Eidgenössische Technische Hochschule Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik

Wintersemester 02/03

Diploma Thesis

A Compact Syntax for XML Schema

Author: Kilian Stillhard

Reader: Erik Wilde

Abstract

The XML Schema language offers a variety of useful features to the XML schema designer. But due to its XML syntax and its inherent complexity, XML Schemas are difficult to read and understand. To reduce XML Schema's syntactical verboseness and complexity, a new syntax has been defined, the *XML Schema Compact Syntax* (XSCS). The compact syntax is designed for the human user, by reusing well-known programming and schema language concepts. It reduces schema size about 60%. A parser for the compact syntax has been implemented to allow conversion from compact syntax to XML Schema and back.

Zürich, 18. March 2003

Kilian Stillhard

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 XML and XML Schema

XML is a standardized syntax for markup languages defined by the W3C consortium. XML documents are text documents containing special character sequences, the markup, that describe the semantics of the documents' content. XML documents have a tree structure where the nodes are *elements* that can contain text and/or other elements. *Attributes* are attached to elements and provide further information about the element and its contents.

The XML standard [1] however does not define any element or attribute names nor any semantics. To represent a particular document type, an *XML application* has to be designed by describing the set of allowed documents. This can be done using prose text, however this approach becomes impractical when computer processing or validation of the described documents is needed. More formal ways of describing rules for allowed document content are suitable in this case.

Many *schema definition languages* have been proposed, all with different advantages and drawbacks. The XML standard itself contains the DTD (Document Type Definition) language [1], but many others standards like RELAX NG [2], Schematron [3], DSD [4] etc. exist. Recently, the W3C has released the *XML Schema* standard [5, 6, 7], thought to replace DTD's with a more powerful and complete tool to describe XML document classes.

DTD, XML Schema and most other schema languages belong to the *grammar-based* schema language class. Schemas in such languages describe documents by defining allowed element and attribute names, setting *content models* for the elements and imposing restrictions on the text contained in elements and attributes. Schematron is a *rule-based* schema language, in which a schema is a collection of constraints that the documents have to fulfill. Further schema language concepts have been proposed.

DTD's were designed to describe weakly structured documents like books or web pages. Such documents don't need strict datatyping as they are interpreted by human readers. However, XML documents are more and more used in applications like business integration and data exchange. Here, heavily structured and strictly typed data is required as these documents are usually interpreted by computers. DTD's contain only rudimentary datatyping features, therefore new means of schema definition were necessary.

XML Schema tries to fill the gaps left by DTD's with a strong dataype concept, object oriented schema development concepts, namespace awareness and many more features designed to simplify the design of schemas.

1.2 Project Motivation and Goals

The goal of this diploma thesis is to develop a new, more compact and better readable syntax for XML Schema and to make this syntax useful by implementing the needed tools.

XML Schema comes with a bunch of modelling concepts, an advanced type inheritance concept and a quite complete datatype library. This, as well as the use of an XML syntax have lead to a complex and verbose language described in a very formal and almost unreadable standard. To describe a simple element *digit* that can contain the values 0 to 9, the following construct is needed:

```
<xs:element name="digit">
  <xs:simpleType>
    <xs:restriction base="xs:nonNegativeInteger">
      <xs:maxInclusive value="9"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Therefore, more real-life schemas tend to become very large and difficult to read. Of course, XML schema documents are mostly interpreted by computers which don't care about verbosity, but to develop and debug schemas, a more dense and readable representation could be useful.

1.3 Solution approach

The XML Schema language is defined on the abstract *Schema Component* model which define a schema as a set of related components having certain properties. The XML elements and attributes described in the W3C XML Schema standard are just the standard representation of the Schema Components, but other representations could be used as well to describe an XML Schema instance. Therefore, an alternative Syntax for XML Schema must be able to represent all possible and allowed sets of Schema Components, but it does not have to reuse the structures used for the XML representation of XML Schema.

Main design goals for the definition of the compact syntax were:

- Good readability and compactness
- Reuse of well-known programming and schema language concepts
- Full compatibility to the XML Schema standard
- Definition in EBNF notation [8] to make automatic parser generation possible

To make the compact syntax useful, tools supporting its use have to be implemented. This includes mainly a software component that is able to validate XML instance documents against a compact syntax schema. Furthermore, conversion from the XML syntax to compact syntax and vice versa should also be possible.

The implementation of a validating parser would however go beyond the scope of this thesis. But as there are some implementations of parsers validating against an XML Schema, a schema in the compact syntax can first be converted to the XML Schema syntax and then used for document validation.

1.4 Related work

For RELAX NG, another schema language for XML documents, a compact non-XML syntax [9] has already been defined. The standard uses a BNF notation for the syntax definition and contains a mapping to the XML version of RELAX NG, which defines the semantics of the language.

Chapter 2

XML Schema

2.1 Introduction

The XML Schema standard consists of two parts, *Structures* [5] and *Datatypes* [6]. *Structures* contains the parts of XML Schema that deal with the structure of XML documents, basically describing the allowed element trees and attribute sets. *Datatypes* contains the parts of XML Schema that impose restrictions on the character sequences in text nodes and attribute values. Among others, the XML Schema standard defines the following concepts:

Schema Components An abstract data model for the various components of a schema. Components have *properties*, which are either of a specific datatype, or components themselves. The standard further defines the *constraints* that apply to these properties to form a valid component. An XML Schema is a set of Schema Components satisfying all component constraints.

XML Representation A definition of the XML elements and attributes used to represent Schema Components. Various constraints also apply to the XML representation of XML Schema in order to guarantee that only valid schema component sets can be described.

Validity assessment The rules used to assess validity of an XML *Information Set* using a set of Schema Components. The XML Information Set [10] is a standardized data model of XML documents.

Information set contributions Various *augmentations* added to the information set of a validated instance document. The added information contains for example a node's validity, the type that has been used for validation, etc. These augmentations are called Post-Schema-Validation Infoset (PSVI).

Validity assessment and the related *PSVI* are essential for the implementation of validating parsers, but not as much for the representation of XML Schemas, therefore there is no further discussion of these topics. The *Schema Components* are described more in-depth in section 2.4, while the *XML Representation* is used for the definition of the compact syntax in chapter 3.

2.2 Using XML Schema

To benefit from the features of document description that XML Schema offers, a *schema-validating parser* is needed. This software reads an XML document, looks for an associated schema and validates the document against this schema. There are several parsers around that support XML Schema validation, for example Apache's Xerces [11], XSV [12] or MSXML [13].

Before an XML Schema can be used for validation, it should be checked whether it is correct and satisfies all constraints defined in the W3C standard. Because there are a high number of constraints, some quite intricately worded, the use of software tools for schema checking is inevitable. Currently, the most stable and complete implementation is SQC [14], but Xerces can also be used for schema checking.

2.3 Theoretical Background

XML documents form a tree of elements. Wellformed XML documents allow elements to have any number of child elements as well as any number of child text nodes. Elements can further contain an unordered collection of attributes, which consist of a name and a string value. Grammar-based schema languages are used to describe:

1. A collection of elements names that can be used in a document.
2. The set of allowed attribute names for every element.
3. Allowed sequences of children elements for every element (the content model).
4. Allowed string values for text nodes and attribute values.
5. Additional constraints on values like uniqueness or relations to other values (identity constraints).

In theory, such trees of elements are described by *tree grammars* [15]. Different variants of tree grammars exist with different expressiveness concerning the set of describable trees. In [16], a *Taxonomy of XML Schema Languages using Formal Language Theory* has been established comparing the most important Schema Languages regarding their formal expressiveness. The following text summarizes the XML Schema related chapters of this paper. Note that not all features of XML

