, the latter is a powerful format for machine access, e.g., when generating or transforming XQuery documents [53]. In analogy to XQuery, one might want to complete this spectrum by defining a non-XML syntax for XML Schema as well, which is primarily intended for human use. A compact syntax has been proposed [103], but has not received wide recognition so far.

6.1.1 Recommendation Version

Version 1.0 is still the authoritative recommendation for XML Schema, and SCX mostly implements this recommendation. However, there are parts in version 1.0 of the recommendation which are either unclear or which require decisions to be made when implementing a representation of the data model. The draft recommendation of version 1.1 of XML Schema (which is currently being developed) is often clearer in such cases. We thus decided to follow XML Schema 1.1 wherever it clarifies version 1.0. SCX is not a complete implementation of XML Schema 1.1, although it can easily be extended to cover the entire recommendation of XML Schema 1.1 (this, of course, depends on whether additional major changes will be included in future draft versions).

SCX is able to represent every XML Schema document which is correct with respect to the XML Schema 1.0 recommendation.

6.1.2 Mapping to XML

The design rationale while developing the XML syntax of SCX was to map Schema components and their properties directly to XML elements wherever possible, and to add additional properties if necessary. This was necessary in four cases, partly because of insufficiencies or vagueness of the recommendation, partly in order to keep the implementation efficient, and in order to support the needs of users.

1. SCX introduces *unique identifiers* (UID). On the one hand, this avoids the problems of component identifiability discussed in Section 2.6.3, on the other hand, it

provides for an efficient implementation. UIDs are only used internally and can be hidden from the user completely.

2. The problem of non-discernable anonymous simple types in unions (see Section 2.6.3) is addressed through the introduction of an additional component property {position}, which is only present for the problematic simple types.

3. SCX can be configured to additionally preserve information concerning the provenance of the Schema components, i.e., the document URI of the XML document where the component was defined, and an XPath expression which points to the corresponding definition element within this document.

4. While the transfer syntax allows annotations to appear in different locations,[1] Schema components gather annotations in one {annotations} property. Since the exact association is often a semantically relevant part of the annotation information,[2] SCX stores a relative XPath which points to the original location of the annotation.

In order to circumvent *choices* in the data model itself, the abstract data model often employs a component property {variety}, which then decides how other component properties are to be interpreted. In the *simple type definition* component for instance, the property {variety} decides which of {primitive type definition}, {item type definition}, and {member type definitions} must be present or absent. It is debatable whether or not this is a good approach. Yet we decided to stay consistent with the abstract data model in most of these cases, with one exception. XML Schema 1.1 specifies *wildcards* to have a property {variety} which is "One of {*any, enumeration, not*}", and which then determines the semantics of the {namespaces} property. If {variety} is "enumeration", {namespaces} contains a list of *allowed* namespaces. If {variety} is "not", {namespaces} contains a list of *excluded* namespaces. In our opinion, the design of the abstract data model should not contain component properties with variable semantics. SCX's representation of wildcards therefore employs component properties {any}, {allowed-namespaces}, and {excluded-namespaces} instead.

Finally, SCX always uses XML container elements in lists, even if the list items have an atomic type. Although lists of atomic values could be represented by whitespace-separated lists, we decided to express list structures through XML markup.[3]

Note that these two deviations from a strict mapping of Schema components (i.e., avoiding {variety} in wildcards and introducing container elements for list items) are pure syntax design decisions and not structural characteristics of SCX.

---

[1] For example, `xs:annotation` in `xs:complexType` elements may appear within `xs:simpleContent` or `xs:extension` et cetera as well.

[2] This is particularly true for non-Schema namespace attributes, which are handled as {annotations} as well.

[3] As XML is a format for structured data, we see no need for mixing it with different, less robust formats for structured data like whitespace-separated lists.

6.1.3 Self-Contained Data Format

One of the key advantages for machine access is that SCX documents contain as much Schema information as XML Schema is capable of representing. SCX therefore includes all information that normally would be external or implicit into one document. Of course, the downside of this approach is a significant increase in document size. However, the advantage of not being forced to collect, compute, and interpret information from various (possibly inaccessible) places vastly outweighs this disadvantage.

Figure 6.1 shows the process of assembling an SCX document from all the different sources that usually make up an XML Schema.
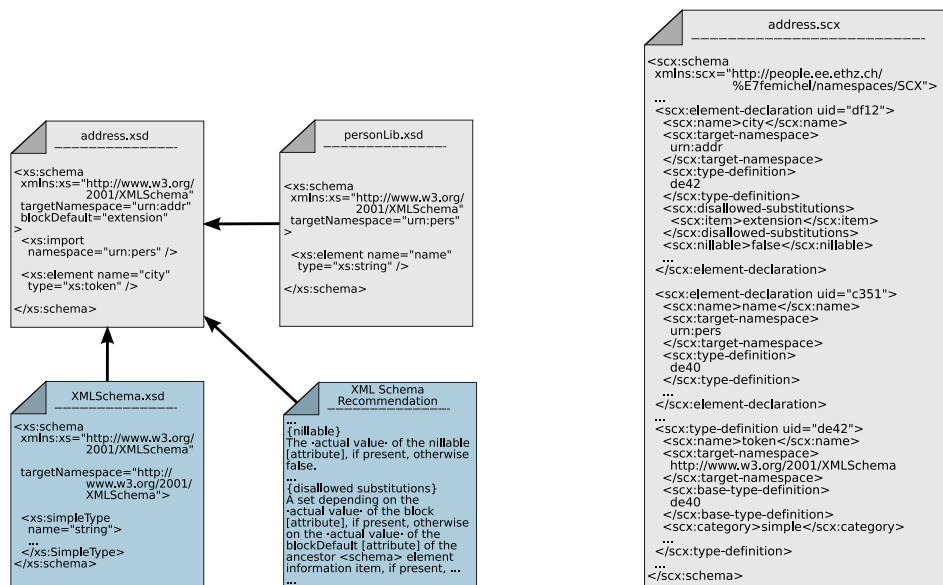
Assembling a Schema from different documents   Self-contained representation in SCX

Figure 6.1: Self-contained representation of an assembled Schema through SCX

The chosen example illustrates different characteristics of the transformation to SCX. The document `personLib.xsd` is an example of an XML Schema document which is imported into `address.xsd`. These two Schema documents have different namespaces. The sample SCX document on the left-hand side illustrates how SCX incorporates components from different namespaces by means of its `scx:target-namespace` element.

The element declaration `city` in `address.xsd` references `xs:token`, an XML Schema built-in type. SCX uses the XML Schema for Schemas document *XMLSchema.xsd* in order to retrieve the type definition of `xs:token`. The sample SCX document shows how the built-in types are then represented exactly like any other (user-defined) type definition. Additionally, the use of SCX's internal unique identifiers can be observed. The property `scx:type-definition` of the topmost element declaration references the

built-in type `xs:token` through the `@uid` of the latter.

Finally, the properties `scx:nillable` and `scx:disallowed-substitutions` demonstrate how definitions given in prose in the recommendation (as shown in the left-hand side figure) are interpreted and applied in SCX.

6.2 Format

As SCX is the *XML syntax* for Schema components, the format is normatively described by an XML Schema document — which of course is also available as an SCX document. This XML Schema for SCX is not only helpful for checking the structure of SCX documents, but it can also be used for checking some of the constraints of (assembled) XML Schemas. The XML Schema for Schemas neither contains any identity constraints for ensuring uniqueness of qualified names for elements and types et cetera, nor does it check the consistency of references. The simple reason for that is that XML Schema cannot express identity constraints which span multiple documents. Because SCX comprises all necessary component in one document, the XML Schema for SCX is able to check many of the constraints which the XML Schema for Schemas cannot check.

Thus, SCX can also be used for basic *Schema-checking*. However, several constraints still cannot be expressed by a grammar-oriented schema like XML Schema. Legal derivation of types, consistent declaration of element types, the unique particle attribution, and similar constraints still require procedural Schema-checking. SCX at least offers a convenient starting point for doing this. Using SCX, it is possible to implement Schema-checking in XSLT.

In addition to the aforementioned global strategy, the design of the XML format follows these principles:

Make the XPaths Nice: The design decisions are governed by the goal to allow easy access paths to be written. This sometimes results in more verbose syntax constructs. But remembering that SCX is designed for machine access, this is less important than the possibility to write powerful XPaths. For example, model groups — even inherited or referenced ones — are always expanded, as we will see in Section 6.3. This has the advantage that an XPath expression

```
$type//scx:content-model//scx:particle
```

readily returns a flattened set of all possible particles within the content model of a given type.[4]

Anticipation of the form of XPath expressions that will work on SCX also influences the next principle:

---

[4]This facilitates the implementation of SPath's `contain::` axis described in Section 7.

Use Elements for Components: SCX always maps Schema components to elements, although many components have only atomic values. In turn, all the additional information described in Section 6.1.2 is encoded as attributes. One reason for doing so is consistency in the representation of Schema components, the other reason is hiding (or encapsulation) of internal information.

Given such a separation, a typical XPath expression which uses only the child axis selects all component information, but none of the additional information, which is meant for internal use anyway. If an application has to access the internal information as well (e.g., the function library introduced in Section 8.1), it can easily do so by using the attribute axis.

Had we decided to use attributes for components with atomic content, the necessary XPath expressions (e.g., for selecting all properties of all type definition Schema components) would look as clumsy as this:

```
//scx:type-definitions/scx:* |
 //scx:type-definition/@*[local-name() ne 'uid']
 [local-name() ne 'document'][local-name() ne 'path'][...]
```

compared to the elegant and robust way it can be done now:

```
//scx:type-definition/scx:*
```

Consistent Naming: Because SCX aims at representing Schema components as faithfully as possible, it employs the names of the respective Schema components, except that spaces are replaced by hyphens. This consistent naming also eases the understanding of SCX documents for the human reader, especially the one familiar with the recommendation.

6.3   Canonicalization

Transformation of Schema documents in the transfer syntax into an SCX document leads to a certain amount of *canonicalization* of the Schema. The term "canonicalization" is used in the same sense as in Section 4.5. That is, the goal is not to transform the Schema into a (however defined) minimal form, but into a standard representation which ensures certain properties. The canonical form of XML instances is defined by the Infoset. The Infoset specifies which properties of an XML document (i.e., the serialization of an XML tree) are relevant, which ones are only present in the serialization, and which degrees of freedom are only fixed in the serialization (e.g., the order of attributes). The reader might want to refer to Appendix D of the Infoset recommendation [34].

And just like *canonical XML* is defined as the serialization of the Infoset, *back-transformation* of an SCX into Schema documents in the conventional transfer syntax may accomplish the process of canonicalization for Schemas. In fact, the function library which sup-

ports dealing with SCX provides functionality for back-transformation, or *serialization*, of SCX documents. The function library is described in Section 8.1.

The extent to which canonicalization should be performed has no fixed boundary and is subject to design decisions. The decisions made for SCX are no unchangeable prerequisite for the format. The trade-offs chosen can be discussed and adjusted. This is a common characteristic of the definition of canonical forms; the Infoset also is the result of lengthy and intense debates.

In the following, we summarize the canonicalizational aspects of SCX.

