XML Schema Compact Syntax (XSCS) Version 1.0

Kilian Stillhard and Erik Wilde
Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology, Zürich

TIK Report 166 (March 2003)

Abstract

XML Schema is a schema language for XML, providing advanced features for creating types, deriving types, and a library of built-in simple datatypes. The model behind XML Schema are *XML Schema components*, and XML Schema uses XML syntax for representing XML Schema components. In this report, we present an alternative syntax for XML Schema, which is defined using EBNF productions. Since the new syntax has been designed with the design goals of readability and compactness, it is called *XML Schema Compact Syntax (XSCS)*. XSCS has been created for making XML Schema easier to read and write by humans, while XML Schema's XML syntax is better suited for automated processing of XML Schemas. Consequently, XSCS is not meant as a replacement of the XML syntax, but as a complementary syntax.

Contents

1 Introduction

XML Schema is a schema language for XML, providing advanced features for creating types, deriving types, and a library of built-in simple datatypes. The model behind XML Schema are *XML Schema components*, and XML Schema uses XML syntax for representing XML Schema components. Since XML syntax typically is very verbose and rather hard to read and write for human users, this report defines an alternative syntax for XML Schema, the *XML Schema Compact Syntax (XSCS)*. XSCS fully supports XML Schema Structures [5] and Datatypes [1], with minor exceptions in in the areas of namespace declarations and annotation/comment features.

In many cases, XML Schema is not authored directly but through the use of software, such as graphical schema editors. While these editors often provide good support for writing and presenting schemas, they are often proprietary for a limited number of platforms, often cost considerable amounts of money, and in many cases only provide access to a subset of XML Schema's full functionality, hiding some features or at least making them difficult to access. XSCS can be regarded as an interface also, but a character-based on instead of a graphical interface. This makes it independent from any special software package, since character-based representations can be used on any platform.

This report only contains the syntax definition of XSCS, Section 2 contains an explanation of the syntax in relation to XML Schema's XML syntax, while Section 3 simply is a summary of the syntax definitions. For more information about XSCS, please refer to existing publications about XSCS [6, 7] or to the XSCS Web page at http://dret.net/projects/xscs/. Also available is a description of a software package implementing XSCS interpretation and generation [4].

2 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 who don't know the Schema Components model (which is the vast majority of XML Schema users).

2.1 Design Principles

An XML schema is basically a collection of Schema Components (XML Schema's components are shown in Figure 1[1]). These components can refer to other components and they can contain components themselves. The Schema Components can be divided into several categories.

[1] Taken from [5] (http://www.w3.org/TR/xmlschema-1); Copyright ©2003 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231.