```
<document>
  <pagesize form="A4" orientation="portrait"/>
  <paragraph font="Courier">
    This is just text, nothing <em>but</em> text.
  </paragraph>
  <paragraph>
    Some more nonsense
  </paragraph>
</document>
```

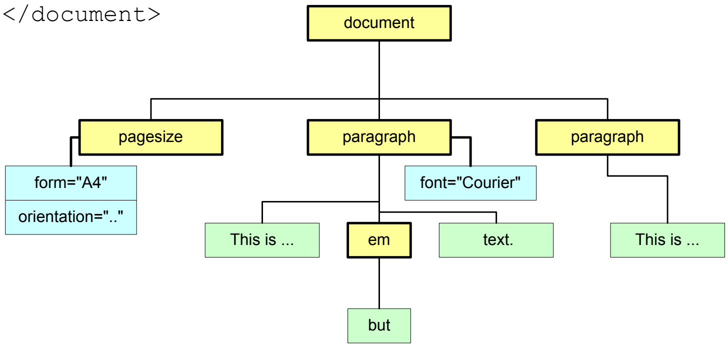

Figure 2.1: Example for element tree

Schema can be covered by this view, identity constraints for example can not be expressed with a tree grammar.

A general form of a tree grammar is the *Regular Tree Grammar* which is defined as a 4-tuple $G = (N, T, S, P)$ where:

- N is a finite set of *non-terminals*.
- T is a finite set of *terminals*.
- S is a set of *start symbols*, where $S \subset N$.
- P is a finite set of *production rules* of the form $n \rightarrow t\ r$, where $n \in N$, $t \in T$ and r is a *regular expression* over N, the *content model*.
- There are no two *production rules* having both the same *non-terminal* and *terminal*.

When used for XML document trees, the terminals represent *element names* while the non-terminals represent *element types*. The document element is the terminal of one of the production rules for the start symbols. Attributes and XML constructs like processing instructions are omitted in this view.

XML Schema implements a class of tree grammars referred to as *Single-Type Tree Grammar*. A Single-Type Tree Grammar is a Regular Tree Grammar with the following restrictions:

1. The content models of all production rules do not contain different non-terminals to which the same terminal is associated in a production rule.
2. All start symbol non-terminals have different associated terminals.

In XML Schema, the non-terminals are the *complex types*, while the terminals are the *elements*. Restriction one is the equivalent to Schema Component Constraint: Element Declarations Consistent, while the second restriction is explained in Names and Symbol Spaces within the XML Schema standard.

Not all schema languages implement the same tree grammar class. DTD's [1] for example implement an even more restrictive *Local Tree Grammar* which requires different terminals for different non-terminals in all production rules. DTD does not differentiate between terminals and non-terminals as it does not have the concept of *element types*. RELAX NG [2] however has the full expressiveness of a regular tree grammar.

2.4 The Schema Components

Figure 2.2 shows a simplified view of the component structure of an XML Schema. Main components within a schema are the following:

Element Declares an element name and associates it with a type.

Attribute Declares an attribute name and sets a datatype for its value.

Complex Type Defines an element type, using a content model that specifies the set of allowed child element sequences as well as the allowed attributes. Complex types can also specify *mixed* content models allowing both elements and text nodes as children.

Simple Type Defines a datatype for text nodes and attribute values.

Additional Schema Components are needed for the definition of content models, the **Model Group** and **Particle** components. They occur within Complex Type components. When using the DTD-style notation for content models, for example:

$$(a \mid b? \mid c* \mid (d , e+)?)$$

a Particle component corresponds to one element name including the *occurence specifiers* like +, * and ?. Several Particles connected with *compositors* like , or | and grouped by parentheses form a Model Group component.

The definition of Simple Types occurs through the use of **Facets**, which restrict the set of allowed values in different dimensions. Existing Simple Types can also be

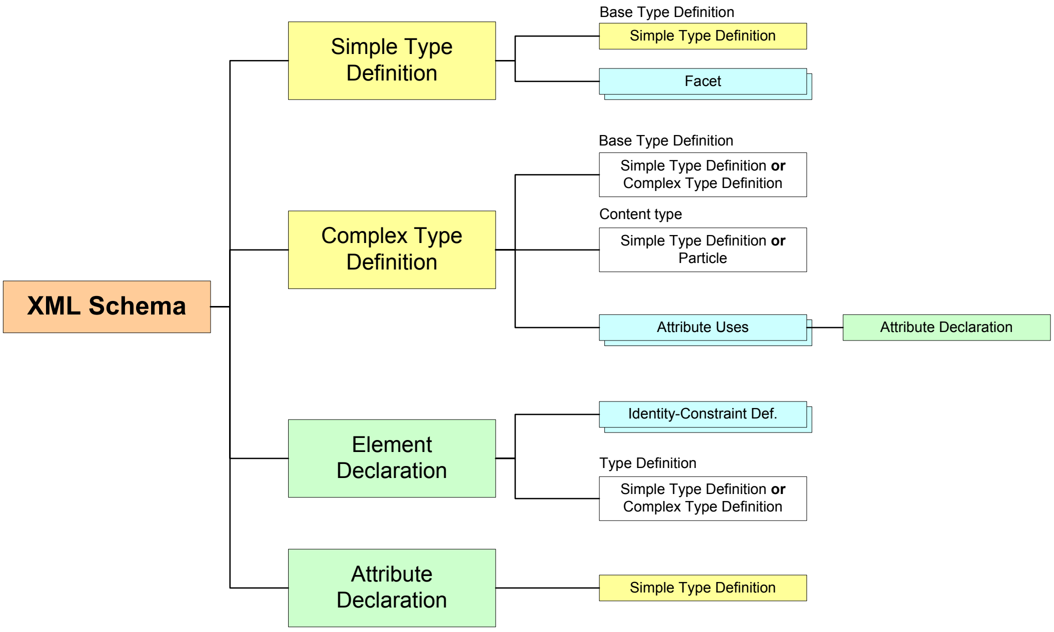

Figure 2.2: Schema Components (simplified)

combined using lists or unions. Some Facets impose *lexical* restrictions on datatypes, regarding values as sequences of characters. Others restrict *value* spaces, regarding values as instance of a certain datatype. For example the **Pattern** facet restricts values on a lexical basis using a regular expression, while the **MinInclusive** facet restricts values on a value basis using a numeric lower border.

Some other Schema Components exist for supportive purposes. The **Group** component allows the definition of reusable content models, while the **Attribute-Group** component can be used to define collections of attributes. **Identity Constraints** can be attached to elements to set integrity constraints. There is also support for the DTD legacy **Notation**.

Chapter 3

Compact Syntax Definition

This chapter describes the compact syntax for XML Schema. It starts with a general overview of the syntax design, followed by a more detailed description of the compact syntax features.

The compact syntax is defined using the XML representation of XML Schema. As the XML standard itself uses the *Schema Component* model to define XML Schema, it would be an obvious approach to define the compact syntax directly using the *Schema Components*. Structurally, however, the compact syntax is much closer to the XML representation, which makes the definition of the compact syntax much easier. Furthermore, the definition of the compact syntax is also useful for XML Schema users that don't know the Schema Components model (which is the vast majority of XML Schema users).

3.1 Design Principles

An XML schema is basically a collection of Schema Components. These components can refer to other components and they can contain components themselves. The Schema Components can be divided into several categories. A whole schema is described by the **Schema** component. This component contains the *top-level* components. There are several components that can occur at the top-level of a schema. Common to all of them is that they are *named*, unlike certain other components that cannot appear at top-level. The top-level components are the following:

> Element, Attribute, Simple Type, Complex Type, Model Group Definition, Attribute Group, Notation

Note that the **Complex Type** and the **Simple Type** components can also appear unnamed (anonymous) inside other schema components. There are some more components which only occur within other components, the *inner components*:

Model Group, Particle, Wildcard, Identity Constraint, Attribute Use, Facet (different Facet components exist).

The main design principle was to represent the top-level components using a regular syntax of the form:

options **component-type** name *extensions* { *inner components* };

Options simply set or unset a specific component property. They are used for boolean and fixed-value list properties. In the XML representation of XML Schema[1], they mostly appear as attributes with a boolean or enumerated datatype. *Extensions* represent properties with a string, name, or reference datatype. In XML Schema, they appear as attributes with a name or string datatype. The *inner components* are the equivalent to component reference properties in the Schema Components and mostly appear as nested elements in XML Schema.

Some of the non-top-level components use the same syntax, whereas others use non-regular constructs. However, the overall structure is always the same: A schema is made up of a list of components, which can contain blocks of inner components. A block is delimited by curly brackets. Components can optionally be terminated with a semicolon.

Another main design goal was to reuse well-known syntactical constructs to simplify the use of the compact syntax for new users. The DTD *content model* notation is certainly the best example. This notation in regular expression style is well-known and concise for the description of element content. Other notation reuses include the *interval* notation used for occurrence specifiers, and the length and range facets. Instead of using two elements or attributes as in XML Schema, it is much clearer and shorter to use a mathematical notation for intervals.

Some syntax elements were borrowed from programming languages like C or Java. The grouping of multiple components with curly brackets is an example, as well as the *options* and *extensions* constructs. Finally, the syntax for the pattern facet was inspired by the scripting language Perl.

3.2 Schemas and Schema Options

The following grammar definition for the compact syntax uses the following conventions: Non-Terminals appear *italic* and terminals are in **bold-face**. Optional components are enclosed in square brackets [ ], a star * is used for zero or more repetitions and the plus + denotes one or more repetitions. The vertical bar | separates alternatives. Parentheses are used for grouping.

[1] In the following text, the term XML Schema is mainly used as a synonym for the XML syntax of XML Schema, while XSCS or compact syntax are used for the newly defined compact syntax.

3.2.1 Schemas as a whole

$$
\begin{aligned}
schema &= [\ schemaOption\ ]*\ [\ schemaInclude\ ]* \\
&\quad [\ schemaBody\ ]+
\end{aligned} \tag{3.1}
$$

$$
\begin{aligned}
schemaOption &= targetNamespace \\
&\quad |\ namespace \\
&\quad |\ blockFinalDefault \\
&\quad |\ elementDefault \\
&\quad |\ attributeDefault \\
&\quad |\ version
\end{aligned} \tag{3.2}
$$

$$
\begin{aligned}
schemaInclude &= include \\
&\quad |\ import \\
&\quad |\ redefine
\end{aligned} \tag{3.3}
$$

$$
\begin{aligned}
schemaBody &= simpleType \\
&\quad |\ complexType \\
&\quad |\ element \\
&\quad |\ attribute \\
&\quad |\ group \\
&\quad |\ attributeGroup \\
&\quad |\ notation
\end{aligned} \tag{3.4}
$$

The *schema* production is the start symbol for the compact syntax. A sequence of tokens matching this production corresponds to an XML file having `xs:schema` as its document element.

SchemaOptions are used to set several attributes of the `xs:schema` element, while the productions in *schemaInclude* and *schemaBody* correspond to the XML Schema elements with the same names.

Annotations are documentation comments using the syntax `/* text... */` and can appear between every token. Depending on their position, they are mapped to a component. The generated `xs:annotation` elements contain a `xs:documentation` element containing the annotation text as a text node. XML markup inside annotations or custom attribute values are not supported by the compact syntax.

Annotations appearing before or inside *schemaOption* productions or after the last *schemaBody* production will become direct children of the `xs:schema` element. All other annotations are mapped to the current or next following component.

Some XML-specific constructs that can appear in XML Schema documents do not have an equivalent in the compact syntax. XML comments, an internal DTD subset or processing instructions will be lost when the XML syntax is translated to the compact syntax.

3.2.2 Schema Options

$$\begin{aligned}
targetNamespace &= \textbf{targetNamespace}\ URI\ [\ \textbf{;}\ ] && (3.5)\\
namespace &= \textbf{namespace}\ [\ Name\ ]\ URI\ [\ \textbf{;}\ ] && (3.6)\\
blockFinalDefault &= \textbf{default}\ qualifier\ [\ \textbf{,}\ qualifier\ ]*\ \ [\ \textbf{;}\ ] && (3.7)\\
elementDefault &= \textbf{elementDefault}\ qualifier\ [\ \textbf{;}\ ] && (3.8)\\
attributeDefault &= \textbf{attributeDefault}\ qualifier\ [\ \textbf{;}\ ] && (3.9)\\
version &= \textbf{version}\ String\ [\ \textbf{;}\ ] && (3.10)
\end{aligned}$$

All schema options are used to set attribute values of the `xs:schema` element. They do not represent schema components themselves, but they are used as default values for some component properties.

Compact Syntax	XML Syntax
targetNamespace *URI*	`targetNamespace="URI"`
namespace *Name URI*	`xmlns:Name="URI"`
namespace *URI*	`xmlns="URI"`

Table 3.1: Namespace options

The *targetNamespace* option (table 3.1) sets the target namespace of the schema. By default, the target namespace will also be declared as the default namespace of the schema, but this can be overridden by explicitly specifying a prefix for the target namespace using the *namespace* option.

Namespace options (table 3.1) can be used to declare additional namespace prefixes. As default, the XML Schema namespace is mapped to the prefix `xs`, this can be changed by defining another prefix for the XML Schema namespace. Note that with the compact syntax, the only possibility to declare namespace prefixes is within the `xs:schema` element. All prefixes used throughout the schema must be declared on the top-level. It is an error for a component name or reference, a type reference or an XPath to contain QNames with undeclared prefixes.

The *default* option (table 3.2) sets values for the `finalDefault` and `blockDefault` attributes. Any combination of values is allowed, but if *final* or *block* is specified, the `#all` value will always be generated.

Compact Syntax	XML Syntax
default final	`finalDefault="#all"`