1. SCX always explicitly inserts the default values from the recommendation for properties which are absent in the transfer syntax (e.g., `use="optional"` for all `xs:attribute` elements).

2. SCX re-delegates all document-wide declarations of defaults back to the respective components. In consequence, an XML Schema document obtained by serializing an SCX document never contains attributes like `finalDefault` in its top-level `xs:schema` element; instead, every declaration or definition element carries a `final` attribute.

3. All qualified names are present in their expanded form. No namespace prefixes are used in SCX, and the mappings from namespace URIs to prefixes are not preserved.[5]

4. Prohibited attributes are removed. While one could argue that this is already a kind of *normalization*, it appears that this removal is no arbitrary decision of SCX, but part of the recommendation, which states:[6]

   > [*The Schema component corresponding to an <attribute> element information item*] corresponds to an attribute use with properties as follows (unless use='prohibited', in which case the item corresponds to nothing at all): [...]

   Serialization of an SCX documents correctly inserts the missing declarations, if type restriction calls for this, yet annotations attached to such declarations are lost during round-trips.

5. Analogously, particles with trivial occurrence (i.e., with `minOccurrs = maxOccurrs` = 0 or `minOccurrs > maxOccurrs`) are removed. Like the neglecting of prohibited attributes, this is justified by the recommendation:

   > [*For an <element> element information item,*] the corresponding schema component is as follows (unless minOccurs=maxOccurs=0, in which case the item corresponds to no component at all): [...]

---

[5]For convenience of use, the serializer reads a mapping file where namespace bindings can be defined.
[6]Parts in oblique font are condensed by the author from parts scattered in the recommendation.

As in the case above, the missing declarations are inserted during the serialization of an SCX documents, where necessary.

6. All inherited parts, both parts of the content model and inherited attributes (or attribute wildcards), are *expanded* in SCX. After back-transformation, only the necessary declarations are part of the resulting Schema documents. For instance, if in a Schema document a type derived by restriction repeats the declaration of an attribute (which is inherited anyway) without restricting, this declaration is no longer present after round-tripping.

7. The same holds true for constraining facets in simple type definitions. Definitions of pointless facets are not preserved.

8. Named groups and named attribute groups are also expanded in SCX's syntax.[7] The order of attribute declarations within named attribute groups is not preserved.

9. When serializing the namespace constraints in wildcards and attribute wildcards, the effective constraint is output. It is no longer discernible whether a namespace constraint has simply been inherited or has been explicitly repeated in the original Schema document.

Canonicalization is an important achievement. When conformance with *design rules* or *best practice guidelines* is to be checked, it is either a prerequisite that the Schema documents are present in a canonical form, or the design rules must take the possible degrees of freedom into account. The presence of a canonical representation of XML Schema thus facilitates this process.

---

[7]However, SCX keeps the information which is necessary in order to recover all named groups and attribute groups in serializations.

6.4   Sample SCX Source

Two brief excerpts illustrate the Schema component XML syntax. Figure 6.3 is an excerpt from the SCX representation of the *International Purchase Order* Schema (IPO), which is the sample Schema used in the W3C's *Primer* for XML Schema [40]. Figure 6.4 is an excerpt from the XML Schema for SCX.

```
<simpleType name="Postcode">
  <restriction base="string">
    <length value="7" fixed="true"/>
  </restriction>
</simpleType>

<simpleType name="UKPostcode">
  <restriction base="ipo:Postcode">
    <pattern value="[A-Z]{2}\d\s\d[A-Z]{2}"/>
  </restriction>
</simpleType>
```

Figure 6.2: Excerpt from the SCX representation of IPO.xsd

Figure 6.2 displays two simple types in the transfer syntax, and Figure 6.3 presents their SCX representation.[8] It is obvious from this piece of sample code that SCX is not suitable for human readers.

---

[8]SCX represents both simple and complex types by the same component, which has a property {category} for disambiguation of the two. This simplification seems justifiable due to the strong structural resemblance and the fact that both share the same namespace partition.

Note how the SCX representation of the derived type (i.e., `UKPostcode`) contains all effective constraining facets. The attribute `inherited` indicates the base type from which the respective constraining facet has been inherited. The property `scx:primitive-type-definition` points to the primitive built-in type, which in this case is the type definition with the unique identifier `d14e769`, which appears to be `xs:string`.

```
<scx:type-definition uid="d11e70">
  ...
  <scx:name>Postcode</scx:name>
  <scx:target-namespace>http://www.example.com/IPO</scx:target-namespace>
  ...
</scx:type-definition>

<scx:type-definition uid="d11e78" document="file:/.../primer-example/address.xsd"
                     path="schema[1]/simpleType[3]" position="28">
  <scx:annotations/>
  <scx:name>UKPostcode</scx:name>
  <scx:target-namespace>http://www.example.com/IPO</scx:target-namespace>
  <scx:base-type-definition>d11e70</scx:base-type-definition>
  <scx:category>simple</scx:category>
  <scx:final/>
  <scx:facets applicable="length minLength maxLength pattern enumeration whiteSpace">
    <scx:whiteSpace inherited="d11e70">
      <scx:annotations>
        <scx:annotation ownerName="whiteSpace">
          <scx:attributes id="string.preserve"/>
        </scx:annotation>
      </scx:annotations>
      <scx:value>preserve</scx:value>
      <scx:fixed>false</scx:fixed>
    </scx:whiteSpace>
    <scx:length inherited="d11e70">
      <scx:annotations/>
      <scx:value>7</scx:value>
      <scx:fixed>true</scx:fixed>
    </scx:length>
    <scx:pattern>
      <scx:annotations/>
      <scx:value>[A-Z]{2}\d\s\d[A-Z]{2}</scx:value>
    </scx:pattern>
  </scx:facets>
  <scx:fundamental-facets>
    ...
  </scx:fundamental-facets>
  <scx:variety>atomic</scx:variety>
  <scx:primitive-type-definition>d14e769</scx:primitive-type-definition>
</scx:type-definition>
```

Figure 6.3: Excerpt from the SCX representation of IPO.xsd

Figure 6.4 shows excerpts from the XML Schema for SCX. Using the example of type definitions, it demonstrates how the XML Schema for SCX is capable of asserting more advanced constraints than the XML Schema for Schema.

```
<xs:element name="type-definitions">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="type-definition" minOccurs="0" maxOccurs="unbounded"
                  type="scx:typeDefinitionType"/>
    </xs:sequence>
  </xs:complexType>
  <xs:unique name="typeNameUnique">
    <xs:annotation><xs:documentation>
        The OR-ing of scx:name and scx:content is safe, because utilization of
        them is mutually exclusive.
    </xs:documentation></xs:annotation>
    <xs:selector xpath="scx:type-definition"/>
    <xs:field xpath="scx:name | scx:context"/>
    <xs:field xpath="scx:target-namespace"/>
    <xs:field xpath="scx:context/@position"/>
  </xs:unique>
</xs:element>
...
<xs:complexType name="nameableComponentType" abstract="true">
  <xs:complexContent>
    <xs:extension base="scx:identifiableComponentType">
      <xs:sequence>
        <xs:choice>
          <xs:element name="name" type="xs:NCName"/>
          <xs:element name="context">
            <xs:complexType><xs:simpleContent><xs:extension base="scx:uidType">
              <xs:attribute name="position" type="xs:positiveInteger"/>
            </xs:extension></xs:simpleContent></xs:complexType>
          </xs:element>
        </xs:choice>
        <xs:element name="target-namespace" type="scx:xmlnsType" minOccurs="0"/>
      </xs:sequence>
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
...
<xs:complexType name="typeDefinitionType">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#dc-defn"/>
    <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#dc-defn"/> ...
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="scx:nameableComponentType"> ...
```

Figure 6.4: Excerpt from the XML Schema for SCX

Chapter 7

SPath: A Path Language for XML Schema

In Section 3, we stressed the analogy between data model issues on the level of XML and the level of XML Schema. For XML, the *Infoset* is the established data model which defines the canonical form of XML. We then discussed how canonicalization of XML Schema can be achieved, using the example of SCX. However, the most important tool for working with XML is not the Infoset, but the path language which operates on top of it:[1] the *XML Path Language*, or XPath for short. XPath has proved extremely convenient for navigating XML data, and it is used by many other technologies, e.g., XSLT, XQuery, DOM, and XML Schema.

In contrast to its predecessor, XPath 2.0 no longer works only on the Infoset. XPath 2.0 operates on a *type-annotated* tree instead (Section 2.3.1 described the process of type-annotating XML documents). But XPath 2.0 exposes the type information only in a very limited and shallow way. Types are merely represented by qualified names, and neither is it possible to inspect the different properties of a given type, nor is it possible to navigate the type hierarchy or other components of the Schema.

In collaboration with Wilde, we created *SPath* [99, 100], the path language for XML Schema. Although a stand-alone path language for XML Schema is conceivable, SPath builds on (and thus complements) XPath. SPath extends both the data model and the syntax constructs of XPath, and it adds new functions to XPath. This has obvious advantages: On the one hand, SPath does not have to re-invent the wheel, but uses XPath's well-known and widely approved language design instead. On the other hand, the two path languages can be seamlessly integrated. This is especially useful for the scenarios described in Section 5.1.2.

In the following, we outline the design principles and the main characteristics of SPath.

---

[1]More precisely, XPath 1.0 and 2.0 operate on the XPath data model and the XDM, respectively, both of them being based on the Infoset.

A detailed and comprehensive description of SPath can be found in the aforementioned publication [99].

7.1 Design Considerations

Designing a path language includes a substantial number of design decisions that require compromises and trade-offs between conflicting needs to be made. As usual with design decisions, there is more than one valid solution, and many of these decisions could have been made in a different way, without changing the overall concept of SPath. While making our design decisions, we applied consistent criteria that were guided by the anticipated application scenarios. We deemed the *instance-driven* scenarios more likely to become the important areas of application.

7.1.1 XPath-Conforming Syntax

The most fundamental design principle of SPath is to be consistent with the syntax of XPath. Although SPath adds new axes, functions, and kindtests to the syntax, it does not change the *principles* of the syntax of XPath. This is important if a future integration into XPath processors is desired. Compliance with XPath's syntax allows to adapt XPath parsers easily. The EBNF description of SPath in Appendix A shows that only one single definition of XPath must be adapted in order to incorporate SPath. The nonterminal `PathExpr` of the XPath grammar has to be adapted in order to allow SPath expressions alongside XPath expressions.

7.1.2 SPath Axes

The most powerful feature of XPath are *axes*. Axes not only allow to traverse XML data trough the use of *location paths*, axes also support the convenient collection of data. In a general sense, XPath axes can be defined as functions that return a set of nodes, starting from a given node (the *context node*), and applying one or more structural criteria. These criteria can either concern the *kind* of the nodes to select (e.g., whether it is an element, attribute, or namespace information item) or the structural relationship to the context node (e.g., whether it is a parent, child, or preceding sibling of the context node, or whether it appears before or after the context node in the document tree).

SPath builds on this general definition of "axes". This proves especially helpful for XML Schema, because the relationships between Schema components are more complex than the relationships among information items in XML instances. While the latter constitute trees, the former form a densely interconnected graph. The basic and intuitive navigational concepts of XPath like *parent-of* and *child-of* are only applicable to a fraction of XML Schema's data model, more precisely, the type hierarchy. But the

general concept of an axis is applicable to the entire data model of XML Schema very well.