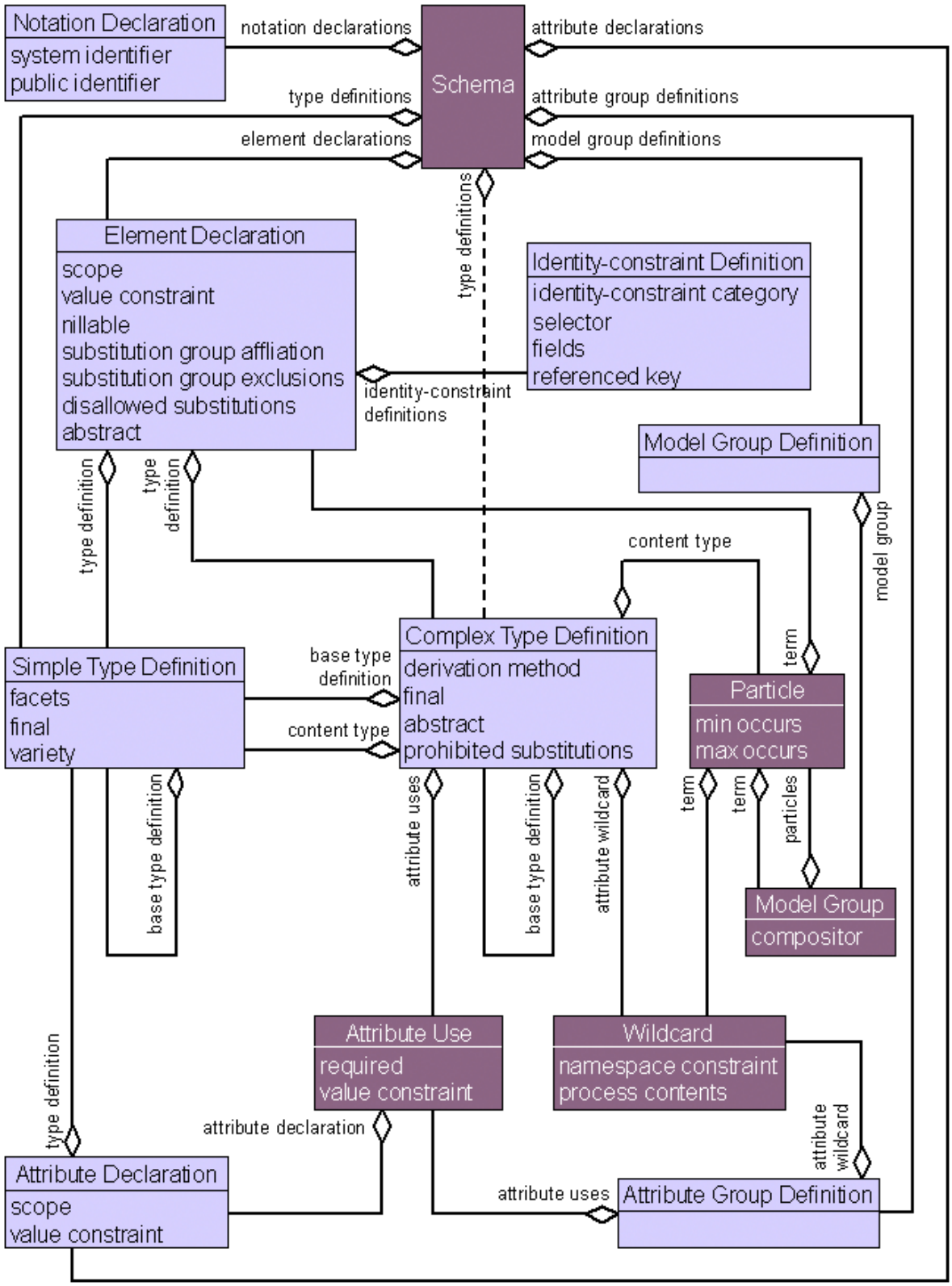

Figure 1: XML Schema Components

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[2], 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.

2.2 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

[2] 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.

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.

2.2.1 Schemas as a whole

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

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

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

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

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.

2.2.2 Schema Options

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

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 1: Namespace options

The *targetNamespace* option (see Table 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 (see Table 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 (see Table 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.

The *elementDefault* and *attributeDefault* options (table 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.

A *version* option (see Table 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.

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 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: Form default settings

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

Table 4: Version specification

2.2.3 Import/Include statements

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

The *import*, *include* and *redefine* statements (table 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.

2.3 Describing Structures

2.3.1 Common Structures

$$
\begin{aligned}
qualifier &= \textbf{final} \mid \textbf{final-restriction} \mid \textbf{final-extension} \mid \textbf{final-list} \\
&\quad \mid \textbf{final-union} \mid \textbf{block} \mid \textbf{block-substitution}
\end{aligned}
$$

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 5: Import, Include and Redefine

$$\begin{aligned} &| \textbf{block-extension} \mid \textbf{block-restriction} \\ &| \textbf{qualified} \mid \textbf{unqualified} \\ &| \textbf{abstract} \mid \textbf{nillable} \\ &| \textbf{required} \mid \textbf{optional} \mid \textbf{prohibited} \end{aligned} \tag{14}$$

$$\textit{derivation} = \textbf{extends}\ Name \mid \textbf{restricts}\ Name \tag{15}$$

$$\textit{substitution} = \textbf{substitutes}\ Name \tag{16}$$

$$fixedDefault = = String \mid <= String \tag{17}$$

Qualifiers (see Table 6) set the values of attributes that are common to some schema components. Multiple *final* and *block* qualifiers can be specified with one component, but *qualified* and *unqualified* as well as *required*, *optional* and *prohibited* exclude each other.

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 6: Qualifiers

The *derivation*, *substitution* and *fixedDefault* extensions (see Table 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 7: Extensions

2.3.2 Elements

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

$$elementShort = Name\ [\ \textbf{\{}\ Name\ \textbf{\}}\ ] \tag{19}$$

$$\begin{aligned} elementContent &= \textbf{\{}\ [\ anonSimpleType \mid anonComplexType \\ &\quad \mid key \mid keyref \mid unique\ ]*\ \textbf{\}} \end{aligned} \tag{20}$$

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 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 2.3.4.