default final-extension	`finalDefault="extension"`
default final-restriction	`finalDefault="restriction"`
default block	`blockDefault="#all"`
default block-extension	`blockDefault="extension"`
default block-restriction	`blockDefault="restriction"`
multiple values can be specified comma-separated	

Table 3.2: Final and block default settings

Compact Syntax	XML Syntax
elementDefault qualified	`elementFormDefault="qualified"`
elementDefault unqualified	*nothing*
attributeDefault qualified	`attributeFormDefault="qualified"`
attributeDefault unqualified	*nothing*

Table 3.3: Form default settings

The *elementDefault* and *attributeDefault* options (table 3.3) are used to control the *target namespace* property of non-global element and attribute components. Applicable values are *qualified* and *unqualified*. They correspond to the `attributeFormDefault` and `elementFormDefault` attributes in XML Schema. Unlike in XML Schema, *elementDefault* defaults to *qualified* while *attributeDefault* defaults to *unqualified*. The defaults have been changed due to the fact that most schema editors use these settings.

Compact Syntax	XML Syntax
version *String*	`version="String"`

Table 3.4: Version specification

A *version* option (table 3.4) can be used with any string as its value. This is for user convenience only and corresponds to the `version` attribute in XML Schema.

3.2.3 Import/Include statements

$$include \;=\; \textbf{include}\; URI\; [\; \textbf{;}\;] \tag{3.11}$$

$$import \;=\; \textbf{import}\; URI\; \textbf{namespace}\; URI\; [\; \textbf{;}\;] \tag{3.12}$$

$$redefine = \textbf{redefine}\ URI\ [\ \textbf{\{}\ [\ simpleType \mid complexType \mid group \mid attributeGroup\ ]*\ \textbf{\}}\ ]\ [\ \textbf{;}\ ] \tag{3.13}$$

The *import*, *include* and *redefine* statements (table 3.5) correspond to the elements with the same name in XML Schema. *Include* simply includes another schema that uses the same (or no) target namespace. *Redefine* does the same, except that simple types, complex types, groups and attribute groups can be redefined inside the *redefine* component. *Import* is used to compose schemas with different namespaces.

Compact Syntax	XML Syntax
include *URI*	`<include schemaLocation="URI"/>`
import *URI* **namespace** *URI*	`<import schemaLocation="URI" namespace="URI"/>`
redefine *URI* { *redefinitions* }	`<redefine schemaLocation="URI"> redefinitions </redefine>`

Table 3.5: Import, Include and Redefine

3.3 Describing Structures

3.3.1 Common Structures

$$\begin{aligned} qualifier = {} & \textbf{final} \mid \textbf{final-restriction} \mid \textbf{final-extension} \mid \textbf{final-list} \\ & \mid \textbf{final-union} \mid \textbf{block} \mid \textbf{block-substitution} \\ & \mid \textbf{block-extension} \mid \textbf{block-restriction} \\ & \mid \textbf{qualified} \mid \textbf{unqualified} \\ & \mid \textbf{abstract} \mid \textbf{nillable} \\ & \mid \textbf{required} \mid \textbf{optional} \mid \textbf{prohibited} \end{aligned} \tag{3.14}$$

$$derivation = \textbf{extends}\ Name \mid \textbf{restricts}\ Name \tag{3.15}$$

$$substitution = \textbf{substitutes}\ Name \tag{3.16}$$

$$fixedDefault = \ = String \mid <= String \tag{3.17}$$

Qualifiers (table 3.6) set the values of attributes that are common to some schema components. Multiple *final* and *block* qualifiers can be specified with one

Compact Syntax	XML Syntax
final	`final="#all"`
final-extension *etc.*	`final="extension"` *etc.*
block	`block="#all"`
block-substitution *etc.*	`block="substitution"` *etc.*
qualified	`form="qualified"`
unqualified	`form="unqualified"`
abstract	`abstract="true"`
nillable	`nillable="true"`
required	`use="required"`
optional	`use="optional"`
prohibited	`use="prohibited"`

Table 3.6: Qualifiers

component, but *qualified* and *unqualified* as well as *required*, *optional* and *prohibited* exclude each other.

The *derivation*, *substitution* and *fixedDefault* extensions (table 3.7) set the values of some attributes with name or string values. The *derivation* extension further influences the derivation method used for a complex type derivation.

Compact Syntax	XML Syntax
extends *Name*	`<extension base="Name"> ... </extension>`
restricts *Name*	`<restriction base="Name"> ... </restriction>`
substitutes *Name*	`substitutionGroup="Name"`
= *String*	`fixed="String"`
<= *String*	`default="String"`

Table 3.7: Extensions

3.3.2 Elements

$$\begin{aligned} element \quad = \quad & [\ qualifier\ ]*\ \textbf{element}\ Name \\ & [\ substitution \mid derivation\ ]*\ [\ elementContent\ ] \\ & [\ fixedDefault\ ]\ [\ \textbf{;}\ ] \end{aligned} \tag{3.18}$$

$$elementShort \;=\; Name\ [\ \{\ Name\ \}\ ] \tag{3.19}$$

$$\begin{aligned} elementContent \;=\; & \{\ [\ anonSimpleType \mid anonComplexType \\ & \mid key \mid keyref \mid unique\ ]*\ \} \end{aligned} \tag{3.20}$$

An *element* component can appear either at top-level or within another *element* or *complexType* component. When used inside another component, its name must be referred from the *contentModel* of this component.

	qualifiers
global	final, final-extension, final-restriction, block, block-extension, block-restriction, block-substitution, nillable, abstract
local	block, block-extension, block-restriction, block-substitution, nillable, qualified, unqualified

	extensions
global	*substitution, derivation*
local	*derivation, fixedDefault*

Table 3.8: Allowed qualifiers and extensions for *element*

To set the type of the declared element, either a reference to an existing type, or an anonymous simple or complex type can be used. Considering the inner components of the element component, these alternatives are chosen as follows:

- If there is a *derivation* extension, an inner *contentModel*, inner *elements* or inner *attributes*, then an anonymous complex type is constructed. The `xs:element` element will therefore contain an `xs:complexType` element that is built using the rules described in section 3.3.4.
- Else if there is an inner *restriction* with facets, *union* or *list* component, then an anonymous simple type is built.
- Else if there is an inner *restriction* component without any facets, the base name of the restriction will be used as the value of the `type` attribute of `xs:element`.
- Else if there is nothing at all, the element will have neither a `type` attribute nor an inner type definition.

The *elementShort* component is a shortcut for *element* which can only appear within *contentModel* components (see 3.3.4). It consists of the element name and an optional second name in curly braces which defines a type reference. When no type reference is present, the given element name is interpreted as a reference to an existing local or global element declaration. With a type reference, an element using the given name and type is defined.

Compact Syntax	**XML Syntax**
element example	<element name="example"/>
element example { xs:string }	<element name="example" type="xs:string"/>
element test { xs:int { [1,5] } }	<element name="test"> <simpleType> <restriction base="xs:int"> <minInclusive value="1"/> <maxInclusive value="5"/> </restriction> </simpleType> </element>

element test2 {
(a{xs:string}, b{xs:integer})*
}
	<element name="test2">
	 <complexType>
	  <sequence maxOccurs="unbounded">
	   <element name="a"
	      type="xs:string"/>
	   <element name="b"
	      type="xs:integer"/>
	  </sequence>
	 </complexType>
	</element>

Table 3.9: Examples for *element*

3.3.3 Attributes

$$\begin{aligned} attribute \quad &= \quad [\ qualifier\ ]*\ \textbf{attribute}\ Name \\ &\quad\ [\ attributeContent\ ]?\ [\ fixedDefault\ ]\ [\ \textbf{;}\ ] \end{aligned} \tag{3.21}$$

$$attributeContent \quad = \quad \textbf{\{}\ [\ anonSimpleType\ ]\ \textbf{\}} \tag{3.22}$$

The *attribute* component can appear at top-level or inside *element*, *complexType*, or *attributeGroup* components. An `xs:attribute` element will be generated, either with a `type` attribute, or an anonymous `xs:simpleType` child. If there is no inner

	qualifiers
global	*none*
local	qualified, unqualified, prohibited, required, optional
	extensions
global	*fixedDefault*
local	*fixedDefault*

Table 3.10: Allowed qualifiers and extensions for *attribute*

type definition or reference, an attribute reference will be created for local *attribute* components.

The `type` alternative is chosen when the attribute component contains a *restriction* component without any facets. If there is a *restriction* component with facets, a *list* or *union* component, an anonymous simple type will be declared.

Compact Syntax	**XML Syntax**
attribute test { xs:string }	`<attribute name="test"` `type="xs:string"/>`

element ex {
xs:integer; attribute foo
}
	`<element name="ex">`
	`  <complexType>`
	`    <simpleContent>`
	`      <extension base="xs:integer">`
	`        <attribute ref="foo"/>`
	`      </extension>`
	`    </simpleContent>`
	`  </complexType>`
	`</element>`

Table 3.11: Examples for *attribute*

3.3.4 Complex Types

$$complexType \;=\; [\; qualifier\;]* \;\textbf{complexType}\; Name \; [\; derivation\;]\; [\; complexTypeContent\;]\; [\; \textbf{;}\;] \tag{3.23}$$

$$complexTypeContent \;=\; \textbf{\{}\; [\; anonComplexType \mid anonSimpleType\;]* \;\textbf{\}} \tag{3.24}$$

$$anonComplexType \;=\; contentModel \mid element \mid attribute \mid attributeWC \mid attributeGroup \tag{3.25}$$

The *complexType* component can appear only at top level. Complex types are declared using a collection of inner components, which will all be used to construct a `xs:complexType` element. These components can also show up in the *element* component to define an anonymous complex type.

To define complex types with simple content, the *restriction* component has to be used. A *derivation* extension must not be used, as the base type for the restriction or extension is set by the *restriction* component. A *restriction* component with facets defines a restriction of the given base type. In XML Schema, this corresponds to the `xs:restriction` element. When no facets are present, the given name is interpreted as the base type name for an extension (`xs:extension` in XML Schema). To enforce a restriction even if there are no facets, an empty pair of curly brackets has to be added after the base name.

When a *contentModel* component is present, or neither a *contentModel* nor a *restriction* is present, complex content will be chosen for the `xs:complexType` element. If a *derivation* extension is given, the produced complex type will be a restriction or extension of the given base type. These three cases are displayed in table 3.12.

Compact Syntax	**XML Syntax**

complexType ct1
{ *modelGroup attributes* }
	<complexType name="ct1">
	  modelGroup
	  attributes
	</complexType>

complexType ct2 extends ct1
{ *modelGroup attributes* }
	<complexType name="ct2">
	  <complexContent>
	    <extension base="ct1">
	      modelGroup
	      attributes
	    </extension>
	  </complexContent>
	</complexType>

complexType ct2 restricts ct1
{ *modelGroup attributes* }
	<complexType name="ct2">
	  <complexContent>
	    <restriction base="ct1">
	      modelGroup
	      attributes
	    </restriction>
	  </complexContent>
	</complexType>

Table 3.12: Complex content in complex types

Any *attribute*, *attributeGroup* or *attributeWC* components will be added inside the `xs:restriction`, `xs:extension` or `xs:complexType` elements as necessary.

$$contentModel = (\ \textbf{empty} \mid [\ \textbf{mixed}\ ]\ (\ modelGroup \mid groupRef\ )\ [\ occurrenceSpec\ ]\ )\ [\ \textbf{;}\ ] \tag{3.26}$$

$$occurrenceSpec = \textbf{?} \mid \textbf{*} \mid \textbf{+} \mid posIntRange \tag{3.27}$$

$$modelGroup = \textbf{(}\ [\ particle\ [\ compositor\ particle\ ]*\ ]\ [\ compositor\ ]\textbf{)} \tag{3.28}$$

$$compositor = \textbf{,} \mid \textbf{|} \mid \textbf{\&} \tag{3.29}$$

$$particle = (\ modelGroup \mid elementShort \mid groupRef \mid \textbf{\{}\ element\ \textbf{\}} \mid \textbf{\{}\ elementWC\ \textbf{\}}\ )\ [\ occurrenceSpec\ ] \tag{3.30}$$

A *contentModel* component is used to define valid element sequences. It can be either *empty*, or consist of a *modelGroup* or *groupRef*. If it is *empty*, no corresponding XML elements will be generated. A *groupRef* creates an `xs:group` element with the `ref` attribute set. The *groupRef* or *modelGroup* can be preceded by the `mixed` keyword to allow text nodes between child elements.

A *modelGroup* stands either for an `xs:sequence`, `xs:choice`, or `xs:all` element containing element declarations or references, group references, model groups, or element wildcards. The compositors are **,** for sequence, | for choice, and **&** for all. *ModelGroups* that do not contain a *compositor* (i.e., *modelGroups* with zero or one particle) default to `xs:sequence`. Additional *compositors* can be added in these cases to force `xs:choice` or `xs:all`.