Elements that would be hard to select if only basic navigational expressions were available become easily selectable when dedicated axes are at hand. The development of powerful *axes* thus was an essential part of SPath's design. It is a question of design which tasks should be supported by axes. We follow the principles of XPath and provide axes for those tasks which either are very frequent or particularly hard to accomplish.

7.1.3 Data Model Simplification

The design of the data model is a delicate task. Looking at the variety of components in the abstract data model of XML Schema, it suggests itself to simplify the data model for use in SPath. Although it is principally possible to introduce new node types for each Schema component, this would limit the conciseness and usefulness of SPath. We decided to only represent a subset of XML Schema's components, and to use a slightly adapted representation of model groups in order to preserve all the essential information nevertheless.

The next section explains the data model of SPath. The simplification of the data model is probably the most substantial and far-reaching modification, and thus presumably the most arguable as well.

7.2 Data Model

SPath adds five new node kinds to the data model of XPath: `type`, `declaration`, `constraint`, `occurrence`, and `schema`. `schema` nodes represent the Schema as a whole, and`constraint` nodes represent identity constraint definitions. `type` and `declaration` are simplifications of two closely related components respectively. `type` nodes represent both simple and complex type definitions; `declaration` nodes represent both attribute and element declarations. This simplification seems justifiable by the structural and semantical similarities. If an application needs to further distinguish these components, it can do so by using new kind tests.

The remaining simplification (i.e., the introduction of `occurrence` components) requires further explanation. Based on the idea of *derivatives* (Section 4.4.2), *marked expressions* (Section 4.4.1), and *follow sets* (Section 4.4.3), SPath represents content models in a different way than XML Schema usually does. The approach is guided by two goals: *Simplification* (of both complexity and number of node kinds) and *ease of use* in instance-driven scenarios.

The basic idea is to *flatten* the (possibly hierarchically nested) model groups of XML Schema, and to unify content models with attributes. This rules out the need for intro-

ducing new node kinds for model groups, particles, attribute uses, named groups, and wildcards. In order to preserve the complete content model information, we calculate the *follow set* for each occurrence, which can be accessed through the `followed-by::` axis. This closely resembles the construction of Glushkov automata from regular expressions as proposed by Brzozowski [24] and Berry and Sethi [7]. Section 8.2 describes the XSLT functions that back this representation in our prototype implementation.

As a part of the flattening, we also expand all numeric exponents, and in consequence, occurrences only need two properties *optional* and *unbounded* in order to express their cardinality. This simplification also allows us to design functions to test these properties that are semantically consistent.

However, while the occurrence-based data model is a convenient perspective when accessing the Schema from instances, it imposes problems when a minute representation of the original structures of the Schema is needed (e.g., in the scenarios described in Section 5.1.1). We emphasize that the choice of the data model, and possible simplifications, are a design decision which can be made in a different way, if necessary. We assume the instance-driven scenarios of use to become the more important ones, and this anticipation governed the design of SPath. Different objectives and target scenarios might lead to a different design of SPath and its data model.

7.3 Path Syntax

A full description of the syntax of SPath can be found in [99]. Here, we only give a brief overview of SPath's language elements.

7.3.1 Node Tests

Node tests in XPath can either be *name tests* or *kind tests*. SPath follows the same principle. As in XPath, name tests in SPath have the following form:

```
((prefix | wildcard) ':' (local-name | wildcard) | *)
```

Since the data model of XML Schema (and thus the data model of SPath) contains *unnamed* nodes, the semantics of the wildcard have been extended to select unnamed nodes as well.

XPath defines *kind tests* which cover all *node kinds* encountered in XPath. Likewise, SPath provides a set of *kind tests* for each of the five SPath node kinds. In accordance with XPath, the kind tests in SPath accept a variable number of arguments, which are used in order to narrow down the set of selected nodes. For instance, the kind test `type()` accepts a first argument that can either be a QName or a wildcard (`*`). (If the wildcard is specified, *anonymous* nodes are returned as well.) Furthermore, a second argument can be specified which is one of the following string constants: {`'simple'`, `'simple-atomic'`,

'simple-union', 'simple-list', 'complex'}. Obviously, this is required in order to distinguish simple types from complex types, one of the simplification of SPath's data model.

The syntax and semantics of *predicates* are exactly the same as in XPath. For each item in the sequence produced by the expression preceding the predicate list, each predicate is evaluated with this item as the context, and only if all predicates evaluate to true, the item remains in the final result sequence of the step. Hence, the full set of both XPath and SPath expressions can be used in predicates.

7.3.2 Functions

SPath follows the principle of defining functions only for information that is either a literal property (rather than a structural one) or where the function requires more or different arguments other than the context node. An example for the latter case is the function `constrains()`, which would be well suited for being expressed as an axis, but which cannot be expressed as an axis because it needs two input arguments. Examples for the former case are functions returning node properties like *name*, *namespace URI*, or *nillable*.

Since many functions like `name()` or `namespace-uri()` are semantically equivalent to the corresponding XPath functions, the respective XPath functions are extended to polymorphic functions accepting nodes from both *universes*.[2]

7.3.3 Axes

Table 7.1 summarizes the possible transitions in the combined universe: for each node kind, the table shows which node kinds are reachable using SPath's axes. The function `constrains()` has been included as well because its functionality is navigational, and only the fact that it needs two input arguments makes it impossible to design it as an axis. In the upper left corner, the traditional XPath axes are indicated by a gray-shaded background.

---

[2]By *universes* we refer to data model of instances and Schemas, respectively.

in \ out	element	attribute	declaration	type	constraint	occurrence	schema
element	parent:: child:: ancestor:: descendant:: preceding:: following:: self:: …	attribute::	declaration:: substituted-by:: substitutes::	type:: basetype:: derivedtype:: supertype:: subtype::	constraint::	occurrence:: followed-by:: preceded-by::	schema::
attribute	parent::	self::	declaration::	type:: basetype:: derivedtype:: supertype:: subtype::		occurrence::	schema::
declaration	instance::	instance::	substituted-by:: substitutes::	type:: basetype:: derivedtype:: supertype:: subtype::	constraint::	occurrence::	schema::
type	instance::	instance::	declaration::	substituted-by:: substitutes:: basetype:: derivedtype:: supertype:: subtype::		contains::	schema::
constraint	constrains($\cdot,\cdot$)		declaration::		refer:: referred-by::	constrains($\cdot,\cdot$)	schema::
occurrence	instance::	instance::	declaration:: substituted-by:: substitutes::	type:: basetype:: derivedtype:: supertype:: subtype::	constraint::	followed-by:: preceded-by::	schema::
schema			declaration::	type::	constraint::	occurrence::	

Table 7.1: The SPath axes, together with the constrains() function.

Chapter 8

Implementation

For both SCX and SPath, we emphasized that design decisions are always influenced by the expected scenarios of use. Often, the majority of design decisions are trade-offs, for which no single best solution exists. Instead, compromises must be found which reconcile the various reqirements, probably preferring the needs of those applications which are expected to become the most important ones. In order to identify the needs of applications, and also the opportunities of the technology, *prototyping* plays a crucial role.

The prototype implementations of the SCX and SPath are both implemented solely using standard XML technologies, namely XSLT 2.0. While this limits the possibilities and the performance in some parts, it ensures utmost flexibility in deployment, and thus testing of the prototypes. Other possibilities would have been to utilize an existing low-level API (e.g., the Xerces Native Interface), or to extend some XSLT processor. Both approaches would have resulted in a proprietary and less flexible prototype. With the approach chosen for SCX and SPath, the prototype is readily useable and requires no modifications to be made to processors.

For prototyping at this early stage, flexibility of deployment and rapid development outweigh performance and seamless integration. The desirable final integration (i.e., of SPath as a language extension of XPath) cannot be done by the developers anyway, this requires the work of a standardization committee.

8.1   SCF: Schema Component Function Library

The *Schema Component Function Library* (SCF) provides the functionality that is necessary in order to navigate the components of SCX, and it backs the language constructs (i.e., axes, kind tests, and functions) of the prototype implementation of SPath.

8.1.1   Component Navigation

Since the Schema components of XML Schema's abstract data model form a graph, many of the arcs have to be cut when representing the components using XML. Therefore, SCX elements often contain references by means of *unique identifiers*. For example, the {type definition} Schema property, which points to the type definition used in an element or attribute declaration, is represented in SCX as follows:

```
<scx:element-declaration uid="d11e45">
  <scx:annotations/>
  <scx:name>postcode</scx:name>
  <scx:target-namespace>http://www.example.com/IPO</scx:target-namespace>
  <scx:type-definition>d11e78</scx:type-definition>
  ...
```

The type definition which is referenced here is defined elsewhere as an element `scx:type-definition`, and can be seen in Figure 6.3 on Page 71. If we want to access the {type definition} property of the element declaration `postcode`, the following XPath expression merely returns the UID:

```
<xsl:variable name="elm" as="element(scx:element-declaration)"
              select="//scx:element-declaration[scx:name eq 'postcode']"/>
<xsl:sequence select="$elm/scx:type-definition"/>
saxon@work> d11e78
```

This is most likely not what the user expects, and the next XPath does not return what might be expected either. It returns the empty set, because `scx:type-definition` simply contains a UID, and not the actual type definition element, for the aforementioned reasons.

```
<xsl:sequence select="$elm/scx:type-definition/scx:name"/>
saxon@work>
```

This is where the function library becomes useful. SCF provides methods which resolve these references-by-UID. The functions bear the same names as the respective component property, in order to facilitate their use in XPath expressions.[1] Employing the correct function call, the XPath finally returns what the user expects:

```
<xsl:value-of select="$elm/scf:type-definition(.)/scx:name"/>
saxon@work> UKPostcode
```

In addition to these basic functions which resolve the references-by-UID in SCX, more advanced navigational functions are available. For instance, the function `scf:get-super-types` returns all ancestor type definitions of a given type definition. It is not hard to see that these functions back the corresponding axes in SPath. Using this function, the complete chain of parent types can easily be determined:[2]

---

[1]Currently, the functions reside in the SCF namespace, though. If this appears to be an issue in practice, the namespace can easily be changed.