- 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 Section 2.3.4). 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 9: Examples for *element*

2.3.3 Attributes

$$attribute \quad = \quad [\ qualifier\ ] * \ \textbf{attribute}\ Name$$

$$[\ attributeContent\ ]?\ [\ fixedDefault\ ]\ [\ \mathbf{;}\ ] \tag{21}$$

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

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

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

Table 10: Allowed qualifiers and extensions for *attribute*

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 11: Examples for *attribute*

2.3.4 Complex Types

$$\begin{aligned} complexType \quad &= \quad [\ qualifier\ ]* \ \mathbf{complexType}\ Name \\ &\quad\ \ [\ derivation\ ]\ [\ complexTypeContent\ ]\ [\ \mathbf{;}\ ] \end{aligned} \tag{23}$$

$$complexTypeContent \quad = \quad \mathbf{\{}\ [\ anonComplexType \mid anonSimpleType\ ]*\ \mathbf{\}}$$

(24)

$$anonComplexType = contentModel \mid element \mid attribute \mid attributeWC \mid attributeGroup \quad (25)$$

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 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 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.

$$\begin{aligned} \textit{contentModel} &= (\ \textbf{empty} \\ &\quad |\ [\ \textbf{mixed}\ ]\ (\ \textit{modelGroup} \mid \textit{groupRef}\ ) \\ &\quad [\ \textit{occurrenceSpec}\ ]\ )\ [\ \textbf{;}\ ] \end{aligned} \tag{26}$$

$$\textit{occurrenceSpec} = \textbf{?} \mid \textbf{*} \mid + \mid \textit{posIntRange} \tag{27}$$

$$\textit{modelGroup} = (\ [\ \textit{particle}\ [\ \textit{compositor particle}\ ]*\ ]\ [\ \textit{compositor}\ ]) \tag{28}$$

$$\textit{compositor} = \textbf{,} \mid — \mid \textbf{\&} \tag{29}$$

$$\begin{aligned} \textit{particle} &= (\ \textit{modelGroup} \mid \textit{elementShort} \mid \textit{groupRef} \mid \textbf{\{}\ \textit{element}\ \textbf{\}} \\ &\quad |\ \textbf{\{}\ \textit{elementWC}\ \textbf{\}}\ )\ [\ \textit{occurrenceSpec}\ ] \end{aligned} \tag{30}$$

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* (see Table 13) can follow to set the number of allowed repetitions of the particle. It defaults to one and exactly one repetition.

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 13: Definition of the occurrence specifiers

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 Section 2.5.3) are defined similar to inline *elements* using curly brackets.

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 14: Complex type examples

2.4 Describing Datatypes

2.4.1 Simple Types

$$\begin{aligned} \mathit{simpleType} &= [\ \mathit{qualifier}\ ]* \ \textbf{simpleType}\ \mathit{Name} \\ &\quad [\ \mathit{simpleTypeContent}\ ]\ [\ \textbf{;}\ ] \end{aligned} \tag{31}$$

$$\mathit{simpleTypeContent} = \textbf{\{}\ [\ \mathit{anonSimpleType}\ ]\ \textbf{\}} \tag{32}$$

$$\mathit{anonSimpleType} = \mathit{restriction} \mid \mathit{union} \mid \mathit{list} \tag{33}$$

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

$$\begin{aligned} \mathit{restriction} &= (\ \mathit{Name}\ [\ \textbf{\{}\ [\ \mathit{facet}\ ]*\ \textbf{\}}\ ] \\ &\quad \mid \textbf{simpleType}\ \textbf{\{}\ \mathit{anonSimpleType}\ \textbf{\}}\ \textbf{\{}\ [\ \mathit{facet}\ ]*\ \textbf{\}}\ )\ [\ \textbf{;}\ ] \end{aligned} \tag{34}$$

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

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

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.