A *particle* denotes one part of a content model, it can be either a choice or sequence model group, an element or group reference, or a local element declaration or element wildcard. Optionally, an *occurence specifier* (table 3.13) can follow to set the number of allowed repetitions of the particle. It defaults to one and exactly one repetition.

An *elementShort* particle can be used to refer or declare an element. If only a name is given, a reference to a locally declared or global element is assumed. An additional type name in curly brackets declares an element of this type. It is also possible to put full *element* declarations inside the content model, simply add curly braces around the element declaration component. To create a *group reference*, an **@** char has to be added before the group name. *Element wildcards* (see 3.5.3) are defined similar to inline *elements* using curly brackets.

Compact Syntax	XML Syntax
*	`minOccurs="0"` `maxOccurs="unbounded"`
?	`minOccurs="0"`
+	`maxOccurs="unbounded"`
$[n]$	`minOccurs="n" maxOccurs="n"`
$[n, m]$	`minOccurs="n" maxOccurs="m"`
$[n,]$	`minOccurs="n"` `maxOccurs="unbounded"`
$[, m]$	`maxOccurs="m"`

Table 3.13: Definition of the occurrence specifiers

Element declarations can also be added inside the *complexType* component. When constructing the content model, references to these elements will be replaced with the appropriate declaration. References that have no corresponding local element declaration will be treated as references to global elements.

Compact Syntax	XML Syntax

complexType ct3 { (a, b)+;
element a { string }
element b { integer }
}
	`<complexType name="ct3">`
	`  <sequence maxOccurs="unbounded">`
	`    <element name="a" type="string"/>`
	`    <element name="b" type="integer"/>`
	`  </sequence>`
	`</complexType>`

complexType ct4 { @grp+
attribute test { token }
}
	`<complexType name="ct4">`
	`  <group ref="grp" maxOccurs="unbounded"/>`
	`  <attribute name="test" type="token"/>`
	`</complexType>`

Table 3.14: Complex type examples

3.4 Describing Datatypes

3.4.1 Simple Types

$$\begin{aligned} simpleType \;\; = \;\; & [\; qualifier \;] * \;\textbf{simpleType}\; Name \\ & [\; simpleTypeContent \;]\; [\; \textbf{;} \;] \end{aligned} \tag{3.31}$$

$$simpleTypeContent \quad = \quad \textbf{\{} \; [\; anonSimpleType \;] \; \textbf{\}} \tag{3.32}$$

$$anonSimpleType \quad = \quad restriction \mid union \mid list \tag{3.33}$$

A *simpleType* component can appear only at top-level. Anonymous simple types however can appear also inside attributes, elements, and complex types.

$$\begin{aligned} restriction \quad = \quad & (\; Name \; [\; \textbf{\{} \; [\; facet \;] * \; \textbf{\}} \;] \\ & \mid \textbf{simpleType} \; \textbf{\{} \; anonSimpleType \; \textbf{\}} \; \textbf{\{} \; [\; facet \;] * \; \textbf{\}} \;) \; [\; \textbf{;} \;] \end{aligned} \tag{3.34}$$

$$union \quad = \quad \textbf{union} \; \textbf{\{} \; [\; anonSimpleType \;] + \; \textbf{\}} \; [\; \textbf{;} \;] \tag{3.35}$$

$$list \quad = \quad \textbf{list} \; \textbf{\{} \; anonSimpleType \; \textbf{\}} \; [\; \textbf{;} \;] \tag{3.36}$$

An anonymous simple type can be defined using either a *restriction*, *list* or *union* component. These components can themselves contain anonymous simple type definitions except for the first alternative of *restriction*.

The *restriction* component is the counterpart of the `xs:restriction` element. The leading *Name* corresponds to the `base` attribute, unless the second variant with an embedded simple type is used. In that case, the `xs:restriction` element contains an `xs:simpleType` element defining the base of the restriction. Any *facets* become child elements of the `xs:restriction` element. The case where only a name but no facets are given is treated special in some contexts, but not inside a *simpleType* component.

Union and *list* correspond to the XML Schema elements with the same name. Unions and lists contain simple type definitions which are either added to the `memberTypes` or `itemType` attributes, or attached as `xs:simpleType` child elements. When only a name is given (a *restriction* component without facets), it is interpreted as a type reference, otherwise a type definition is assumed.

3.4.2 Facets

$$fixed \quad = \quad \textbf{fixed} \mid \textbf{fixed-minimum} \mid \textbf{fixed-maximum} \tag{3.37}$$

$$\begin{aligned} facet \quad = \quad & [\; fixed \;] * \quad (\; lengthFacet \mid rangeFacet \\ & \mid patternFacet \mid enumFacet \mid whiteSpaceFacet \\ & \mid totalDigitsFacet \mid fractionDigitsFacet \;) \; [\; \textbf{;} \;] \end{aligned}$$

Compact Syntax	**XML Syntax**
simpleType int { integer }	<simpleType name="int"> <restriction base="integer/> </simpleType>

simpleType digit {
nonNegativeInteger { [,9] }
}
	<simpleType name="digit">
	  <restriction base="nonNegativeInteger">
	    <maxInclusive value="9"/>
	  </restriction>
	</simpleType>

simpleType intu {
union { integer;
token { ”undefined” } }
}
	<simpleType name="intu">
	  <union memberTypes="integer">
	    <simpleType>
	      <restriction base="token">
	        <enumeration value="undefined"/>
	      </restriction>
	    </simpleType>
	  </union>
	</simpleType>

Table 3.15: Simple Type examples

$$ \tag{3.38} $$

$$lengthFacet = \textbf{length} = (\; PosInt \mid posIntRange \;) \tag{3.39}$$

$$rangeFacet = numRange \tag{3.40}$$

$$patternFacet = / \; Pattern \; / \tag{3.41}$$

$$enumFacet = String \; [\; , \; String \;] * \tag{3.42}$$

$$whiteSpaceFacet = \textbf{whiteSpace} = (\; \textbf{preserve} \mid \textbf{collapse} \mid \textbf{replace} \;) \tag{3.43}$$

$$totalDigitsFacet = \textbf{totalDigits} = PosInt \tag{3.44}$$

$$fractionDigitsFacet = \textbf{fractionDigits} = PosInt \tag{3.45}$$

$$posIntRange = [\; (\; PosInt \; [\; , \; PosInt \;] \mid , \; PosInt \;) \;] \tag{3.46}$$

$$numRange = (\; [\mid (\;) \; (\; Number \; [\; , \; Number \;] \mid , \; Number \;) \; (\;] \mid \textbf{)} \;) \tag{3.47}$$

Facets are used to restrict simple types in various dimensions. Some facets can be fixed using the *fixed* keyword which prohibits further modifications to the facet in type restrictions. For the *lengthFacet* and the *rangeFacet* which can collect two XML Schema facets specifiying a lower and upper bounds, also the keywords *fixed-*

minimum and *fixed-maximum* exist.

The *lengthFacet* constrains the length of several datatypes. It can either be set to a fixed value, or a range of values can be given. For a fixed value, a `xs:length` facet is generated, while for the range variant, either `xs:minLength` or `xs:maxLength` or both are used. This facet can be fixed using the *fixed* keyword, which sets the `fixed` attribute of all generated facet elements to true. *Fixed-minimum*, and *fixed-maximum* can be used in combination with a range to only fix minimum or maximum.

The *rangeFacet* is the counterpart to the `xs:minInclusive`, `xs:minExclusive`, `xs:maxInclusive`, and `max:Exclusive` elements. Ranges have to be defined with mathematical interval notation using parentheses () for exclusive and brackets [] for inclusive bounds. The range facet can be applied for all ordered datatypes (see 3.64). The *fixed*, *fixed-minimum* and *fixed-maximum* keywords can be applied similar to the length facet.

Most datatypes can also be required to match a regular expression using the *patternFacet*. Regular expressions must be enclosed in slashes /. Pattern facets (`xs:pattern` in XML Schema) cannot be fixed.

To restrict a datatype to a list of enumerated values, the *enumFacet* has to be used. A comma-separated list of quoted values has to be specified. For every value specified, one `xs:enumeration` element will be generated. Enumeration facets cannot be fixed.

WhiteSpaceFacets control the normalization of string values. The three options *preserve*, *collapse*, and *replace* are available. A corresponding `xs:whiteSpace` element is generated. Whitespace facets can be fixed, but *fixed-minimum* or *fixed-maximum* may not be used.

TotalDigitsFacets and *FractionDigitsFacets* control the number of digits that datatypes derived from `xs:decimal` can have. A non-negative integer has to be specified, and the optional *fixed* keyword can be used. They correspond to the `xs:totalDigits` and `xs:fractionDigits` elements.

3.5 Other Features

3.5.1 Model Groups

$$group = \textbf{group}\ Name\ [\ \textbf{\{}\ [\ contentModel \mid element\ ] *\ \textbf{\}}\ ]\ [\ \textbf{;}\ ] \tag{3.48}$$

$$groupRef = @\ Name \tag{3.49}$$

The *group* component is used to define reusable content models. It can be used only at top-level. *Groups* can be referred to from the content model of a complex

Compact Syntax	XML Syntax
length=8	<length value="8"/>
length=[3,6]	<minLength value="3"/> <maxLength value="8"/>
length=[,9]	<maxLength value="9"/>
[2,200]	<minInclusive value="2"/> <maxInclusive value="200"/>
(2,]	<minExclusive value="2"/>
[,2000-12-02)	<maxExclusive value="2000-12-02"/>
/.*test.*/	<pattern value=".*test.*"/>
"A3","A4","A5"	<enumeration value="A3"/> <enumeration value="A4"/> <enumeration value="A5"/>
whiteSpace=preserve	<whiteSpace value="preserve"/>
totalDigits=8	<totalDigits value="8"/>
fractionDigits=0	<fractionDigits value="0"/>

Table 3.16: Facet examples

type using the *groupRef* component. A group that does not contain a content model implicitly contains an empty sequence model group. The corresponding XML Schema constructs are:

Compact Syntax	XML Syntax
group name { *modelGroup* }	<group name="name"> modelGroup </group>
@grp	<group ref="grp"/>
group name	<group name="name"> <sequence/> </group>

Table 3.17: Group examples

3.5.2 Attribute Groups

$$
\begin{aligned}
\mathit{attributeGroup} \quad = \quad & \textbf{attributeGroup}\ \mathit{Name}\ [\ \boldsymbol{\{}\ [\ \mathit{attribute} \mid \mathit{attributeWC} \\
& \mid \mathit{attributeGroup}\ ]+\ \boldsymbol{\}}\ ][\ \textbf{;}\ ]
\end{aligned} \tag{3.50}
$$

AttributeGroups define reusable sets of attributes for the use within complex type definitions. When the *attributeGroup* appears at top-level, it is interpreted as an attribute group definition, inside complex types or other attribute groups a reference is generated. The corresponding XML Schema constructs are:

Compact Syntax	**XML Syntax**

attributeGroup name { *attributes* }	`<attributeGroup name="name">`
	`  attributes...`
	`</attributeGroup>`

attributeGroup ref	`<attributeGroup ref="ref"/>`

Table 3.18: Attribute group examples

3.5.3 Wildcards

$$
\begin{aligned}
\textit{process} &= \textbf{lax} \mid \textbf{strict} \mid \textbf{skip} && (3.51)\\
\textit{wildcardNSDecl} &= \textbf{\#\#targetNS} \mid \textbf{\#\#other} \mid \textbf{\#\#local} \mid \textit{URI} && (3.52)\\
\textit{elementWC} &= [\ \textit{process}\ ]\ \textbf{any}\ [\ \textbf{namespace} \\
&\quad \textit{wildcardNSDecl}\ [\ \textbf{,}\ \textit{wildcardNSDecl}\ ]*\ ]\ [\ \textbf{;}\ ] && (3.53)\\
\textit{attributeWC} &= [\ \textit{process}\ ]\ \textbf{anyAttribute}\ [\ \textbf{namespace} \\
&\quad \textit{wildcardNSDecl}\ [\ \textbf{,}\ \textit{wildcardNSDecl}\ ]*\ ]\ [\ \textbf{;}\ ] && (3.54)
\end{aligned}
$$

Wildcards (table 3.20) define placeholders for arbitrary elements or attributes. Element wildcards (*elementWC*) must be used within a *contentModel*, they cannot be declared outside the content model like elements. Attribute wildcards are used in complex types or attribute groups. In XML, the following constructs are generated:

Compact Syntax	**XML Syntax**
any	`<any/>`
anyAttribute	`<anyAttribute/>`

Table 3.19: Wildcard examples

3.5.4 Identity Constraints

$$
\begin{aligned}
\textit{idConstrField} &= \textbf{field}\ \textit{XPath}\ [\ \textbf{,}\ \textit{XPath}\ ]*\ \textbf{in}\ \textit{XPath} && (3.55)\\
\textit{key} &= \textbf{key}\ \textit{Name}\ \textit{idConstrField}\ [\ \textbf{;}\ ] && (3.56)
\end{aligned}
$$

Compact Syntax	XML Syntax
lax	`process="lax"`
skip	`process="skip"`
strict	`process="strict"`
namespace ##targetNS	`namespace="##targetNamespace"`
namespace ##other	`namespace="##other"`
namespace ##local	`namespace="##local"`
namespace URI1, URI2	`namespace="URI1 URI2"`

Table 3.20: Wildcard options

$$\begin{aligned} keyref &= \textbf{keyref}\ Name \\ &\quad \textbf{refers}\ Name\ idConstrField\ [\ \mathbf{;}\ ] \qquad (3.57) \\ unique &= \textbf{unique}\ Name\ idConstrField\ [\ \mathbf{;}\ ] \qquad (3.58) \end{aligned}$$

Identity constraints can be used to define consistency constraints similar to the ID/IDREF(S) feature in DTDs. *Keys* can be used to define values that must be unique within the document and that have to exist, while *unique* constraints only require uniqueness. *Keyrefs* define values that must refer to an existing *key* value. XPaths are used to define which values — either attribute values or text nodes — are used for identity constraints. An additional XPath defines the location of these values.