[2]In this example, the type names are ordered by their appearance in the SCX docu-

```
<xsl:value-of select="$elm/scf:type-definition(.)/
                      scf:get-super-types-or-self(.)/scx:name"/>
saxon@work> Postcode UKPostcode string anyType anySimpleType
```

Furthermore, *comparison* operators for Schema components in their SCX representation are available, e.g., `scf:type-equal`, which compares two types by comparing their {name}, {target namespace}, and {context}, if the type is anonymous. Note that in general, *node comparison* can be used for comparing components, because SCX reflects component identities by node identities.

```
<xsl:if test="$elm/scf:type-definition(.) is
                      //scx:type-definition[scx:name eq 'UKPostcode']">
  <xsl:text>Yes, they're equal!</xsl:text>
</xsl:if>
saxon@work> Yes, they're equal!
```

8.1.2 Initialization

Special diligence has been taken in the design of the *initialization functions* for SCX. In order to accommodate the variety of conceivable ways in which the transformation of a Schema into an SCX document may occur, the initialization functionality has been design to be flexible. We assume that SPath requires XML Schemas to be imported in the same way as XSLT 2.0 does, i.e., by means of the `xs:import-schema` construct. From the recommendation of XSLT 2.0 we can see that `xs:import-schema` supports five different ways of how to specify the Schemas to be imported. The definition if the recommendation looks as follows:

```
<xsl:import-schema
  namespace? = uri-reference
  schema-location? = uri-reference>
  <!-- Content: xs:schema? -->
</xsl:import-schema>
```

Note that both attributes, `namespace` and `schema-location`, are optional, and that the element may contain an XML Schema *inlined*. The initialization functions support all these ways, because they are intended to back the `xs:import-schema` element in SPath. Moreover, they are designed for two additional cases. They accept an XML Schema as the input file of the XSLT stylesheet, and they accept XML tree fragments which contain XML Schemas. The latter case is important if, for instance, XML Schemas are embedded within WSDL documents; the former case is important if the application is a *standalone* application (as described in Section 5.1.1). For example, the transformation stylesheet which turns a set of XML Schema documents in the transfer syntax into an SCX document looks as simple as this (namespace declarations are omitted):

---

ment. This is due to XPath's evaluation of path steps. Rewriting the XPath to `for $i in scf:get-super-types-or-self($elm/scf:type-definition(.)) return $i/scx:name` yields the type names in the order of the type hierarchy, i.e., `UKPostcode Postcode string anySimpleType anyType`.

```
<xsl:stylesheet version="2.0">
  <xsl:import href="SClib.xsl"/>
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="/">
    <xsl:sequence select="scf:schema(scf:import-schema-explicitly(/))"/>
  </xsl:template>
</xsl:stylesheet>
```

An alternative initialization function `scf:import-schema-implicitly` takes an instance document and a namespace URI as parameters and tries to fetch a Schema document by looking for `xsi:schemaLocation` attributes in the instance. If neither namespace nor Schema location are specified, the function `scf:import-schema-heedlessly` attempts to collect every Schema it knows about in a *best-effort* fashion.

8.1.3 Instance-Based Schema Access

The function library also provides functionality for accessing Schema information from instances. This backs the universe-crossing axes in SPath. For instance, the `scf:get-type` backs the `type::` axis, i.e., it returns the type definition component for the actual type of an element in the instance. Utilization of this class of SCF functions requires a global variable `scf:schema` with type `document-node(element(scx:schema))` to be present.

Given the following instance document:

```
<ex:root
  xmlns:ex="urn:ex"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:ex example-1.xsd">
  <ex:base xsi:type="ex:extType" ex:globatt-a="3">
  ...
```

The function `scf:get-type` is able to determine the type definition component for the `ex:base` element:

```
<xsl:variable name="scf:schema" as="document-node(element(scx:schema))">
  <xsl:document>
    <xsl:sequence select="scf:schema(scf:import-schema-heedlessly(/))"/>
  </xsl:document>
</xsl:variable>

<xsl:template match="/">
  <xsl:variable name="ty" as="element(scx:type-definition)"
              select="ex:root/ex:base[1]/scf:get-type(.)"/>
  <xsl:value-of select="$ty/scx:name"/>
</xsl:template>
saxon@work> extType
```

A disadvantage of this approach (i.e., of the XSLT-based prototype implementation) is the impossibility to access the type-annotations of the PSVI. Instead, the respective type definition components have to be computed each time the accessor function is called. Due to recursion, the complexity of determining the type of an instance node is roughly $\mathcal{O}(S * N!)$, with $S = \#_{types} + \#_{declarations}$ in the Schema, and $N = \#_{ancestorNodes}$ of the instance node. This factorial complexity could be avoided if the PSVI was used. Using the PSVI, the accessor functions would not introduce any additional complexity, because type annotation would occur during the Schema-validation of the instance, and hence only once for each instance node.[3] Appendix B.1 lists the algorithms employed.

8.2 The Occurrence-Based Data Model

Based on SCX, an alternative representation of the content models of complex types has been implemented. In this representation, hierarchical model groups are transformed into flat sets of *occurrences*. Each occurrence has the following properties:

1. A boolean property *optional*.
2. A boolean property *unbounded*.
3. A *term*, which is a reference to either an element declaration component or to a wildcard component.
4. A *follow set* is associated with each occurrence. The follow set contains all occurrences that legally can follow the current occurrence in a Schema-valid instance.

Obviously, this is the same data model perspective as employed in SPath's data model, and indeed, this XSLT-based implementation of the occurrence-based data model backs SPath's data model. It is based on the idea of *derivatives*, *marked expressions*, and *follow sets*, which are described in Section 4.4.

The occurrence-based data model emphasizes another useful achievement of SCX. Once an XML representation and a set of functions is present, alternative data model representations can be implemented efficiently and in an easy way. Only the parts which have to be remodeled must be implemented, and the components can then be referenced through their unique identifiers. Hence, the `occurrence` components can be implemented in the light-weight manner described above. Alternative representations can be thought of as an representation *overlay*, which only rearranges the access structure without affecting the content of the Schema components.

8.2.1 Problems

McNaughton and Yamada [64] introduced *marked regular expressions*. While particle components in the conventional data model of XML Schema correspond to symbols in

---

[3]For local grammars (e.g., DTDs), which are the most restrictive class of tree grammars, the complexity becomes linear in the number of nodes of the instance document.

regular expressions, occurrences in our alternative data model correspond to positions (i.e., marked symbols). XML Schema content models have a richer set of operators than regular expressions. In particular, *numeric exponents* ({min occurs} and {max occurs} rather than the Kleene operators ?, $^+$, and *) and *all groups* (i.e., the interleave operator &) complicate the process. Occurrence components only have the boolean properties available which are listed above. Therefore, numeric exponents have to be expanded. A content model $(a\{2,3\} + b\{1, unbounded\})$ thus becomes the regular expression $((a_1, a_2, a_3?) + (b_1, b_2^*))$. The algorithm used is shown in Appendix B.1.

In order to make *all groups* amenable to follow sets, they are expanded as well. The problematic property of *all groups* is that extended expressions (i.e., regular expressions extended by the interleave operator &) are no longer *local*. In an extended expression $(a_1 \& b_1) c_1$, the computation of the follow set of position a_1 cannot be carried out without knowing whether b_1 has already been satisfied or not. Therefore, each all group is expanded into all possible sequences. In Figure 8.1, the expansion of the all group $(a \& b \& c)$ is displayed as a directed graph. Vertices represent positions, and the possible *paths* through the graph correspond to legal sequences. The final vertex is the special *endmarker* position.
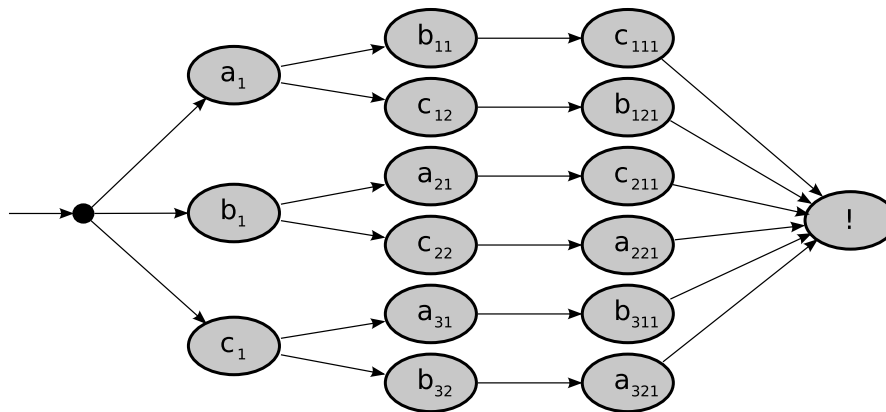

Figure 8.1: Expansion of an *all group* $(a \& b \& c)$

Unfortunately, this expansion is very resource intensive. The recursive algorithm that we employ has a space complexity which can be given recursively as $\mathcal{F}(1) = 1$; $\mathcal{F}(n) = n * \mathcal{F}(n-1) + n$. This is $\approx \mathcal{O}(n!)$, the complexity thus is factorial. As illustration, the first seven values of the sequence are listed: $(1, 4, 15, 64, 325, 1956, 13699, ...)$. Obviously, the complexity can cause actual resource problems, as *all groups* with more than five elements are not too uncommon.[4]

As a *proof of concept* for a prototype implementation, expansion of *all groups* seems affordable. Future implementations, which perhaps use a programming language without the practical limitations of functional programming, may find more efficient algorithms.

---

[4]In particular, this likely becomes problematic when using this approach for attributes as well.

8.2.2 Applications

Berry and Sethi [7] demonstrate how *positions* in marked expression (i.e., symbols made distinct by subscripts) can be viewed as states of an automaton. Based on this observation, we view the set of occurrences that belongs to a complex type as the automaton that accepts all legal sequences of children nodes. While the occurrences correspond to the states, the follow sets describe the outgoing transitions of the respective occurrence. Brüggemann-Klein and Wood [23] also construct the Glushkov automaton of a regular expression using occurrences as the states of the automaton.

We can now emulate the finite state automaton by following the transitions of the automaton for each input symbol. Doing so, we are able to determine the respective occurrence for each input node, as long as there is a coherent path leading to this occurrence. This "coherent path" exists exactly if the sequence of preceding sibling nodes are valid with respect to the content model. We therefore can utilize the special case of determining the occurrence of the last input symbol for *validation* of the content model.

Both methods have been implemented in the XSLT 2.0 function library for the occurrence-based data model. As a consequence, an XSLT application may now perform local validation using the function `occ:validate`, which takes an instance node and returns an XML fragment tree `val:result`. One of the advantages of this follow set-based validation algorithm is that very precise information is available if validation fails.

```
<xsl:variable name="validation-result" as="element(val:result)"
              select="occ:validate(ox:root/ox:seq)"/>
```

8.3 X2Doc: Extensible XML Schema Documentation

Based on SCX, *X2Doc* has been implemented, a framework for generating schema-documentation solely using XSLT [68]. Using a modular set of XSLT stylesheets, X2Doc is highly configurable and carefully crafted to provide extensibility. This proves especially useful for *composite schemas*.

The framework uses SCX as intermediate format and produces XML-based output formats. However, it is important to note that the use of this intermediate format does not necessarily require two-step processing. The transformation of conventional Schema documents into SCX documents is performed by pure XSLT 2.0, and thus the SCX representation can be constructed as a fragment tree at run-time.

8.3.1 Applications

The main advantages of a documentation-generating application based on open technologies like XSLT are high portability, versatile configuration, and simple, yet powerful extensibility.

Extensibility: Extensibility is particularly important in the context of *composite schemas*, where additional information is embedded into XML Schemas. An interesting case is the embedding of other, complementary XML-based schemas like Schematron[5], but other additional information like annotations to control the mapping behavior to relational data bases, or references to a conceptual model, are important cases as well. X2Doc can easily be extended to cover such additional parts by simply adding corresponding template rules.