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 15: Simple Type examples

2.4.2 Facets

$$fixed \;=\; \textbf{fixed} \mid \textbf{fixed-minimum} \mid \textbf{fixed-maximum} \tag{37}$$

$$\begin{aligned} facet \;=\; & [\; fixed \;]* \quad (\; lengthFacet \mid rangeFacet \\ & \mid patternFacet \mid enumFacet \mid whiteSpaceFacet \\ & \mid totalDigitsFacet \mid fractionDigitsFacet \;)\; [\; \textbf{;} \;] \end{aligned} \tag{38}$$

$$lengthFacet = \textbf{length} = (PosInt \mid posIntRange) \quad (39)$$
$$rangeFacet = numRange \quad (40)$$
$$patternFacet = / Pattern / \quad (41)$$
$$enumFacet = String [\textbf{,} String] * \quad (42)$$
$$whiteSpaceFacet = \textbf{whiteSpace} = (\textbf{preserve} \mid \textbf{collapse} \mid \textbf{replace}) \quad (43)$$
$$totalDigitsFacet = \textbf{totalDigits} = PosInt \quad (44)$$
$$fractionDigitsFacet = \textbf{fractionDigits} = PosInt \quad (45)$$

$$posIntRange = [(PosInt [\textbf{,} PosInt] \mid \textbf{,} PosInt)] \quad (46)$$
$$numRange = ([\mid \textbf{(}) (Number [\textbf{,} Number] \mid \textbf{,} Number) ( ] \mid \textbf{)}) \quad (47)$$

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 Section 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.

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 16: Facet examples

2.5 Other Features

2.5.1 Model Groups

$$group \quad = \quad \textbf{group}\ Name\ [\ \textbf{\{}\ [\ contentModel \mid element\ ]*\ \textbf{\}}\ ]\ [\ \textbf{;}\ ] \tag{48}$$

$$groupRef \quad = \quad @\ Name \tag{49}$$

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 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 17: Group examples

2.5.2 Attribute Groups

$$\begin{aligned} \mathit{attributeGroup} \;=\; & \textbf{attributeGroup}\ \mathit{Name}\ [\ \textbf{\{}\ [\ \mathit{attribute} \mid \mathit{attributeWC} \\ & \mid \mathit{attributeGroup}\ ]+\ \textbf{\}}\ ][\ \textbf{;}\ ] \end{aligned} \tag{50}$$

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 18: Attribute group examples

2.5.3 Wildcards

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

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

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

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

Wildcards (see Table 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 19: Wildcard examples

2.5.4 Identity Constraints

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

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

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 20: Wildcard options

$$\begin{aligned} \mathit{keyref} &= \textbf{keyref}\ \mathit{Name} \\ &\quad\ \textbf{refers}\ \mathit{Name}\ \mathit{idConstrField}\ [\ \textbf{;}\ ] \qquad (57) \\ \mathit{unique} &= \textbf{unique}\ \mathit{Name}\ \mathit{idConstrField}\ [\ \textbf{;}\ ] \qquad (58) \end{aligned}$$

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 21: Identity constraint examples

2.5.5 Notations

$$\mathit{notation} = \textbf{notation}\ \mathit{Name}\ \textbf{public}\ \mathit{String}\ \textbf{system}\ \mathit{URI}\ [\ \textbf{;}\ ] \qquad (59)$$

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 22: Notation example

2.5.6 Literals

$$Name \quad = \quad NCName \mid QName \mid \backslash\ NCName \tag{60}$$

A *Name* is either a QName or NCName as defined in the XML Namespace Standard [2]. For names that are equal to any of the keywords (see Table 23), a preceding backslash has to be added.

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 23: Reserved keywords

$$String \quad = \quad \text{"} \ [\ [\ \hat{}\ \text{"}\ \backslash\ \langle\mathbf{nl}\rangle\ \langle\mathbf{cr}\rangle\ \langle\mathbf{ff}\rangle\ ]\ \mid\ \backslash\text{"}\ \mid\ \backslash\backslash\ \mid\ \backslash\mathbf{n}\ \mid\ \backslash\mathbf{r}\ \mid\ \backslash\mathbf{f}\ \mid\ \backslash\mathbf{t}\ ]\ \text{"} \tag{61}$$

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.

$$XPath \quad = \quad \text{"} \; Selector \; \text{"} \tag{62}$$

The *XPaths* used in XML Schema are a subset of the XPath specification [3] defined in the XML Schema standard as the Selector production. XPaths must be enclosed in double quotes.