Compact Syntax	XML Syntax
key key1 field *XPath1* in *XPath2*	<key name="key1"> <field xpath="XPath1"/> <selector xpath="XPath2"/> </key>
keyref ref1 refers key1 field *XPath3* in *XPath2*	<keyref name="ref1" refer="key1"> <field xpath="XPath3"/> <selector xpath="XPath2"/> </keyref>
unique un1 field *XPath4*, *XPath5* in *XPath2*	<unique name="un1"> <field xpath="XPath4"/> <field xpath="XPath5"/> <selector xpath="XPath2"/> </unique>

Table 3.21: Identity constraint examples

3.5.5 Notations

$$\mathit{notation} \quad = \quad \textbf{notation}\ \mathit{Name}\ \textbf{public}\ \mathit{String}\ \textbf{system}\ \mathit{URI}\ [\ \textbf{;}\ ] \tag{3.59}$$

Notations are supported for DTD backwards compatibility. A notation definition consists of a name, a public and a system identifier.

Compact Syntax	XML Syntax
notation not1 public "pubID" system "sysURI"	<notation name="not1" public="pubID" system="sysURI"/>

Table 3.22: Notation example

3.5.6 Literals

$$\mathit{Name} \quad = \quad \mathit{NCName}\ |\ \mathit{QName}\ |\ \backslash\ \mathit{NCName} \tag{3.60}$$

A *Name* is either a QName or NCName as defined in the XML Namespace Standard [17]. For names that are equal to any of the keywords (see table 3.23), a preceding backslash has to be added.

$$\mathit{String} \quad = \quad \textbf{"}\ [\ [\ \hat{}\ \textbf{"}\ \backslash\ \textbf{<nl>}\ \textbf{<cr>}\ \textbf{<ff>}\ ]\ |\ \backslash\textbf{"}\ |\ \backslash\backslash\ |\ \backslash\textbf{n}\ |\ \backslash\textbf{r}\ |\ \backslash\textbf{f}\ |\ \backslash\textbf{t}\ ]\ \textbf{"} \tag{3.61}$$

Strings are enclosed in double quotes. Quotes and backslashes inside the string must be escaped using a backslash. The XML special characters $<$ and **&** can be used literally. For **n**ewline, carriage **r**eturn, **f**orm feed and **t**abulator, the well-known escapes can be used.

$$\mathit{XPath} \quad = \quad \textbf{"}\ \mathit{Selector}\ \textbf{"} \tag{3.62}$$

The *XPaths* used in XML Schema are a subset of the XPath specification [18] defined in the XML Schema standard as the Selector production. XPaths must be enclosed in double quotes.

targetNamespace	attributeGroup	nillable	empty
namespace	anyAttribute	qualified	fixed
default	any	unqualified	fixed-minimum
elementDefault	notation	final	fixed-maximum
attributeDefault	key	final-extension	lax
version	keyref	final-restriction	strict
include	unique	final-list	skip
import	refers	final-union	length
redefine	field	block	whiteSpace
complexType	in	block-substitution	preserve
simpleType	restricts	block-restriction	collapse
union	extends	block-extension	replace
list	substitutes	required	totalDigits
element	public	optional	fractionDigits
attribute	system	prohibited	
group	abstract	mixed	

Table 3.23: Reserved keywords

$$PosInt \quad = \quad [\,\mathbf{0}-\mathbf{9}\,]+ \tag{3.63}$$

PosInt are positive Integers (including zero), with no leading + allowed.

$$\begin{aligned} Number \quad = \quad & NumberStart\ [\ NumberChar\ ]\ * \\ & \mid \mathbf{INF} \mid \mathbf{-INF} \mid \mathbf{NaN} \end{aligned} \tag{3.64}$$

$$NumberStart \quad = \quad \mathbf{0}-\mathbf{9} \mid \mathbf{+} \mid \mathbf{-} \mid \mathbf{.} \mid \mathbf{P} \tag{3.65}$$

$$NumberChar \quad = \quad \mathbf{0}-\mathbf{9} \mid \mathbf{+} \mid \mathbf{-} \mid \mathbf{.} \mid \mathbf{e} \mid \mathbf{E} \mid \mathbf{T} \mid \mathbf{Z} \mid \mathbf{Y} \mid \mathbf{M} \mid \mathbf{D} \mid \mathbf{H} \mid \mathbf{S} \tag{3.66}$$

Number can be a literal value of all the XML Schema datatypes for which the range facets *minExclusive*, *maxExclusive*, *minInclusive*, and *maxInclusive* can be applied. This includes the date, time, dateTime, duration and all gregorian calendar[2] types, the decimal type, and the double and float types.

[2]gYearMonth, gYear, gMonthDay, gMonth, gDay

$$URI \quad = \quad \text{"}\, anyURI \,\text{"} \tag{3.67}$$

URIs are strings that are valid literals of the anyURI type as defined in the XML Schema datatypes standard.

$$Pattern \quad = \quad /\; regExp \;/ \tag{3.68}$$

Patterns are strings that are valid literals of the regExp production in the XML Schema datatypes standard. As they are enclosed with slashes, any slash inside the regular expression has to be escaped using a backslash.

Chapter 4

Implementation

4.1 Introduction

As mentioned in the introduction, the implementation of a schema-validating XML parser using compact syntax schemas would go beyond the scope of this thesis. Therefore, a schema in compact syntax has to be converted to XML syntax before it can be used for validation.

Thus, the software to be implemented has two main tasks. It must be able to convert a schema from compact syntax to XML syntax to make it useful for validation. Also the conversion from XML syntax to compact syntax should be possible to allow reuse of existing schemas with the compact syntax.

4.2 Evaluation

4.2.1 Compact Syntax to XML

As most of the XML-related tools are written in the Java language, no evaluation of XML tools in other programming languages was made. Therefore, the most important decision was the choice of a parser generation tool. Several packages are available for Java, mainly differing in the following characteristics:

- The supported grammar classes.
- Possibilities to embed parser action code.

Three software packages have been evaluated: SableCC [19], JavaCC [20], and JFlex/CUP [21, 22]. Table 4.1 shows the most important features of each product.

While SableCC and CUP support *LALR* parsing, JavaCC supports *LL* grammars. The main difference between those two parsing methods is the way that

	SableCC	**JavaCC**	**JFlex/CUP**
Grammar Class	LALR	LL(k)	LALR
Action Embedding	syntax tree generation	direct	direct

Table 4.1: Parser generation tools

productions are matched during parsing. More information about parsing can be found in [23].

LL parsing, also known as top-down or recursive parsing, examines the next or the next few token and then decides to which production or alternative it has to branch. The generated code contains a method for each production. All these methods consume the next token and then call the appropriate production method. Therefore, an LL grammar must be unambiguous at every *choice point*. This means that at every point in the grammar where the parser has to choose which production or alternative to follow, the next (or the few next in the case of LL(k) grammars) token must be enough information to decide where to branch to.

LALR parsing, on the other hand, is based on a finite automaton using a table of parser states. This table contains basically the next state for every combination of parser state and input token. The LALR grammar class is less restrictive than the LL class, because different productions can start with the same tokens and split up later. But for LALR grammars there are other constraints, because the generation of an unambiguous parser state table must be possible.

Considering the embedding of parser actions, SableCC differs strongly from the other two solutions. It does not allow direct action embedding, but rather constructs an *abstract syntax tree*, a tree-like object model for the parsed file. The advantage of this concept is that grammar and code can be cleanly separated. However, a Java class has to be generated for every alternative in every production, which can lead to hundreds of generated classes for more complex grammars.

JavaCC and CUP require action code to be directly embedded into the grammar file, which leads to larger and more complex grammar input files. However, it also simplifies the execution of user code during parsing. LL parser generators like JavaCC come with the major advantage that user code can be executed already at the *start* of productions, unlike LALR parsers, where user code can be added only at the *end* of a production. This is due to the fact that LALR parsers do not know in which production they currently are until the end of a production is reached.

Finally, it was decided to use the JavaCC parser, mainly because of the simpler concept of action embedding, which makes a step-by-step generation of the schema much easier. The grammar for the compact syntax is not LL(1), but JavaCC offers flexible means to specifiy expanded token lookahead at critical choice points.

4.2.2 XML to Compact Syntax

For the conversion from XML Schema to compact syntax, basically two implementation alternatives have been considered. A solution using an XSLT stylesheet (see Figure 4.1) has been evaluated as well as a DOM-based Java component (see Figure 4.2).

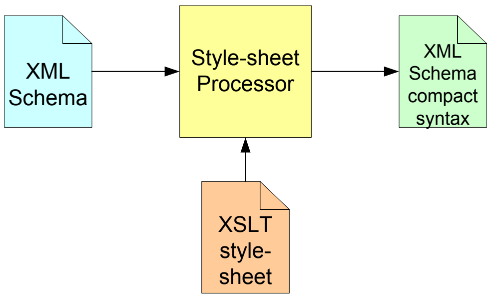

Figure 4.1: Conversion using an XSLT style-sheet

XSLT style-sheets [24] are an XML application that defines a transformation of an XML document into another XML, HTML, or text document. An XSLT processor like Saxon [25] can be used to apply the transformation to documents.

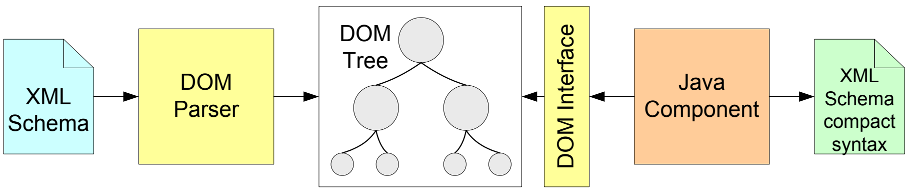

Figure 4.2: Conversion using a Java component

The Document Object Model (DOM) [26] is an API that offers a tree view of an XML document to the programmer. All parts of an XML document can be accessed and modified through Java methods[1]. The DOM specification defines *interfaces*, when using it an actual *implementation* like Apache's Xerces [11] has to be used. The pros and cons of both solutions are:

- XSLT offers a better interface to XML documents through the use of XPath expressions for XML document node tree addressing. Document tree traversal as in DOM[2] is not needed.

[1]many other programming language bindings exist for the DOM
[2]DOM Level 3 (W3C working draft at time of this writing) also offers XPath traversal.

- String manipulation as needed for the escaping of special characters is difficult in XSLT. The Java language offers more comfort here.
- Several algorithms, for example the decision whether an element will be declared inside the content model or not (see 3.3.4), are difficult to implement in XSLT due to the lack of modifiable variables.
- XSLT style-sheets are usually shorter and easier to maintain as a DOM-based solution.
- XSLT lacks software engineering support for larger projects and it is tedious to debug, partly due to its use of implicit type conversions.

The first experiments with the compact syntax have been made using XSLT, however it has become clear that a sensible implementation will be difficult in XSLT. Especially the problem of string escaping, but also some other issues like namespace control for attributes (see 3.2.2) finally led to the Java-based approach.

4.3 Implementation Design

4.3.1 Overview

The compact syntax parser consists of two components, the generated parser class, and a class that generates the DOM representation in XML Schema syntax. When converting from compact syntax to XML, a DOM tree of the schema is first generated, and then written to a file using a standard DOM serializer module. From XML to compact, the process starts by parsing the XML Schema file using a standard DOM parser, and then handing over the generated DOM tree to the compact syntax serializer component.

All coding and tests have been conducted using the Xerces parser library, however other DOM implementations could be used as well. However, some code changes will be necessary because the DOM interface cannot *boot-strap* itself[3]. This means that there is no implementation independent way of creating a *DOMImplementation* object, which is needed to create new documents.

4.3.2 The Parser

Figure 4.4 shows the structure of the compact syntax parser. The parser, the lexer, and some helper classes are generated by JavaCC from the input file *XSCParser.jj*. This file contains the definition of all keywords and literals of the compact syntax, as well as the grammar itself, including the embedded user action code.

[3] DOM Level 3 offers the boot-strap feature.

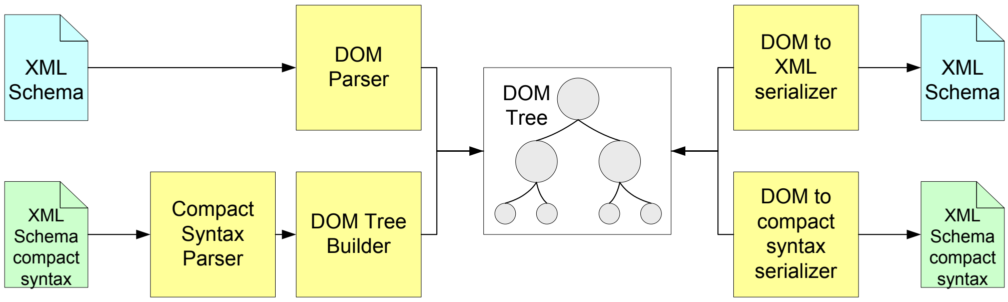

Figure 4.3: Implementation design overview

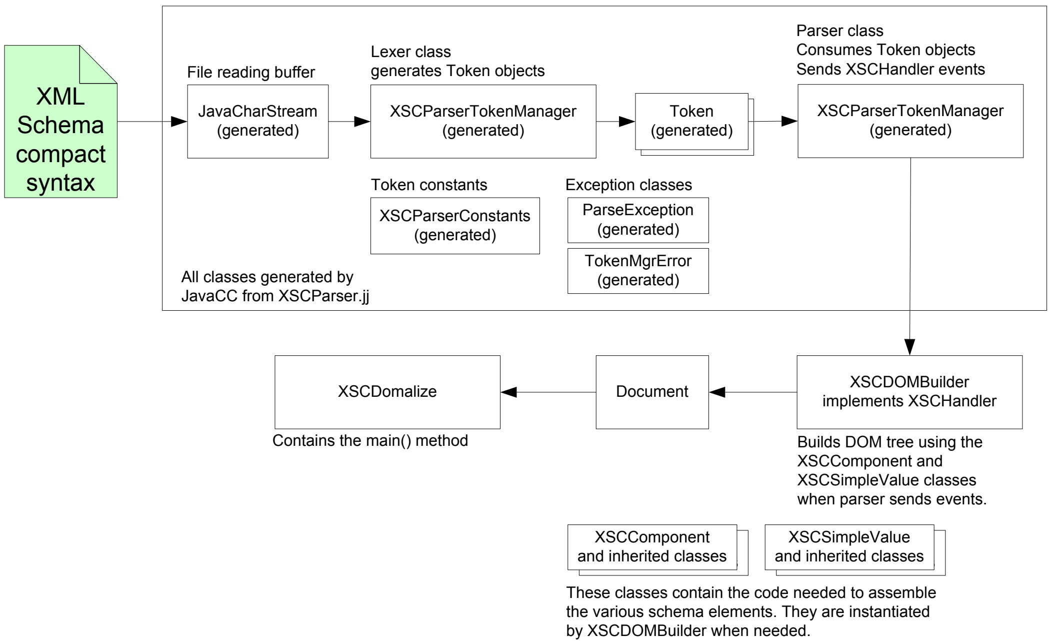

Figure 4.4: Class structure of the parser

The interface between the parser *XSCParser* and the DOM generator class *XSCDOMBuilder* is defined in the *XSCHandler* interface (see below). This interface defines a set of events that are emitted by the parser and processed by the DOM generator. The algorithm used by *XSCDOMBuilder* to assemble the DOM tree is described in the following paragraphs.

For every component that appears in the compact syntax, a class inherited from *XSCComponent* exists. These classes contain the code that interprets the compact syntax and creates the according XML Schema elements and attributes. A shortened version of the class *XSCComponent* is displayed below.