The following example uses the Schema from the *Primer* for XML Schema 1.0. We assume a company, which uses this Schema, to have defined certain rules for Schema management and documentation: Schema documents are annotated with a company-internal XML vocabulary which resides in a namespace mapped to the prefix `doc`.

```
<complexType name="RegionsType">
  <annotation><appinfo>
    <doc:uri>documentation.html</doc:uri>
    <doc:part>RegionsType</doc:part>
    <doc:author mail="...">Peter Sample</doc:author>
  </appinfo></annotation>
  <sequence><element name="zip" maxOccurs="unbounded">
  ...
```

The annotations relate Schema components to further documentation managed externally in HTML format.

---

[5]http://www.schematron.com/, standardized as ISO/IEC 19757-2:2003.

```
<div id="RegionsType"><p>This is a type for expressing...
```

While most documentation tools could not adapt to this convention without having some code rewritten, documentation generated by our framework is readily extensible to incorporate such embedded parts. It requires only a template rule to be defined that matches application information components. This is easily done, and the resulting stylesheet looks as follows:

```
<xsl:import href="X2Doc-xhtml.xsl"/>
<xsl:template match="scx:application-information">
  <h4>Company-Internal Documentation:</h4>
  <div class="exampleComDoc">
    <xsl:sequence select="id(doc:part, doc(doc:uri))"/>
    <p>Last Autor: <a href="mailto:doc:author/@mail">
        <xsl:value-of select="doc:author"/></a>
    </p>
  </div>
</xsl:template>
```

The first line imports the XHTML module of X2Doc. XSLT's import precedence makes sure that the above template rule overrides the rule defined in the imported module. Figure 8.2 displays the resulting documentation, extended to cover the company-specific annotations.In this example, X2Doc additionally has been configured to include a custom CSS for formatting the company-specific section.

Figure 8.2: Output of X2Doc's XHTML module which has been extended to render company-specific annotations

Configurability: Configurability of X2Doc is possible in different ways: Substantial structural changes can be made by adapting the XSLT template rules. For instance, the appearance and order of basic structural *blocks* (e.g., the table of contents) can be influenced in a central switching template rule. A wide variety of configurations can be made in a configuration XML document. The choice of CSS documents is one example, definition of XML namespace prefixes, configuration of the TOC, and control of the output format and of serialization options are others. Finally, the most important parameters can be overridden through stylesheet parameters.

8.3.2 Documentation Features

Based on SCX's access to the Schema components, the properties of these components can be displayed straightforward. If the transformation had to be carried out on the XML transfer syntax, many of those component properties would have to be collected cumbersomely. Furthermore, the XSLT function library, which is part of SCX, allows for convenient navigation of the relationships between Schema component, e.g., traversal of the type hierarchy. As a result, the documentation generated from SCX is densely hyperlinked. This feature proves to be especially useful for representing the complex structure of XML Schema.

Figure 8.3: A complex type definition as rendered by X2Doc

Figure 8.3 shows the documentation for a complex type definition In the section *Summary*, many of the relationships mentioned above are made navigable through hyperlinks. This comprises the complete path of derivation steps, a list of element declarations referencing the type definition given, a list of types that are derived from this type, and a collection of elements that may appear in this type's model group.

It might seem hard to determine this list of types, but using SCX, only two lines of codes are needed:

```
<xsl:apply-templates
  select="scx:content-type//scf:element-declaration(.)" mode="listing"/>
<xsl:apply-templates select="scx:content-type//scx:wildcard" mode="listing"/>
```

This might also give an idea of how easily custom template rules for alternative documentation formats can be written.

One of the drawbacks of the verbosity of the transfer syntax is that many structures are difficult to understand at first sight. For example, in the syntax of DTDs, model groups are much more concisely defined for the human reader. X2Doc therefore provides a DTD-like notation of content models.

Finally, Figure 8.4 is the rendering of a simple type definition. Here, the list of constraining facets contains hyperlinks to the simple type definitions which originally defined the respective facet. Inherited facets are hard to track in the transfer syntax, but can be readily retrieved from SCX documents.

Figure 8.4: A simple type definition rendered by X2Doc

However, X2Doc is currently a *work in progress.* At present, we work on the completion of the core stylesheets to cover all Schema components, and the extension of the hyperlinked connectivity. The next steps might include the addition of further output formats (using XSL-FO) and the generation of graphics (using SVG).

Chapter 9

Evaluation

Chapter 3 outlined the concept of an accessible data model for XML Schema, and it provided an overview of use cases — both scenarios where an accessible data model would solve existing problems, and scenarios where new opportunities evolve. The aim of this report is not to design the concept in detail and in its ultimate form. This must be the task of a standardization committee. The objective of this report is rather to discuss the prerequisites, to identify possible scenarios of use, and to provide prototypes with which future scenarios can be explored and evaluated. This is ultimately also of importance for the final design, because the knowledge about areas of application, about the needs of different applications, and about problems and requirements of the implementations will help finding appropriate trade-offs, thus essentially shaping this design.

The evaluation in this chapter studies *SPath* (Section 7) and *SCX* (Section 6), together with the function library and the occurrence-based data model perspective, in the context of the scenarios from Section 3. Although SPath and SCX might not achieve all goals, the evaluation is done with the general idea of an accessible data model in mind.

9.1 Documentation

SPath can be helpful for generating documentation, but essentially, SCX is sufficient for this scenario of use, and even superior to SPath in the case of XSLT-based documentation generation. This is mainly due to the SPath's focus on the second class of Schema-processing applications, i.e., *instance-driven* applications, which has been described in Section 5.1.2.

SCX, however, proves extremely helpful in two ways: The *canonicalization* introduced through the transformation from the transfer syntax into SCX makes the generation of documentation straightforward. And the fact that SCX is an XML format allows for

standard technologies, in particular, XSLT 2.0, to be employed.

The main advantage of XSLT-based generation of documentation from XML Schemas is the high extensibility, the versatile configuration, and the interoperability with other XML-based formats and technologies. X2Doc, the framework presented in Section 8.3, exhibits all these properties. As XML Schema becomes increasingly used in large and heterogeneous projects, e.g., in the context of XML pipelines, the need for configurable and extensible documentation grows as well. We expect our modular framework to be a promising platform in this context.

Furthermore, the XML format allows not only documentation to be generated from Schemas. It also substantially facilitates the transformation into any other format. In particular, the generation of XForms [44], stylesheets [53], or XQueries [27] through *meta-stylesheets* is promising.

Because back-transformation of an SCX document into conventional Schema documents is possible, SCX enables annotation, maintenance, and evaluation of XML Schema in a powerful new way. For instance, model information (e.g., from an RDF description) can be injected into an SCX document, which is then transformed back into conventional Schema files. Using SCX, the problems of generating, annotating, and maintaining such composite Schemas are significantly alleviated.

9.2   Information Retrieval

Annotation of conventional Schemas is one use case of SCX in the context of composite Schemas. Another use is the *retrieval* of this additional data from composite Schemas. The canonicalization aspects of SCX guarantee reliable access paths to the {annotation} properties of the Schema components. Section 3.3 already pointed out the potential of model-aware data retrieval. A recent example is *Semantic Annotations for WSDL* [41], which defines the policies and facilities for connecting various parts of WSDL 2.0 [28] documents with an external model. As mentioned in Section 2.4.3, the *data model* in WSDL descriptions of services usually is described by an XML Schema. Hence, *Semantic Annotations for WSDL* also defines annotation mechanisms for XML Schema elements. If stable and navigable paths to such embedded information in a Schema are present, applications can exploit the semantic information, e.g., for *type reflection*. Section 9.3 discusses this in the broader context of Web-based services.

In contrast to documentation, the class of instance-driven applications is relevant for the scenario of information retrieval and data mining, and thus SPath plays an important role as well. Section 3.3 listed conceivable use cases for Schema-aware processing of, and data mining in, instances. It appears that the capabilities of SPath are far more powerful than the basic expressions which are possible in XPath 2.0. For instance, more subtle comparisons of derived types are possible in SPath, as opposed to XPath 2.0. Resuming

the example from Section 3.3, the following comparison illustrates the capabilities of SPath:

```
<!-- XPath 2.0, type-aware -->
//element(*,ex:baseType)
<!-- SPath, schema-aware -->
//*[supertype-by-restriction::ex:baseType]
```

The utilization of XPath's concept of axes in its general interpretation also shows that the power and usefulness of an accessible data model can strongly be augmented by well-designed language constructs which support common tasks. This holds for SPath's axes, but for the functions from the function library of the prototype implementation just as well.

9.3 Web-Based Services

The issues and potential of Web-based services overlap in large parts with the problems of and opportunities of data retrieval, validation, and especially versioning and extensibility. In this section, we therefore only briefly discuss the general properties, and we leave the detailed description to the Sections 9.2, 9.5, and 9.4, respectively.

Two categories of how data model accessibility can improve Web-based services can be discerned, although the differences between the two are of gradual nature.

Resilient Operation: Data model accessibility enables *type introspection*. This makes Web-based services more adaptable and resilient. More resilient operation can involve both sophisticated versioning strategies and novel concepts of validation — or the ability to perform *compatibility transformations* of incoming data in general. Schema-aware validation-by-projection is a prominent example.

Both SPath and SCX support type introspection. *Type reflection,* which is more advanced than type introspection, and which involves more fundamental run-time changes of the behavior of an application, is better supported by SCX. Type reflection for XML-based service applications might be done in XML pipelines, where meta-stylesheets generate the actual processing stylesheets at run-time, based on Schema information from an SCX document.

Model-Driven Integration or Development: Starting with type reflection, the logical next step can be model-driven integration or development of parts of service applications. SCX and the function library are a good starting point, and they provide already many of the features needed. Of course, XML-based representations face strong competition from Schema APIs, which provide Schema access, combined with less restricted and more high-performance programming languages. However, current APIs do not expose Schema information in a unified manner, which is a drawback with respect to development, maintenance, and portability.

With the spread of XQuery 1.0 and XSLT 2.0, which are diligently designed for optimizations, there is also some evidence that larger parts of XML applications increasingly will be using these standard XML technologies, rather than general-purpose programming languages like Java. On the other hand, frameworks like XJ demonstrate the possibility of integrating XML technologies (e.g., XPath) into conventional programming languages [48]. SPath, or a more general form of data model accessibility, are amenable to integration just as well.

9.4 Versioning and Extensibility

Both SCX and SPath can play an essential role for versioning and extensibility. Up to the present, consistent versioning strategies for XML have been rare. One of the reasons might be that well-designed, XML Schema-based versioning did not offer a sufficient number of direct advantages. With the ability to access and navigate Schema information, versioning information becomes exploitable as well, if the versioning history has been encoded in a suitable way, as it is the case for UBL, for instance.

It is very likely that further development of more sophisticated, suitable, and useful versioning strategies for XML vocabularies sets off once technologies which allow to utilize versioning information are available. Many of the current obstacles with versioning and extensibility will become obsolete. This hopefully opens new perspectives onto more advanced questions.

Two examples should justify this assumption. Orchard advocates the introduction of an additional Schema-related attribute in instances, `xsi:basetype`. This should enable processing applications to determine a fallback type of instances, if the type specified through `xsi:type` is unknown. This attribute is obsolete in most of the cases.[1] Inspection of the type hierarchy readily yields the base type. And if no up-to-date Schema is retrievable, an ancestor type can most likely be determined, due to XML Schema's *element declaration consistent* and *unique particle attribution* rules.