$$PosInt \quad = \quad [\ \mathbf{0-9}\ ]+ \tag{63}$$

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

$$\begin{aligned} Number \quad &= \quad NumberStart\ [\ NumberChar\ ]* \\ &\quad\ \ | \ \mathbf{INF} \ | \ \mathbf{\text{-}INF} \ | \ \mathbf{NaN} \end{aligned} \tag{64}$$

$$NumberStart \quad = \quad \mathbf{0-9} \ | \ + \ | \ \mathbf{\text{-}} \ | \ \mathbf{.} \ | \ \mathbf{P} \tag{65}$$

$$NumberChar \quad = \quad \mathbf{0-9} \ | \ + \ | \ \mathbf{\text{-}} \ | \ \mathbf{.} \ | \ \mathbf{e} \ | \ \mathbf{E} \ | \ \mathbf{T} \ | \ \mathbf{Z} \ | \ \mathbf{Y} \ | \ \mathbf{M} \ | \ \mathbf{D} \ | \ \mathbf{H} \ | \ \mathbf{S} \tag{66}$$

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[3] types, the decimal type, and the double and float types.

$$URI \quad = \quad \text{"} \; anyURI \; \text{"} \tag{67}$$

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

$$Pattern \quad = \quad / \; regExp \; / \tag{68}$$

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.

3 Syntax Summary

The following syntax summary uses the same numbering as the syntax description in the preceding section. The summary is separated into syntax descriptions containing further structural elements (Section 3.1), and literals (Section 3.2).

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

3.1 Structure

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

$$
\begin{aligned}
\mathit{schemaOption} &= \mathit{targetNamespace} \\
&\quad | \ \mathit{namespace} \\
&\quad | \ \mathit{blockFinalDefault} \\
&\quad | \ \mathit{elementDefault} \\
&\quad | \ \mathit{attributeDefault} \\
&\quad | \ \mathit{version}
\end{aligned} \tag{2}
$$

$$
\begin{aligned}
\mathit{schemaInclude} &= \mathit{include} \\
&\quad | \ \mathit{import} \\
&\quad | \ \mathit{redefine}
\end{aligned} \tag{3}
$$

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

$$\mathit{targetNamespace} = \textbf{targetNamespace}\ \mathit{URI}\ [\ \textbf{;}\ ] \tag{5}$$

$$\mathit{namespace} = \textbf{namespace}\ [\ \mathit{Name}\ ]\ \mathit{URI}\ [\ \textbf{;}\ ] \tag{6}$$

$$\mathit{blockFinalDefault} = \textbf{default}\ \mathit{qualifier}\ [\ \textbf{,}\ \mathit{qualifier}\ ]*\ [\ \textbf{;}\ ] \tag{7}$$

$$\mathit{elementDefault} = \textbf{elementDefault}\ \mathit{qualifier}\ [\ \textbf{;}\ ] \tag{8}$$

$$\mathit{attributeDefault} = \textbf{attributeDefault}\ \mathit{qualifier}\ [\ \textbf{;}\ ] \tag{9}$$

$$\mathit{version} = \textbf{version}\ \mathit{String}\ [\ \textbf{;}\ ] \tag{10}$$

$$\mathit{include} = \textbf{include}\ \mathit{URI}\ [\ \textbf{;}\ ] \tag{11}$$

$$\mathit{import} = \textbf{import}\ \mathit{URI}\ \textbf{namespace}\ \mathit{URI}\ [\ \textbf{;}\ ] \tag{12}$$

$$
\begin{aligned}
\mathit{redefine} &= \textbf{redefine}\ \mathit{URI}\ [\ \textbf{\{}\ [\ \mathit{simpleType}\ |\ \mathit{complexType} \\
&\quad |\ \mathit{group}\ |\ \mathit{attributeGroup}\ ]*\ \textbf{\}}\ ]\ [\ \textbf{;}\ ]
\end{aligned} \tag{13}
$$

$$
\begin{aligned}
\mathit{qualifier} &= \textbf{final}\ |\ \textbf{final-restriction}\ |\ \textbf{final-extension}\ |\ \textbf{final-list} \\
&\quad |\ \textbf{final-union}\ |\ \textbf{block}\ |\ \textbf{block-substitution}
\end{aligned}
$$

| **block-extension** | **block-restriction**
| **qualified** | **unqualified**
| **abstract** | **nillable**
| **required** | **optional** | **prohibited** (14)

derivation = **extends** *Name* | **restricts** *Name* (15)

substitution = **substitutes** *Name* (16)

fixedDefault = **=** *String* | **<=** *String* (17)

element = [*qualifier*] ∗ **element** *Name*
[*substitution* | *derivation*] ∗ [*elementContent*]
[*fixedDefault*] [**;**] (18)