```
public interface XSCHandler {
   void startComponent(int type, XSCNameValue name);
   void endComponent();
   void extension(int type, XSCValue value);
   void qualifier(int type);
   void startBlock();
   void endBlock();
   void annotation(String text);
}

public abstract class XSCComponent extends XSCValue {
   protected NodeList fNodes;
   protected boolean fFrozen;

   public XSCComponent() {...}
   public NodeList getNodes() {...}
   abstract public int getType();
   public void property(XSCValue value,int subtype)
   abstract protected void property_(XSCValue value,int subtype);
   public void freeze() {...}
   abstract protected void freeze_();
}
```

To build up the DOM tree of a schema in compact syntax, *XSCDOMBuilder* handles the events generated by the parser as shown in the following pseudo code. The internal state of *XSCDOMBuilder* consists of a component stack, the current component, and the state (inside or outside component).

The contents of a component are assembled by multiple calls to the *property()* method, either with a certain value, to set an option of that component, or with a component, which becomes an inner component. Upon a call to the *freeze()* method, the elements and attributes of the component are assembled. Using method *getNodes()*, the parent component can get the created elements and attributes and add them to its own elements.

```
startComponent: push current to stack
                current = new component
                send saved qualifiers and annotations to current
                state = in component

endComponent:   freeze current component
                send current component to parent component
                current = pop parent from stack
                state = not in component
```

```
extension:      if in component
                   send to component
                else
                   error

qualifier:      if in component
                   send to component
                else
                   save for next component

annotation:     if in component
                   send to component
                else
                   save for next component