Obasanjo suggests to use different namespaces for extensions [75]:

> If the namespace name of the extensions is an HTTP URI that points to human- and machine-readable information about the extensions then it allows consumers of the format the chance to learn about the extensions they encounter.

This is not true for most namespace URIs today. Moreover, namespace URIs are not intended to provide this functionality, while other constructs are explicitly designed for this purpose. The annotation facilities in XML Schema are a good example. If navigable Schema information is available (e.g., through an SCX document), meta-information

---

[1]In rare cases, where no up-to-date Schema is available and the incriminated node matches a wildcard particle, determination of the fallback type may fail.

about the structure and semantics of an extension can be retrieved in a much more stable and powerful way. Without (ab-)using namespace URIs for annotation purposes, Orchards rule can then be applied again, which advocates to re-use namespace URIs for backward-compatible changes, which extensions always should be.

Based on canonical representation and deep comparison operators — that is: based on SCX and the function library — equivalence checking of Schemas becomes possible. A *Schema-diff* — similar to POSIX's `diff` — will be particularly useful for versioning strategies, but has general applications as well. Comparison of Schemas might be the foundation of *version mapping*, as a special case of Schema mapping. *Compatibility transformations* similar to validation-by-projection also are important in the context of versioning and extensibility.

9.5 Validation

SCX, SPath, and accessible data models in general are the starting point for building more versatile, adaptable, and tolerant validation. This can be done on different levels, and both for instances and Schemas.

9.5.1 Instance Validation

The concept of *Schema-aware validation-by-projection*, which has been proposed by Orchard and Bau, is an example of a more tolerant variant of Schema-validation of instances. It can also be seen as a sub-case of Schema-mapping. More recent versions of Schemas are mapped to preceding versions, causing some elements to be discarded. Both SPath and SCX can be used to implement validation-by-projection. In its most basic interpretation, validation-by-projection merely means "identifying non-recognizable elements in type extensions, and removing them." It is evident that this can be accomplished straightforwardly using SCX and the accessor functions from the function library.

The occurrence-based data model introduces even more radical ways of new forms of adaptable validation. The subsumption-checking methods can be used in order to perform *partial validation* at *run-time*, as described in Section 8.2.2. This is useful if an application is very tolerant in most parts of an instance, but very sensitive in particular parts. The application then can perform the required validation while processing the instance. Using partial validation, the processing modes can be differentiated within one document. At present, the processing mode can only be adjusted for the whole document and for wildcard content. But recalling the twofold purpose of Schema-validation from Section 2.3.1, it may be desirable to specify different processing modes for different parts of a document (or, more precisely, for different complex types of a Schema).

Overall, both validation variants — validation-by-projection and partial validation — indicate a shift of control from schemas to validators, and a shift of responsibilities from producers to consumers. Extensions (or other structural deviations) which cannot be avoided on the producing side (e.g., because no central versioning control is possible) are handled on the consuming side (e.g., through validation-by-projection). Rules which cannot be expressed by XML Schema (e.g., different importance of grammar constraints in different contexts of the document) are built into the validator, rather than into the Schema.

It is worth considering to include the ability of validation into SPath as well. However, this would have strong and far-reaching consequences for the way Schema-validation is carried out today. Essentially, it could result in a complete disassociation of validation and type-annotation, the latter then becoming a *best effort* type annotation.

Finally, SPath — or SCX together with the accessor functions — can readily be used in order to implement a type-aware rule-based schema language. This would address one of the flaws of Schematron, which is that Schematron only addresses *elements*, whereas the relevant properties often are associated to *types*. Currently, this requires to write and maintain path expressions for every element which references a given type. This is cumbersome and error-prone. A type-aware or *Schema-aware* kind of Schematron would be more powerful and robust.

9.5.2 Schema Checking

SCX is helpful for Schema-checking. Firstly, the validation of SCX documents through the XML Schema for SCX covers a wider range of constraints than the XML Schema for Schemas (6.2). Secondly, a rule-based language for Schema-checking can easily be developed using SCX and the function library. A rule-based Schema-checking language can be utilized in order to assert the compliance of a Schema with certain *Naming and Design Rules* (NDR) or Schema *guidelines* or *best practices*. Companies or organizations usually define such rules, but it is often hard to check conventional Schema documents against these rules.

A canonical form of Schema avoids the problems of too many degrees of freedom. And an XML-based representation allows one to use standard technologies like XSLT for the implementation of a Schema checker. SPath can make the rules in such a Schema-checker more robust, because language elements like axes let the developer write more generic path expressions, which are less likely to fail if the Schema changes.

Chapter 10

Conclusion and Outlook

The evaluation in Section 9 clearly demonstrates the appropriateness of our approaches in various fields, and the benefits they yield. Of course, our prototype implementation can be refined, and next steps can be taken in order to improve efficiency, robustness, and seamless integration with current XML technologies.

X2Doc: The documentation-generating framework presented in Section 8.3 could be augmented with more modules, e.g., for supporting different output formats (e.g., using XSL-FO).

SCF: The robustness of the accessor functions still can be improved; for instance, the current implementation fails at determining the correct element type if the corresponding Schema has `elementFormDefault` set to `false`.

SPath: As a first step, a pre-processing meta-stylesheet could be written, which maps SPath expressions in an SPath-enabled XSLT 2.0 stylesheet to SCF function calls. For example, an SPath expression `$node/super-type::*/name()` then would be mapped to the XPath expression `$node/scf:get-super-types(.)/scx:name`.

XNI-Based Implementation: As a next step, accessor functions could be written to utilize the PSVI through the Xerces Native Interface (XNI). This would increase the efficiency, as discussed in Section 8.1.3.

Standardization SPath: The — admittedly somewhat utopian — long-term goal would be to integrate SPath into XPath, and to standardize the language and its syntax, semantics, and behavior.

In conclusion it can be said that SCX, the Schema components XML syntax, proves very helpful and offers a wide range of applications — even without the support of an XML Schema path language like SPath.

The class of instance-driven Schema-aware applications will certainly benefit from a Schema-aware extension of XPath. A prototype implementation of SPath will demon-

strate the advantages and enable further improvements of the syntax. Use cases will then show that SPath is fully Schema-aware, whereas XPath is only type-aware, and how true Schema-awareness permits novel ways of XML processing.

A unified representation of the data model of XML Schema will offer even more fundamental advantages. The processing of XML Schema will become substantially less difficult, and XML Schema will finally become part not only of the specifications of other technologies, but also of their applications. A unified representation includes aspects of canonicalization; and a canonical form is a necessary prerequisite for a concise and useful formal description, which in turn will be important for a future generation of more robust and optimizable XML technologies.

The fact that SCX is an XML format might seem to be a concession made for a prototype implementation. On a closer look, however, it appears to be a powerful feature. Wilde proposes an XML syntax for his *Extensible XML Information Set* [95], and Thompson proposes to represent the PSVI as a kind of "synthetic XML" [92] in order to enable reflection to be used in XPath. SCX utilizes a similar concept on a much more powerful level. Not only allows SCX for thorough introspection and reflection of Schema information, due to the *hooks* (or *extension points*) in XML Schema, it allows to access every kind of XML-based meta-information. Assume an XML Schema which contains annotations that link into an XML-based representation of an underlying model (e.g., an RDF model, or ontology): given an accessible data model, applications can use an XML path language — be it XPath, together with accessor functions, or SPath — in order to navigate through the Schema, the model, and beyond.

Bibliography

[1] Jürgen Albert, Dora Giammarresi, and Derick Wood. Normal form algorithms for extended context-free grammars. *Theor. Comput. Sci.*, 267(1-2):35–47, 2001.

[2] Marcelo Arenas and Leonid Libkin. A normal form for XML documents. *ACM Trans. Database Syst.*, 29:195–232, 2004.

[3] David Bau. Theory of Compatibility (Parts 1-3). http://www.webcitation.org/5OWcyVW4R, Dec 2003.

[4] Sean Bechhofer, Frank van Harmelen, James A. Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL Web Ontology Language Reference. World Wide Web Consortium, Recommendation REC-owl-ref-20040210, February 2004.

[5] Anders Berglund, Scott Boag, Donald D. Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path Language (XPath) 2.0. World Wide Web Consortium, Recommendation REC-xpath20-20070123, January 2007.

[6] Tim Berners-Lee, Roy T. Fielding, and Henrik Frystyk Nielsen. Hypertext Transfer Protocol – HTTP/1.0. IETF RFC 1945, May 1996.

[7] Gérard Berry and Ravi Sethi. From Regular Expressions to Deterministic Automata. *Theoretical Computer Science*, 48(3):117–126, 1986.

[8] Jean Berstel and Luc Boasson. Formal properties of XML grammars and languages. *Acta Informatica*, 38(9):649–671, 2002.

[9] Geert Jan Bex, Wim Martens, Frank Neven, and Thomas Schwentick. Expressiveness of XSDs: From Practice to Theory, There and Back Again. In *Proceedings of the 14th International World Wide Web Conference*, pages 712–721, Chiba, Japan, May 2005. ACM Press.

[10] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes Second Edition. World Wide Web Consortium, Recommendation REC-xmlschema-2-20041028, October 2004.

[11] SCOTT BOAG, DONALD D. CHAMBERLIN, MARY F. FERNÁNDEZ, DANIELA FLORESCU, JONATHAN ROBIE, and JÉRÔME SIMÉON. XQuery 1.0: An XML Query Language. World Wide Web Consortium, Recommendation REC-xquery-20070123, January 2007.

[12] DON BOX, DAVID EHNEBUSKE, GOPAL KAKIVAYA, ANDREW LAYMAN, NOAH MENDELSOHN, HENRIK FRYSTYK NIELSEN, SATISH THATTE, and DAVE WINER. Simple Object Access Protocol (SOAP) 1.1. World Wide Web Consortium, Note NOTE-SOAP-20000508, May 2000.

[13] ROBERT BRADEN. Requirements for Internet Hosts – Communication Layers. IETF RFC 1122, October 1989.

[14] WALTER S. BRAINERD. Tree Generating Regular Systems. *Information and Control*, 14(2):217–231, 1969.

[15] TIM BRAY, CHARLES FRANKSTON, and ASHOK MALHOTRA. Document Content Description for XML. World Wide Web Consortium, Note NOTE-dcd-19980731, July 1998.

[16] TIM BRAY, DAVE HOLLANDER, and ANDREW LAYMAN. Namespaces in XML. World Wide Web Consortium, Recommendation REC-xml-names-19990114, January 1999.

[17] TIM BRAY, DAVE HOLLANDER, ANDREW LAYMAN, and RICHARD TOBIN. Namespaces in XML 1.0 (Second Edition). World Wide Web Consortium, Recommendation REC-xml-names-20060816, August 2006.

[18] TIM BRAY, JEAN PAOLI, and C. MICHAEL SPERBERG-MCQUEEN. Extensible Markup Language (XML) 1.0 (Fourth Edition). World Wide Web Consortium, Recommendation REC-xml-20060816, August 2006.