elementShort = *Name* [**{** *Name* **}**] (19)

elementContent = **{** [*anonSimpleType* | *anonComplexType*
| *key* | *keyref* | *unique*] ∗ **}** (20)

attribute = [*qualifier*] ∗ **attribute** *Name*
[*attributeContent*]? [*fixedDefault*] [**;**] (21)

attributeContent = **{** [*anonSimpleType*] **}** (22)

complexType = [*qualifier*] ∗ **complexType** *Name*
[*derivation*] [*complexTypeContent*] [**;**] (23)

complexTypeContent = **{** [*anonComplexType* | *anonSimpleType*] ∗ **}** (24)

anonComplexType = *contentModel* | *element* | *attribute*
| *attributeWC* | *attributeGroup* (25)

contentModel = (**empty**
| [**mixed**] (*modelGroup* | *groupRef*)
[*occurrenceSpec*]) [**;**] (26)

occurrenceSpec = **?** | ***** | + | *posIntRange* (27)

$$modelGroup = \textbf{(}\ [\ particle\ [\ compositor\ particle\ ]*\ ]\ [\ compositor\ ]\textbf{)} \quad (28)$$

$$compositor = \textbf{,} \mid \textbf{—} \mid \textbf{\&} \quad (29)$$

$$particle = (\ modelGroup \mid elementShort \mid groupRef \mid \textbf{\{}\ element\ \textbf{\}} \mid \textbf{\{}\ elementWC\ \textbf{\}}\ )\ [\ occurrenceSpec\ ] \quad (30)$$

$$simpleType = [\ qualifier\ ]*\ \textbf{simpleType}\ Name\ [\ simpleTypeContent\ ]\ [\ \textbf{;}\ ] \quad (31)$$

$$simpleTypeContent = \textbf{\{}\ [\ anonSimpleType\ ]\ \textbf{\}} \quad (32)$$

$$anonSimpleType = restriction \mid union \mid list \quad (33)$$

$$restriction = (\ Name\ [\ \textbf{\{}\ [\ facet\ ]*\ \textbf{\}}\ ] \mid \textbf{simpleType \{}\ anonSimpleType\ \textbf{\} \{}\ [\ facet\ ]*\ \textbf{\}}\ )\ [\ \textbf{;}\ ] \quad (34)$$

$$union = \textbf{union \{}\ [\ anonSimpleType\ ]+\ \textbf{\}}\ [\ \textbf{;}\ ] \quad (35)$$

$$list = \textbf{list \{}\ anonSimpleType\ \textbf{\}}\ [\ \textbf{;}\ ] \quad (36)$$

$$fixed = \textbf{fixed} \mid \textbf{fixed-minimum} \mid \textbf{fixed-maximum} \quad (37)$$

$$facet = [\ fixed\ ]*\quad (\ lengthFacet \mid rangeFacet \mid patternFacet \mid enumFacet \mid whiteSpaceFacet \mid totalDigitsFacet \mid fractionDigitsFacet\ )\ [\ \textbf{;}\ ] \quad (38)$$

$$lengthFacet = \textbf{length} = (\ PosInt \mid posIntRange\ ) \quad (39)$$

$$rangeFacet = numRange \quad (40)$$

$$patternFacet = /\ Pattern\ / \quad (41)$$

$$enumFacet = String\ [\ \textbf{,}\ String\ ]* \quad (42)$$

$$whiteSpaceFacet = \textbf{whiteSpace} = (\ \textbf{preserve} \mid \textbf{collapse} \mid \textbf{replace}\ ) \quad (43)$$

$$totalDigitsFacet = \textbf{totalDigits} = PosInt \quad (44)$$

$$fractionDigitsFacet = \textbf{fractionDigits} = PosInt \quad (45)$$

$$posIntRange = [\ (\ PosInt\ [\ \textbf{,}\ PosInt\ ] \mid \textbf{,}\ PosInt\ )\ ] \quad (46)$$

$$numRange = ([| ( ) (Number [, Number] | , Number) ( ] |)) \quad (47)$$

$$group = \textbf{group}\ Name\ [\ \textbf{\{}\ [\ contentModel \mid element\ ]*\ \textbf{\}}\ ]\ [\ \textbf{;}\ ] \quad (48)$$