startBlock:     state = not in component

endBlock:       state = in component
```

4.3.3 Serializing the DOM tree

The serialization module consists of two classes, *XSCSerialize* and *XSCFormatter*. The serializer class traverses a given DOM tree and generates a sequence of tokens for the formatter class. The formatter inserts the appropriate whitespace and linebreaks and writes the resulting text to a file.

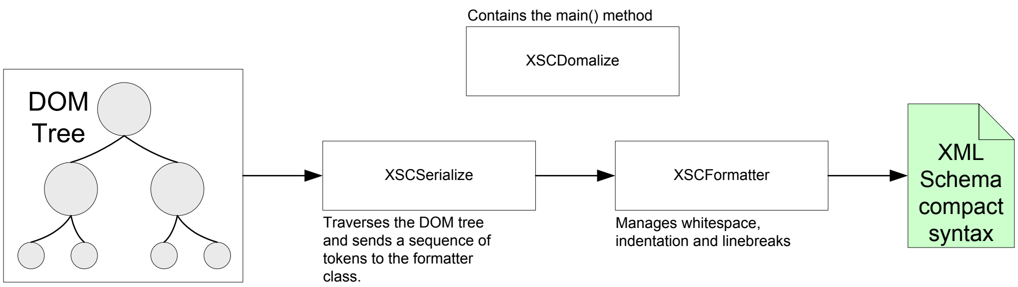

Figure 4.5: Class structure of the serializer

Chapter 5

Summary and Outlook

5.1 Summary

The compact syntax [27, 28] implements the full functionality of XML Schema. Restrictions apply only to annotations and namespace declarations, however they only limit user comfort, but not the expressive power of the compact syntax.

The thesis has shown that it is possible and valuable to invent non-XML syntaxes for some XML standards. XML has its strengths in the document and data exchange areas, but it is suboptimal for programming languages and similar tasks.

Some tests have been made using the implemented software to measure the size reduction that the compact syntax offers to the user. The XML Schema for XML Schema itself, as well as some other schemas available on the net have been compared in XML and compact syntax. To guarantee fairness, all annotations and XML comments have been stripped from the schemas before converting them to compact syntax. Character counts do not include whitespace characters.

	Lines			**Characters**		
Schema	XML	XSCS	Savings	XML	XSCS	Savings
datatypes.xsd	502	126	**74.9%**	14030	4520	**67.8%**
structures.xsd	936	315	**66.3%**	23820	9238	**61.2%**
dxl.xsd	2266	882	**61.1%**	67443	26977	**60.0%**
LandXML.xsd	3615	1512	**58.2%**	107960	42349	**60.8%**
logml.xsd	373	231	**38.1%**	12135	5330	**56.1%**
xgmml.xsd	323	179	**44.6%**	10419	4566	**56.2%**
spaceXML.xsd	1124	330	**70.6%**	28606	8760	**69.4%**

Table 5.1: Schema size reduction

A parser for the compact syntax and the serializer module have been successfully implemented. Some testing has been done with the software, however it is still in a prototype stadium. Annotation handling could be improved as well as namespace behaviour, which have been implemented in a rather simple way.

5.2 Outlook

Currently, compact syntax schemas have to be converted to the XML syntax before they can be used for validation. For a better integration of the compact syntax into the XML processing tool-chain, the compact syntax parser could be directly integrated into a validating parser.

The DOM based implementation has been chosen with regard to the integration in the Xerces parser. To validate XML documents, Xerces uses a DOM parser to read the associated schema, and then traverses this document to create an internal representation similar to the Schema Components data model. Therefore, only the code that loads schemas had to be altered to allow validation against compact syntax schemas.

The compact syntax itself could be extended, particularly in the field of annotation handling, in order to achieve one-to-one compatibility with XML Schema. Probably, further optimizations concerning user friendliness could be done by creating shortcuts for the most used parts of the language.

On the implementation side, many optimizations could be done to make the software more user friendly and complete. Switches could be added for various conversion options, as there is sometimes more than one possiblity for the translation from XML to compact syntax. One example are the content models, where there is the choice of embedding element declarations directly or to outsource them.

Extended error checking and warnings could be added to the compact syntax parser. At the moment, only a syntax check — done by the generated parser — and some error checking is done. The generated XML Schemas should always validate against the Schema for Schema, but no checks are done for the various constraints defined on the Schema Components.

Bibliography

[1] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0 (Second Edition). Technical report, W3C, October 2000.

[2] James Clark and Murata Makoto. RELAX NG Specification. Technical report, Oasis, December 2001.

[3] Rick Jelliffe. Resource Directory (RDDL) for Schematron 1.5. Technical report, Academia Sinica Computing Centre, September 2002.

[4] Anders Møller. Document Structure Description 2.0. Technical report, BRICS, 2003.

[5] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures. Technical report, W3C, May 2001.

[6] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes. Technical report, W3C, May 2001.

[7] Eric van der Vlist. *XML Schema*. O'Reilly & Associates, Inc., 2002.

[8] International Organization for Standardization. *Information Technology – Syntactic Metalanguage – Extended BNF*. ISO/IEC 14977, 1996.

[9] James Clark. RELAX NG Compact Syntax. Technical report, Oasis, November 2002.

[10] John Cowan and Richard Tobin. XML Information Set. Technical report, W3C, October 2001.

[11] Apache Software Foundation. *Xerces2*.

[12] HCRC, University of Edinburgh. *XSV*.

[13] Microsoft Corporation. *MSXML*.

[14] IBM Alphaworks. *XML Schema Quality Checker*.

[15] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. Available on: http://www.grappa.univ-lille3.fr/tata, 1997. release October, 1rst 2002.

[16] Makoto Murata, Dongwon Lee, and Murali Mani. Taxonomy of XML Schema Languages using Formal Language Theory. In *Extreme Markup Languages*, 2000.

[17] Tim Bray, Dave Hollander, and Andrew Layman. Namespaces in XML. Technical report, W3C, January 1999.

[18] James Clark and Steve DeRose. XML Path Language (XPath) Version 1.0. Technical report, W3C, November 1999.

[19] Sable Research Group, McGill University. *SableCC*.

[20] WebGain, Inc. *JavaCC*.

[21] *JFlex Scanner Generator for Java*.

[22] *CUP Parser Generator for Java*.

[23] Dick Grune and Ceriel Jacobs. Parsing Techniques - A Practical Guide. Available on: http://www.cs.vu.nl/~dick/PTAPG.html, 1998.

[24] James Clark. XSL Transformations (XSLT) Version 1.0. Technical report, W3C, November 1999.

[25] *SAXON - The XSLT Processor*.

[26] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. Document Object Model (DOM) Level 2 Core Specification. Technical report, W3C, November 2000.

[27] Erik Wilde and Kilian Stillhard. A Compact XML Schema Syntax. In *XML Europe*, 2003.

[28] Kilian Stillhard and Erik Wilde. XML Schema Compact Syntax (XSCS) Version 1.0. Technical report, Computer Engineering and Networks Laboratory (TIK), ETH Zürich, March 2003.

Appendix A

Complete Grammar

A.1 Structure

$$\begin{aligned} schema \quad &= \quad [\ schemaOption\ ]*\ \ [\ schemaInclude\ ]* \\ &\quad\quad [\ schemaBody\ ]+ \end{aligned} \tag{A.1}$$

$$\begin{aligned} schemaOption \quad = \quad & targetNamespace \\ & \mid namespace \\ & \mid blockFinalDefault \\ & \mid elementDefault \\ & \mid attributeDefault \\ & \mid version \end{aligned} \tag{A.2}$$

$$\begin{aligned} schemaInclude \quad = \quad & include \\ & \mid import \\ & \mid redefine \end{aligned} \tag{A.3}$$

$$\begin{aligned} schemaBody \quad = \quad & simpleType \\ & \mid complexType \\ & \mid element \\ & \mid attribute \\ & \mid group \\ & \mid attributeGroup \\ & \mid notation \end{aligned} \tag{A.4}$$

$$\begin{aligned}
targetNamespace &= \textbf{targetNamespace}\ URI\ [\ \textbf{;}\ ] && \text{(A.5)}\\
namespace &= \textbf{namespace}\ [\ Name\ ]\ URI\ [\ \textbf{;}\ ] && \text{(A.6)}\\
blockFinalDefault &= \textbf{default}\ qualifier\ [\ \textbf{,}\ qualifier\ ]*\ [\ \textbf{;}\ ] && \text{(A.7)}\\
elementDefault &= \textbf{elementDefault}\ qualifier\ [\ \textbf{;}\ ] && \text{(A.8)}\\
attributeDefault &= \textbf{attributeDefault}\ qualifier\ [\ \textbf{;}\ ] && \text{(A.9)}\\
version &= \textbf{version}\ String\ [\ \textbf{;}\ ] && \text{(A.10)}
\end{aligned}$$

$$\begin{aligned}
include &= \textbf{include}\ URI\ [\ \textbf{;}\ ] && \text{(A.11)}\\
import &= \textbf{import}\ URI\ \textbf{namespace}\ URI\ [\ \textbf{;}\ ] && \text{(A.12)}\\
redefine &= \textbf{redefine}\ URI\ [\ \textbf{\{}\ [\ simpleType \mid complexType\\
&\quad \mid group \mid attributeGroup\ ]*\ \textbf{\}}\ ]\ [\ \textbf{;}\ ] && \text{(A.13)}
\end{aligned}$$

$$\begin{aligned}
qualifier &= \textbf{final} \mid \textbf{final-restriction} \mid \textbf{final-extension} \mid \textbf{final-list}\\
&\quad \mid \textbf{final-union} \mid \textbf{block} \mid \textbf{block-substitution}\\
&\quad \mid \textbf{block-extension} \mid \textbf{block-restriction}\\
&\quad \mid \textbf{qualified} \mid \textbf{unqualified}\\
&\quad \mid \textbf{abstract} \mid \textbf{nillable}\\
&\quad \mid \textbf{required} \mid \textbf{optional} \mid \textbf{prohibited} && \text{(A.14)}
\end{aligned}$$

$$derivation = \textbf{extends}\ Name \mid \textbf{restricts}\ Name \qquad \text{(A.15)}$$

$$substitution = \textbf{substitutes}\ Name \qquad \text{(A.16)}$$

$$fixedDefault = \textbf{=}\ String \mid \textbf{<=}\ String \qquad \text{(A.17)}$$

$$\begin{aligned}
element &= [\ qualifier\ ]*\ \textbf{element}\ Name\\
&\quad [\ substitution \mid derivation\ ]*\ [\ elementContent\ ]\\
&\quad [\ fixedDefault\ ]\ [\ \textbf{;}\ ] && \text{(A.18)}
\end{aligned}$$

$$elementShort = Name\ [\ \textbf{\{}\ Name\ \textbf{\}}\ ] \qquad \text{(A.19)}$$

$$elementContent = \textbf{\{}\ [\ anonSimpleType \mid anonComplexType$$

$$| \ key \ | \ keyref \ | \ unique \ ] * \ \mathbf{\}} \tag{A.20}$$

$$attribute = [\ qualifier \ ] * \ \mathbf{attribute} \ Name \\ [\ attributeContent \ ]? \ [\ fixedDefault \ ] \ [\ \mathbf{;} \ ] \tag{A.21}$$

$$attributeContent = \mathbf{\{} \ [\ anonSimpleType \ ] \ \mathbf{\}} \tag{A.22}$$

$$complexType = [\ qualifier \ ] * \ \mathbf{complexType} \ Name \\ [\ derivation \ ] \ [\ complexTypeContent \ ] \ [\ \mathbf{;} \ ] \tag{A.23}$$

$$complexTypeContent = \mathbf{\{} \ [\ anonComplexType \ | \ anonSimpleType \ ] * \ \mathbf{\}} \tag{A.24}$$

$$anonComplexType = contentModel \ | \ element \ | \ attribute \\ | \ attributeWC \ | \ attributeGroup \tag{A.25}$$

$$contentModel = (\ \mathbf{empty} \\ | \ [\ \mathbf{mixed} \ ] \ (\ modelGroup \ | \ groupRef \ ) \\ [\ occurrenceSpec \ ] \ ) \ [\ \mathbf{;} \ ] \tag{A.26}$$

$$occurrenceSpec = \mathbf{?} \ | \ \mathbf{*} \ | \ \mathbf{+} \ | \ posIntRange \tag{A.27}$$

$$modelGroup = \mathbf{(} \ [\ particle \ [\ compositor \ particle \ ]* \ ] \ [\ compositor \ ]\mathbf{)} \tag{A.28}$$

$$compositor = \mathbf{,} \ | \ \mathbf{|} \ | \ \mathbf{\&} \tag{A.29}$$

$$particle = (\ modelGroup \ | \ elementShort \ | \ groupRef \ | \ \mathbf{\{} \ element \ \mathbf{\}} \\ | \ \mathbf{\{} \ elementWC \ \mathbf{\}} \ ) \ [\ occurrenceSpec \ ] \tag{A.30}$$

$$simpleType = [\ qualifier \ ] * \ \mathbf{simpleType} \ Name \\ [\ simpleTypeContent \ ] \ [\ \mathbf{;} \ ] \tag{A.31}$$

$$simpleTypeContent = \mathbf{\{} \ [\ anonSimpleType \ ] \ \mathbf{\}} \tag{A.32}$$

$$anonSimpleType = restriction \ | \ union \ | \ list \tag{A.33}$$

$$restriction = (\mathit{Name} \; [\; \mathbf{\{} \; [\; \mathit{facet} \;] * \; \mathbf{\}} \;] \; | \; \textbf{simpleType} \; \mathbf{\{} \; \mathit{anonSimpleType} \; \mathbf{\}} \; \mathbf{\{} \; [\; \mathit{facet} \;] * \; \mathbf{\}} \;) \; [\; \mathbf{;} \;] \tag{A.34}$$

$$union = \textbf{union} \; \mathbf{\{} \; [\; \mathit{anonSimpleType} \;] + \; \mathbf{\}} \; [\; \mathbf{;} \;] \tag{A.35}$$

$$list = \textbf{list} \; \mathbf{\{} \; \mathit{anonSimpleType} \; \mathbf{\}} \; [\; \mathbf{;} \;] \tag{A.36}$$

$$fixed = \textbf{fixed} \; | \; \textbf{fixed-minimum} \; | \; \textbf{fixed-maximum} \tag{A.37}$$

$$facet = [\; \mathit{fixed} \;] * \quad (\; \mathit{lengthFacet} \; | \; \mathit{rangeFacet} \; | \; \mathit{patternFacet} \; | \; \mathit{enumFacet} \; | \; \mathit{whiteSpaceFacet} \; | \; \mathit{totalDigitsFacet} \; | \; \mathit{fractionDigitsFacet} \;) \; [\; \mathbf{;} \;] \tag{A.38}$$

$$lengthFacet = \textbf{length} \; \mathbf{=} \; (\; \mathit{PosInt} \; | \; \mathit{posIntRange} \;) \tag{A.39}$$

$$rangeFacet = \mathit{numRange} \tag{A.40}$$

$$patternFacet = \mathbf{/} \; \mathit{Pattern} \; \mathbf{/} \tag{A.41}$$

$$enumFacet = \mathit{String} \; [\; \mathbf{,} \; \mathit{String} \;] * \tag{A.42}$$

$$whiteSpaceFacet = \textbf{whiteSpace} \; \mathbf{=} \; (\; \textbf{preserve} \; | \; \textbf{collapse} \; | \; \textbf{replace} \;) \tag{A.43}$$

$$totalDigitsFacet = \textbf{totalDigits} \; \mathbf{=} \; \mathit{PosInt} \tag{A.44}$$

$$fractionDigitsFacet = \textbf{fractionDigits} \; \mathbf{=} \; \mathit{PosInt} \tag{A.45}$$

$$posIntRange = \mathbf{[} \; (\; \mathit{PosInt} \; [\; \mathbf{,} \; \mathit{PosInt} \;] \; | \; \mathbf{,} \; \mathit{PosInt} \;) \; \mathbf{]} \tag{A.46}$$

$$numRange = (\; \mathbf{[} \; | \; \mathbf{(} \;) \; (\; \mathit{Number} \; [\; \mathbf{,} \; \mathit{Number} \;] \; | \; \mathbf{,} \; \mathit{Number} \;) \; (\; \mathbf{]} \; | \; \mathbf{)} \;) \tag{A.47}$$

$$group = \textbf{group} \; \mathit{Name} \; [\; \mathbf{\{} \; [\; \mathit{contentModel} \; | \; \mathit{element} \;] * \; \mathbf{\}} \;] \; [\; \mathbf{;} \;] \tag{A.48}$$

$$groupRef = \mathbf{@} \; \mathit{Name} \tag{A.49}$$

$$attributeGroup = \textbf{attributeGroup} \; \mathit{Name} \; [\; \mathbf{\{} \; [\; \mathit{attribute} \; | \; \mathit{attributeWC} \; | \; \mathit{attributeGroup} \;] + \; \mathbf{\}} \;] [\; \mathbf{;} \;] \tag{A.50}$$

$$\mathit{process} = \textbf{lax} \mid \textbf{strict} \mid \textbf{skip} \tag{A.51}$$

$$\mathit{wildcardNSDecl} = \textbf{\#\#targetNS} \mid \textbf{\#\#other} \mid \textbf{\#\#local} \mid \mathit{URI} \tag{A.52}$$

$$\begin{aligned}\mathit{elementWC} = {} & [\ \mathit{process}\ ]\ \textbf{any}\ [\ \textbf{namespace} \\ & \mathit{wildcardNSDecl}\ [\ \textbf{,}\ \mathit{wildcardNSDecl}\ ]{*}\ ]\ [\ \textbf{;}\ ]\end{aligned} \tag{A.53}$$

$$\begin{aligned}\mathit{attributeWC} = {} & [\ \mathit{process}\ ]\ \textbf{anyAttribute}\ [\ \textbf{namespace} \\ & \mathit{wildcardNSDecl}\ [\ \textbf{,}\ \mathit{wildcardNSDecl}\ ]{*}\ ]\ [\ \textbf{;}\ ]\end{aligned} \tag{A.54}$$

$$\mathit{idConstrField} = \textbf{field}\ \mathit{XPath}\ [\ \textbf{,}\ \mathit{XPath}\ ]*\ \textbf{in}\ \mathit{XPath} \tag{A.55}$$

$$\mathit{key} = \textbf{key}\ \mathit{Name}\ \mathit{idConstrField}\ [\ \textbf{;}\ ] \tag{A.56}$$

$$\begin{aligned}\mathit{keyref} = {} & \textbf{keyref}\ \mathit{Name} \\ & \textbf{refers}\ \mathit{Name}\ \mathit{idConstrField}\ [\ \textbf{;}\ ]\end{aligned} \tag{A.57}$$

$$\mathit{unique} = \textbf{unique}\ \mathit{Name}\ \mathit{idConstrField}\ [\ \textbf{;}\ ] \tag{A.58}$$

$$\mathit{notation} = \textbf{notation}\ \mathit{Name}\ \textbf{public}\ \mathit{String}\ \textbf{system}\ \mathit{URI}\ [\ \textbf{;}\ ] \tag{A.59}$$

A.2 Literals

$$\mathit{Name} = \mathit{NCName} \mid \mathit{QName} \mid \backslash\ \mathit{NCName} \tag{A.60}$$

$$\mathit{String} = \textbf{"}\ [\ [\ \hat{}\ \textbf{"}\ \backslash\ \textbf{<nl>}\ \textbf{<cr>}\ \textbf{<ff>}\ ] \mid \textbf{\textbackslash"} \mid \textbf{\textbackslash\textbackslash} \mid \textbf{\textbackslash n} \mid \textbf{\textbackslash r} \mid \textbf{\textbackslash f} \mid \textbf{\textbackslash t}\ ]\ \textbf{"} \tag{A.61}$$

$$\mathit{XPath} = \textbf{"}\ \mathit{Selector}\ \textbf{"} \tag{A.62}$$

$$\mathit{PosInt} = [\ \mathbf{0} - \mathbf{9}\ ]+ \tag{A.63}$$

$$\begin{aligned}\mathit{Number} = {} & \mathit{NumberStart}\ [\ \mathit{NumberChar}\ ]* \\ & \mid \textbf{INF} \mid \textbf{-INF} \mid \textbf{NaN}\end{aligned} \tag{A.64}$$

$$\mathit{NumberStart} = \mathbf{0} - \mathbf{9} \mid \mathbf{+} \mid \textbf{-} \mid \textbf{.} \mid \textbf{P} \tag{A.65}$$

$$\mathit{NumberChar} = \mathbf{0} - \mathbf{9} \mid \mathbf{+} \mid \textbf{-} \mid \textbf{.} \mid \textbf{e} \mid \textbf{E} \mid \textbf{T} \mid \textbf{Z} \mid \textbf{Y} \mid \textbf{M} \mid \textbf{D} \mid \textbf{H} \mid \textbf{S} \tag{A.66}$$

$$\mathit{URI} = \textbf{"}\ \mathit{anyURI}\ \textbf{"} \tag{A.67}$$

$$\mathit{Pattern} = \textbf{/}\ \mathit{regExp}\ \textbf{/} \tag{A.68}$$

Appendix B

User Manual

B.1 Software Installation

Copy the following files from directory `impl` to a local directory:

- xsc.jar
- xerces.jar
- xercesImpl.jar
- xsc2xsd.bat
- xsd2xsc.bat

The software needs Java 1.4 or later to run. The batch files for MS Windows are provided for user convenience, on other platforms the Java Runtime has to be started directly. Testing has been done using Apache Xerces version 2.2.1 on Windows XP.

B.2 Conversion

To run the conversion programs, use the following commands. The examples shown convert from compact syntax (xsc) to XML syntax (xsd), just rotate these shortcuts for the other direction.

`xsc2xsd file.xsc`
converts *file.xsc* to XML syntax, creates file *file.xsd*.
`xsc2xsd file1 file2`
converts *file1* to XML syntax and creates file *file2*.