[19] ALLEN BROWN, MATTHEW FUCHS, JONATHAN ROBIE, and PHILIP WADLER. XML Schema: Formal Description. World Wide Web Consortium, Working Draft WD-xmlschema-formal-20010925, September 2001.

[20] ANNE BRÜGGEMANN-KLEIN. Formal Models in Document Processing. ftp://ftp.informatik.uni-freiburg.de/documents/papers/brueggem/habil.ps, 1993.

[21] ANNE BRÜGGEMANN-KLEIN. Unambiguity of Extended Regular Expressions in SGML Document Grammars. In THOMAS LENGAUER, editor, *ESA*, volume 726 of *Lecture Notes in Computer Science*, pages 73–84. Springer, 1993.

[22] ANNE BRÜGGEMANN-KLEIN and DERICK WOOD. Balanced Context-Free Grammars, Hedge Grammars and Pushdown Caterpillar Automata. In IDEAlliance, editor, *Extreme Markup Languages*, 2004.

[23] Anne Brüggemann-Klein and Derick Wood. The Validation of SGML Content Models. *Mathematical and Computer Modelling*, 25:73–84(12), February 1997.

[24] Janusz A. Brzozowski. Derivatives of Regular Expressions. *J. ACM*, 11(4):481–494, 1964.

[25] Ben Chang, Elena Litani, Joe Kesselman, and Rezaur Rahman. Document Object Model (DOM) Level 3 Abstract Schemas Specification (Version 1.0). World Wide Web Consortium, Note NOTE-DOM-Level-3-AS-20020725, July 2002.

[26] Peter Pin-Shan Chen. The Entity-Relationship Model — Toward a Unified View of Data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[27] Ya Bing Chen, Tok Wang Ling, and Mong-Li Lee. Automatic Generation of XQuery View Definitions from ORA-SS Views. In Il-Yeol Song, Stephen W. Liddle, Tok Wang Ling, and Peter Scheuermann, editors, *ER*, volume 2813 of *Lecture Notes in Computer Science*, pages 158–171. Springer, 2003.

[28] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. World Wide Web Consortium, W3C Candidate Recommendation CR-wsdl20-20060327, March 2006.

[29] N. Chomsky. Three Models for the Description of Language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[30] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. World Wide Web Consortium, Note NOTE-wsdl-20010315, March 2001.

[31] James Clark. RELAX NG Specification. Organization for the Advancement of Structured Information Standards (OASIS), Committee Specification, December 2001.

[32] Dan Connolly. Gleaning Resource Descriptions from Dialects of Languages (GRDDL). World Wide Web Consortium, Working Draft WD-grddl-20061024, October 2006.

[33] Roger L. Costello. XML Schemas: Best Practices. http://xfront.com/BestPracticesHomepage.html, November 2006.

[34] John Cowan and Richard Tobin. XML Information Set (Second Edition). World Wide Web Consortium, Recommendation REC-xml-infoset-20040204, February 2004.

[35] Andrew Davidson, Matthew Fuchs, Mette Hedin, Mudita Jain, Jari Koistinen, Chris Lloyd, Murray Maloney, and Kelly Schwarzhof.

Schema for Object-Oriented XML 2.0. World Wide Web Consortium, Note NOTE-SOX-19990730, July 1999.

[36] Denise Draper, Peter Fankhauser, Mary F. Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. World Wide Web Consortium, Recommendation REC-xquery-semantics-20070123, January 2007.

[37] Daniel Dui and Wolfgang Emmerich. Compatibility of XML Language Versions. In Bernhard Westfechtel and André van der Hoek, editors, *SCM*, volume 2649 of *Lecture Notes in Computer Science*, pages 148–162. Springer, 2003.

[38] David W. Embley, Stephen W. Liddle, and Reema Al-Kamha. Enterprise Modeling with Conceptual XML. In Paolo Atzeni, Wesley W. Chu, Hongjun Lu, Shuigeng Zhou, and Tok Wang Ling, editors, *er2004*, volume 3288 of *Lecture Notes in Computer Science*, pages 150–165. Springer, 2004.

[39] David W. Embley and Wai Yin Mok. Developing XML Documents with Guaranteed "Good" Properties. In Hideko S. Kunii, Sushil Jajodia, and Arne Sølvberg, editors, *er2001*, volume 2224 of *Lecture Notes in Computer Science*, pages 426–441. Springer, 2001.

[40] David C. Fallside and Priscilla Walmsley. XML Schema Part 0: Primer Second Edition. World Wide Web Consortium, Recommendation REC-xmlschema-0-20041028, October 2004.

[41] Joel Farrell and Holger Lausen. Semantic Annotations for WSDL. Working Draft WD-sawsdl-20060928, September 2006.

[42] Mary F. Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. XQuery 1.0 and XPath 2.0 Data Model (XDM). World Wide Web Consortium, Recommendation REC-xpath-datamodel-20070123, January 2007.

[43] Charles Frankston and Henry S. Thompson. XML-Data Reduced (Version 0.21), July 1998.

[44] Patrick Garvey and Bill French. Generating User Interfaces from Composite Schemas. In *Proceedings of XML 2003*, Philadelphia, Pennsylvania, December 2003.

[45] Arofan Gregory and Eduardo Gutentag. XML Schema Libraries and Versioning: The UBL Case. In IDEAlliance, editor, *Extreme Markup Languages*, 2003.

[46] Arofan Gregory and Eduardo Gutentag. UBL and Object-Oriented XML: Making Type-Aware Systems Work. In IDEAlliance, editor, *Extreme Markup Languages*, 2004.

[47] HARRY HALPIN. XMLVS: Using Namespace Documents for XML Versioning. In IDEAlliance, editor, *Extreme Markup Languages*, 2006.

[48] MATTHEW HARREN, MUKUND RAGHAVACHARI, ODED SHMUELI, MICHAEL G. BURKE, RAJESH BORDAWEKAR, IGOR PECHTCHANSKI, and VIVEK SARKAR. XJ: Facilitating XML Processing in Java. In *Proceedings of the 14th International World Wide Web Conference*, pages 278–287, Chiba, Japan, May 2005. ACM Press.

[49] RALPH HODGSON, PAUL KELLER, and HOLGER KNUBLAUCH. Ontology-Based XML Schemas for Interoperability between Systems and Tools. In *Proceedings of XML 2006*, Boston, Massachusetts, December 2006.

[50] MARY HOLSTEGE and ASIR S. VEDAMUTHU. XML Schema: Component Designators. World Wide Web Consortium, Working Draft WD-xmlschema-ref-20050329, March 2005.

[51] JOHN E. HOPCROFT, RAJEEV MOTWANI, and JEFFREY D. ULLMAN. *Introduction to automata theory, languages, and computation (2nd edition)*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2001.

[52] STIJN HOPPENBROUWERS, HENDERIK ALEX PROPER, and THEO P. VAN DER WEIDE. A Fundamental View on the Process of Conceptual Modeling. In LOIS M. L. DELCAMBRE, CHRISTIAN KOP, HEINRICH C. MAYR, JOHN MYLOPOULOS, and OSCAR PASTOR, editors, *er2005*, volume 3716 of *Lecture Notes in Computer Science*, pages 128–143. Springer, 2005.

[53] MICHAEL KAY. Meta-stylesheets. In *Proceedings of XML 2006*, Boston, Massachusetts, December 2006.

[54] MICHAEL KAY. XSL Transformations (XSLT) Version 2.0. World Wide Web Consortium, Recommendation REC-xslt20-20070123, January 2007.

[55] GRAHAM KLYNE and JEREMY J. CARROLL. Resource Description Framework (RDF): Concepts and Abstract Syntax. World Wide Web Consortium, Recommendation REC-rdf-concepts-20040210, February 2004.

[56] RALF LÄMMEL, STAN KITSIS, and DAVE REMY. Analysis of XML Schema Usage. In *Proceedings of XML 2005*, Atlanta, Georgia, November 2005.

[57] ARNAUD LE HORS, PHILIPPE LE HÉGARET, LAUREN WOOD, GAVIN THOMAS NICOL, JONATHAN ROBIE, MIKE CHAMPION, and STEVEN BYRNE. Document Object Model (DOM) Level 2 Core Specification. World Wide Web Consortium, Recommendation REC-DOM-Level-2-Core-20001113, November 2000.

[58] ELENA LITANI. XML Schema API. World Wide Web Consortium, Member Submission SUBM-xmlschema-api-20040309, March 2004.

[59] ELENA LITANI and LISA MARTIN. An API to Query XML Schema Components and the PSVI. In *Proceedings of XML Europe 2004*, Amsterdam, Netherlands, April 2004.

[60] JAYANT MADHAVAN, PHILIP A. BERNSTEIN, and ERHARD RAHM. Generic Schema Matching with Cupid. Technical report, Microsoft Corporation, Redmond, Washington, August 2001.

[61] ASHOK MALHOTRA, JIM MELTON, and NORMAN WALSH. XQuery 1.0 and XPath 2.0 Functions and Operators. World Wide Web Consortium, Recommendation REC-xpath-functions-20070123, January 2007.

[62] MURALI MANI. EReX: A Conceptual Model for XML. In ZOHRA BELLAHSENE, TOVA MILO, MICHAEL RYS, DAN SUCIU, and RAINER UNLAND, editors, *xsym2004*, volume 3186 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2004.

[63] MURALI MANI, DONGWON LEE, and RICHARD R. MUNTZ. Semantic Data Modeling Using XML Schemas. In HIDEKO S. KUNII, SUSHIL JAJODIA, and ARNE SØLVBERG, editors, *ER*, volume 2224 of *Lecture Notes in Computer Science*, pages 149–163. Springer, 2001.

[64] R. MCNAUGHTON and H. YAMADA. Regular Expressions and State Graphs for Automata. *IRE Transactions on Electronic Computers*, EC-9(1):38–47, March 1960.

[65] JIM MELTON and SUBRAMANIAN MURALIDHAR. XML Syntax for XQuery 1.0 (XQueryX). World Wide Web Consortium, Recommendation REC-xqueryx-20070123, January 2007.

[66] FELIX MICHEL. Opening XML Schema's Data Model to XPath 2.0. Technical Report TIK Report No. 264, Computer Engineering and Networks Laboratory, ETH Zürich, Zürich, Switzerland, November 2006.

[67] FELIX MICHEL and ERIK WILDE. XML Schema Editors — A Comparison of Real-World XML Schema Visualizations. Technical Report TIK Report No. 265, Computer Engineering and Networks Laboratory, ETH Zürich, Zürich, Switzerland, December 2006.

[68] FELIX MICHEL and ERIK WILDE. Extensible Schema Documentation with XSLT 2.0. In *Poster Proceedings of the 16th International World Wide Web Conference*, Banff, Alberta, May 2007. ACM Press.

[69] SRIRAM MOHAN and ARIJIT SENGUPTA. Conceptual Modeling for XML — A Myth or a Reality? In ZONGMIN MA, editor, *Database Modeling for Industrial Data Management: Emerging Technologies and Applications*, chapter X, pages 293–322. Idea Group Inc., Hershey, Pennsylvania, December 2005.

[70] ANDERS MØLLER. Document Structure Description 2.0, December 2002.

[71] MAKOTO MURATA. Hedge automata: a formal model for XML schemata. http://www.xml.gr.jp/relax/hedge_nice.html, October 1999.