$$groupRef = @\ Name \quad (49)$$

$$attributeGroup = \textbf{attributeGroup}\ Name\ [\ \textbf{\{}\ [\ attribute \mid attributeWC \mid attributeGroup\ ]+\ \textbf{\}}\ ][\ \textbf{;}\ ] \quad (50)$$

$$process = \textbf{lax} \mid \textbf{strict} \mid \textbf{skip} \quad (51)$$

$$wildcardNSDecl = \textbf{\#\#targetNS} \mid \textbf{\#\#other} \mid \textbf{\#\#local} \mid URI \quad (52)$$

$$elementWC = [\ process\ ]\ \textbf{any}\ [\ \textbf{namespace}\ wildcardNSDecl\ [\ \textbf{,}\ wildcardNSDecl\ ]*\ ]\ [\ \textbf{;}\ ] \quad (53)$$

$$attributeWC = [\ process\ ]\ \textbf{anyAttribute}\ [\ \textbf{namespace}\ wildcardNSDecl\ [\ \textbf{,}\ wildcardNSDecl\ ]*\ ]\ [\ \textbf{;}\ ] \quad (54)$$

$$idConstrField = \textbf{field}\ XPath\ [\ \textbf{,}\ XPath\ ]*\ \textbf{in}\ XPath \quad (55)$$

$$key = \textbf{key}\ Name\ idConstrField\ [\ \textbf{;}\ ] \quad (56)$$

$$keyref = \textbf{keyref}\ Name\ \textbf{refers}\ Name\ idConstrField\ [\ \textbf{;}\ ] \quad (57)$$

$$unique = \textbf{unique}\ Name\ idConstrField\ [\ \textbf{;}\ ] \quad (58)$$

$$notation = \textbf{notation}\ Name\ \textbf{public}\ String\ \textbf{system}\ URI\ [\ \textbf{;}\ ] \quad (59)$$

3.2 Literals

$$Name = NCName \mid QName \mid \backslash\ NCName \quad (60)$$

$$String = \text{”}\ [\ [\ \hat{}\ \text{”}\ \backslash\ \textbf{<nl>}\ \textbf{<cr>}\ \textbf{<ff>}\ ]\ |\ \backslash\text{”}\ |\ \backslash\backslash\ |\ \backslash\textbf{n}\ |\ \backslash\textbf{r}\ |\ \backslash\textbf{f}\ |\ \backslash\textbf{t}\ ]\ \text{”} \quad (61)$$

$$XPath = \text{”}\ Selector\ \text{”} \quad (62)$$

$$PosInt = [\ \textbf{0} - \textbf{9}\ ]+ \quad (63)$$

$$Number = NumberStart\ [\ NumberChar\ ]* \mid \textbf{INF} \mid \textbf{-INF} \mid \textbf{NaN} \quad (64)$$

$$NumberStart = \mathbf{0-9} \mid \mathbf{+} \mid \mathbf{-} \mid \mathbf{.} \mid \mathbf{P} \quad (65)$$

$$NumberChar = \mathbf{0-9} \mid \mathbf{+} \mid \mathbf{-} \mid \mathbf{.} \mid \mathbf{e} \mid \mathbf{E} \mid \mathbf{T} \mid \mathbf{Z} \mid \mathbf{Y} \mid \mathbf{M} \mid \mathbf{D} \mid \mathbf{H} \mid \mathbf{S} \quad (66)$$

$$URI = \text{" } anyURI \text{ "} \quad (67)$$

$$Pattern = \text{/ } regExp \text{ /} \quad (68)$$

References

[1] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes. World Wide Web Consortium, Recommendation REC-xmlschema-2-20010502, May 2001.

[2] Tim Bray, Dave Hollander, and Andrew Layman. Namespaces in XML. World Wide Web Consortium, Recommendation REC-xml-names-19990114, January 1999.

[3] James Clark and Steven J. DeRose. XML Path Language (XPath) Version 1.0. World Wide Web Consortium, Recommendation REC-xpath-19991116, November 1999.

[4] Kilian Stillhard. A Compact Syntax for XML Schema. Master's thesis, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology, Zürich, Switzerland, March 2003.

[5] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures. World Wide Web Consortium, Recommendation REC-xmlschema-1-20010502, May 2001.

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

[7] Erik Wilde and Kilian Stillhard. Making XML Schema Easier to Read and Write. In *Poster Proceedings of the Twelfth International World Wide Web Conference*, Budapest, Hungary, May 2003.