```
java -classpath "xsc.jar;xerces.jar;xercesImpl.jar"
noown.domain.xsc.util.XSCtoXSD file.xsc
```
the same for non-Windows platforms

Options for `xsc2xsd`:
`-d` output debug information
`-p` no indentation for the XML output file
`-e encoding` specify encoding of input and output file
`-a` skip annotations
`-v` validate output file with Xerces

Options for `xsd2xsc`:
`-a` skip annotations
`-v` validate input file (depends on parser)
`-i indent` number of blanks or `t` to be used for indentation
`-n nl` specify line separator (`CR`, `LF` or `CR-LF`)
`-e encoding` specify encoding for input file
`-w n` wrap lines after n characters

Note that the encoding names that can be specified are simply passed to the stream reader and writer classes. Please refer to your Java documentation for the names of the available encodings.

Appendix C

Schema Components

The following tables show the various properties of the Schema Components. All *named* components have additional *name* and *target namespace* properties and all *annotated* components have an additional *annotation* property as shown in table C.1.

Property	**Value**
name	NCName
target namespace	URI
annotation	**Annotation**

Table C.1: General Schema Component Properties

Simple Type Definition *named, annotated*	
Property	**Value**
base type definition	**Simple Type Definition**
facets	list of **Facet**
fundamental facets	list of **Facet**
final	subset of (*extension*, *restriction*, *list*, *union*)
variety	one of (*atomic*, *list*, *union*)
primitive type definition	**Simple Type Definition**
item type definition	**Simple Type Definition**
member type definitions	list of **Simple Type Definition**

Table C.2: Simple Type Definition Schema Component

Complex Type Definition *named, annotated*	
Property	**Value**
base type definition	one of (**Simple Type Definition**, **Complex Type Definition**)
derivation method	one of (*extension*, *restriction*)
final	subset of (*extension*, *restriction*)
abstract	boolean
attribute uses	list of **Attribute Use**
attribute wildcard	**Wildcard**
content type	*empty* or **Simple Type Definition** or pair of (one of (*mixed*, *element-only*) , **Particle**)
prohibited substitutions	subset of (*extension*, *restriction*)

Table C.3: Complex Type Definition Schema Component

Element Declaration *named, annotated*	
Property	**Value**
type definition	one of (**Simple Type Definition**, **Complex Type Definition**)
scope	one of (*global*, **Complex Type Definition**)
value constraint	pair of (one of (*default*, *fixed*) , value)
nillable	boolean
identity-constraint definitions	list of **Identity Constraint Definition**
substitution group affiliations	**Element Declaration**
substitution group exclusions	subset of (*extension*, *restriction*)
disallowed substitutions	subset of (*substitution*, *extension*, *restriction*)
abstract	boolean

Table C.4: Element Declaration Schema Component

Attribute Declaration *named, annotated*	
Property	**Value**
type definition	**Simple Type Definition**
scope	one of (*global*, **Complex Type Definition**)
value constraint	pair of (one of (*default*, *fixed*) , value)

Table C.5: Attribute Declaration Schema Component

Attribute Group Declaration *named, annotated*	
Property	**Value**
attribute uses	list of **Attribute Use**
attribute wildcard	**Wildcard**

Table C.6: Attribute Group Declaration Schema Component

Attribute Use	
Property	**Value**
required	boolean
attribute declaration	**Attribute Declaration**
value constraint	pair of (one of (*default*, *fixed*) , value)

Table C.7: Attribute Use Schema Component

Model Group Definition *named, annotated*	
Property	**Value**
model group	**Model Group**

Table C.8: Model Group Definition Schema Component

Model Group *annotated*	
Property	**Value**
compositor	one of (*all*, *choice*, *sequence*)
particles	list of **Particle**

Table C.9: Model Group Schema Component

Particle	
Property	**Value**
min occurs	non-negative integer
max occurs	one of (*unbounded*, non-negative integer)
term	one of (**Model Group**, **Wildcard**, **Element Declaration**)

Table C.10: Particle Schema Component

Wildcard *annotated*	
Property	**Value**
namespace constraints	*any* or pair of (*not* , NamespaceURI) or list of NamespaceURI
process contents	one of (*skip*, *lax*, *strict*)

Table C.11: Wildcard Schema Component

Identity Constraint Definition *named, annotated*	
Property	**Value**
identity constraint category	one of (*key*, *keyref*, *unique*)
selector	XPath (restricted)
fields	list of XPath (restricted)
referenced key	**Identity Constraint Definition**

Table C.12: Identity Constraint Definition Schema Component

Notation Declaration *named, annotated*	
Property	**Value**
system identifier	URI
public identifier	Public Identifier

Table C.13: Notation Declaration Schema Component

Facet *annotated*	
Property	**Value**
value	depends on facet
fixed	boolean

Table C.14: Facet Schema Component