[72] MAKOTO MURATA, DONGWON LEE, and MURALI MANI. Taxonomy of XML Schema Languages Using Formal Language Theory. In IDEAlliance, editor, *Extreme Markup Languages*, 2001.

[73] ANDREAS NEUMANN. Unambiguity of SMGL Content Models - Pushdown Automata Revisited. *Universität Trier, Mathematik/Informatik, Forschungsbericht*, 97-05, 1997.

[74] MARTIN NEČASKÝ. Conceptual Modeling for XML: A Survey. In VÁCLAV SNÁŠEL, KAREL RICHTA, and JAROSLAV POKORNÝ, editors, *Proceedings of the DATESO 2006 Annual International Workshop on Databases, Texts, Specifications, and Objects*, Desná — Černá Říčka, Czech Republic, April 2006.

[75] DARE OBASANJO. Designing Extensible, Versionable XML Formats. http://www.xml.com/pub/a/2004/07/21/design.html, July 2004.

[76] DAVID ORCHARD. Versioning XML Vocabularies. http://www.xml.com/pub/a/2003/12/03/versioning.html, December 2003.

[77] DAVID ORCHARD. Extensibility, XML Vocabularies, and XML Schema. http://www.xml.com/pub/a/2004/10/27/extend.html, October 2004.

[78] DAVID ORCHARD. Providing Compatible Schema Evolution. http://www.webcitation.org/5OXVLVEQl, January 2004.

[79] DAVID ORCHARD. Guide to Versioning XML Languages using XML Schema 1.1. W3C Working Draft WD-xmlschema-guide2versioning-20060928, September 2006.

[80] DAVID ORCHARD and NORMAN WALSH. [Editorial Draft] Extending and Versioning Languages Part 1. W3C Draft TAG Finding versioning-20060726, July 2006.

[81] IGOR PESHANSKY and MUKUND RAGHAVACHARI. Language Support for Web Service Development. In *Proceedings of XML 2006*, Boston, Massachusetts, December 2006.

[82] DAVID PETERSON, PAUL V. BIRON, ASHOK MALHOTRA, and C. MICHAEL SPERBERG-MCQUEEN. XML Schema 1.1 Part 2: Datatypes. World Wide Web Consortium, Working Draft WD-xmlschema11-2-20060217, February 2006.

[83] LARRY L. PETERSON and BRUCE S. DAVIE. *Computer Networks, A System Approach*. The Morgan Kaufmann Series in Networking. Morgan Kaufmann Publishers, 3rd edition, 2003.

[84] GIUSEPPE PSAILA. ERX: An Experience in Integrating Entity-Relationship Models, Relational Databases, and XML Technologies. In AKMAL B. CHAUDHRI,

Rainer Unland, Chabane Djeraba, and Wolfgang Lindner, editors, *EDBT Workshops*, volume 2490 of *Lecture Notes in Computer Science*, pages 242–265. Springer, 2002.

[85] Arijit Sengupta. XER — Extensible Entity Relationship Modeling. In *Proceedings of XML 2003*, Philadelphia, Pennsylvania, December 2003.

[86] Arijit Sengupta and Sriram Mohan. Formal and Conceptual Models for XML Structures — The Past, Present, and Future. Technical Report 137-1, Indiana University, Information Systems Department, Bloomington, Indiana, April 2003.

[87] Arijit Sengupta and Erik Wilde. The Case for Conceptual Modeling for XML. Technical Report TIK Report No. 244, Computer Engineering and Networks Laboratory, ETH Zürich, Zürich, Switzerland, February 2006.

[88] Jérôme Siméon and Philip Wadler. The Essence of XML. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–13, New Orleans, Louisiana, January 2003. ACM Press, ACM Press.

[89] C. M. Sperberg-McQueen. Applications of Brzozowski derivatives to XML Schema processing. In IDEAlliance, editor, *Extreme Markup Languages*, 2005.

[90] Henry S. Thompson. Towards a logical foundation for XML Schema. In *Proceedings of XML Europe 2004*, Amsterdam, Netherlands, April 2004.

[91] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures Second Edition. World Wide Web Consortium, Recommendation REC-xmlschema-1-20041028, October 2004.

[92] Henry S. Thompson, K. Ari Krupnikov, and Jo Calder. Uniform access to infosets via reflection. In IDEAlliance, editor, *Extreme Markup Languages*, 2003.

[93] Henry S. Thompson, C. Michael Sperberg-McQueen, Shudi Gao, Noah Mendelsohn, David Beech, and Murray Maloney. XML Schema 1.1 Part 1: Structures. World Wide Web Consortium, Working Draft WD-xmlschema11-1-20060831, August 2006.

[94] Norman Walsh and Alex Milowski. XProc: An XML Pipeline Language. World Wide Web Consortium, Working Draft WD-xproc-20061117, November 2006.

[95] Erik Wilde. The Extensible XML Information Set. Technical Report TIK Report No. 160, Computer Engineering and Networks Laboratory, ETH Zürich, Zürich, Switzerland, February 2003.

[96] Erik Wilde. Semantically Extensible Schemas for Web Service Evolution. In Liang-Jie Zhang and Mario Jeckle, editors, *Web Services — Proceedings of the 2004 European Conference on Web Services*, volume 3250 of *Lecture Notes in Computer Science*, pages 30–45, Erfurt, Germany, September 2004. Springer-Verlag.

[97] Erik Wilde. Describing Namespaces with GRDDL. In *Poster Proceedings of the 14th International World Wide Web Conference*, pages 1002–1003, Chiba, Japan, May 2005. ACM Press.

[98] Erik Wilde. Structuring Namespace Descriptions. In *Poster Proceedings of the 15th International World Wide Web Conference*, Edinburgh, UK, May 2006. ACM Press.

[99] Erik Wilde and Felix Michel. SPath: A Path Language for XML Schema. Technical Report UCB iSchool Report 2007-001, School of Information, UC Berkeley, Berkeley, California, February 2007.

[100] Erik Wilde and Felix Michel. SPath: A Path Language for XML Schema. In *Poster Proceedings of the 16th International World Wide Web Conference*, Banff, Alberta, May 2007. ACM Press.

[101] Erik Wilde and Felix Michel. XML-Based XML Schema Access. In *Poster Proceedings of the 16th International World Wide Web Conference*, Banff, Alberta, May 2007. ACM Press.

[102] Erik Wilde and Willy Müller. Organizing Federal E-Government Schemas. Technical Report TIK Report No. 212, Computer Engineering and Networks Laboratory, ETH Zürich, Zürich, Switzerland, February 2005.

[103] Erik Wilde and Kilian Stillhard. A Compact XML Schema Syntax. In *Proceedings of XML Europe 2003*, London, UK, May 2003.

Appendix A

SPath EBNF Syntax

A.1   Necessary Adaptations on XPath 2.0's EBNF

```
PathExpr          ::= ( "/" RelativePathExpr? )
                    | ( "//" RelativePathExpr )
                    | RelativePathExpr
                    | SPathExpr
```

A.2   Extensions through SPath

```
SPathExpr         ::= SStepExpr ( "/" SStepExpr )*

SStepExpr         ::= SAxisStep | SFilterExpr

SAxisStep         ::= ( TypeAxisStep
                    | ElDeclAxisStep
                    | OccurrAxisStep
                    | ConstrAxisStep
                    | InstncAxisStep
                    | SchemaAxisStep ) | PredicateList


TypeAxisStep    ::= TypeAxis "::" NodeTest
ElDeclAxisStep  ::= ElDeclAxis "::" NodeTest
OccurrAxisStep  ::= OccurrAxis "::" NodeTest
ConstrAxisStep  ::= ConstrAxis "::" NodeTest
InstncAxisStep  ::= InstncAxis "::" NodeTest
SchemaAxisStep  ::= SchemaAxis "::" NodeTest


TypeAxis          ::= "type"
```

```
                          | ( ( DerivedTypeAxis |  BaseTypeAxis)
                              ("-" TypeAxisModifier)?
                          )

DerivedTypeAxis  ::= ( "derivedtype"
                     | "subtype"
                     | "subtype-or-self"
                     | "substituted-by" )

BaseTypeAxis     ::= ( "basetype"
                     | "supertype"
                     | "supertype-or-self"
                     | "substitutes-for" )

ElDeclAxis       ::= "declaration"
                   | ( ( "substitution-group" | "substitution-head" )
                       ("-" TypeAxisModifier)?
                     )

OccurrAxis       ::= ( "occurrence"
                     | "contains"
                     | "preceded-by"
                     | "followed-by" )

ConstrAxis       ::= ( "constraint"
                     | "constrained-by"
                     | "refer"
                     | "referred-by" )

InstncAxis       ::= "instance"

SchemaAxis       ::= "schema"

TypeAxisModifier ::= ( "by-extension" | "by-restriction" )

SFilterExpr      ::= SPrimaryExpr PredicateList

SPrimaryExpr     ::= FuncCall | ContextItemExpr
```

A.3   Excerpt from XPath 2.0

```
PrimaryExpr      ::= Literal | VarRef | ParenthesizedExpr
                   | ContextItemExpr | FunctionCall

PredicateList    ::= Predicate *

Predicate        ::= "[" Expr "]"
```

Appendix B

Algorithms

B.1   Instance-Based Accessor Functions

```
function Type
  in:  - S: a Schema component
       - N: an instance node N
  out: a type definition component

  1: if (N has a xsi:type attribute)
  2:   return the global type definition where QName eq N/@xsi:type
  3: else
  4:   return S/type-definition
  5: endif

function GetDeclaration
  in:  - N: an instance node N
  out: an element declaration component

  1: Q := the QPath of N
  2: S := the Schema-as-a-whole component
  3: return Declaration(S, N, Q)

function GetType
  in:  - N: an instance node N
  out: a type definition component

  1: E := GetDeclaration(N)
  2: return Type(E, N)
```

```
function Declaration
  in:  - S: a Schema component
       - N: an instance node N
       - Q: a sequence of QNames describing the canonical path
         from the instance root to this node N
  out: an element declaration component

  1: if (S is the Schema-as-a-whole component) then
  2:   C := all global element declarations
  3: else
  4:   A := the ancestor nodes of N
  5:   T := Type(S, A[count(Q) - 1])
  6:   M := the resolved model group of T
  7:   C := all element declarations in M
  8: endif
  9: E := the declaration in C where particle/term matches Q[0]
 10: if (S contains only 1 item) then
 11:   return E
 12: else
 13:   pop(Q)
 14:  Declaration(E, N, Q)
 15: endif
```

B.2   Expansion of Numeric Exponents

```
function ExpandNumericExponents
  in:  - P: a particle component
  out: a set of occurrence components

  # occurrences are a 3-tupel: <term:, optional, unbounded>
  1: S := the empty set
  2: for (I in 1 to P/min-occurs) do
  3:   push(S, <P/term, FALSE, FALSE>)
  4: endfor
  5: if (P/max-occurs is unbounded) then
  6:   push(S, <P/term, TRUE, TRUE>)
  7: else
  8:   for (I in P/min-occurs + 1 to P/max-occurs) do
  9:     push(S, <P/term, TRUE, FALSE>)
 10:   endfor
 11: endif
 12: return S
```