Design of the BonXai Schema Language

Wim Martens Volker Mattick Matthias Niewerth

Shivam Agarwal Najoua Douib Oliver Garbe Daniel Günther
Denis Oliana Jens Kroniger Fabian Lücke Taner Melikoglu
Kore Nordmann Gökhan Oezen Tobias Schlitt Lars Schmidt
Jakob Westhoff Dominik Wolff

December 11, 2015

Contents

1 Introduction

This document describes the syntax and semantics of the language "BonXai", a pattern based XML Schema language. This language has been developed with two main goals in mind:

- Compatibility to XML Schema in terms of expressiveness
- Easier to write and understand than XML Schema

Essentially, BonXai is aimed to be a language with the expressiveness of XML Schema but with the ease-of-use as DTD. We aim at achieving these two goals by building the design of BonXai on *pattern-based schema languages*. A more detailed description of the goals and underlying principles of BonXai can be found in [3].

In the following sections, we provide syntax examples for the whole range of features in BonXai. Sections 3–5 explain the structural core of BonXai, which is the interplay between ancestor- and child patterns. This is where BonXai is quite different from languages such as XML Schema. Sections 6–13 explain features that are present in XML Schema; for which BonXai largely inherits the semantics from XML Schema; and for which the purpose of this document is mostly to provide syntax examples in BonXai as well in XML Schema.

Throughout this document we will always give EBNF rules for the currently relevant part of the syntax. In some cases we first give a simplified syntax to explain the overall idea and define the exact syntax afterwards.

Finally, the complete EBNF syntax of BonXai is given in Section 14.

Those who are interested in the foundations of pattern-based schema languages that lie at the root of BonXai, can find more information in [1, 2, 4, 5].

2 Basic Structure and Philosophy

Here, we describe the basic structure of a BonXai schema by a commented example. Comments in BonXai schemas can be specified using the # character. Comments extend to the end of the line.

Code-example 2.1: Example BonXai Schema

```
# I'm declaring my target namespace here
target namespace http://example.org/ns

# I'd like to use some stuff from the XML Schema namespace
namespace xs = http://www.w3.org/2001/XMLSchema

# At the root of XML trees, I'll allow only one element name, called myRootElement
global { myRootElement }

# Groups are nice to make the schema succinct
groups {
  group abc = { element a, element b, element c }
}

# Now I'm ready to define the heart of the schema: the grammar
# The grammar is simply a list of rules of the form:
# ancestorPattern = child pattern
# or
# ancestorPattern = type reference
grammar {
  # The content of myRootElement is a sequence of three elements
  /myRootElement = {
    element firstChild,
    element secondChild,
    element thirdChild
  }

  # Let's define what the children of my root element should look like
  //firstChild = {xs:string}
  //secondChild = {group abc, element d}
  //thirdChild = {attribute myAtt {xsd:string}, element a {xsd:string}}
}

# Finally, I'd like to add some integrity constraints
constraints {
  unique /myRootElement {
    ./thirdChild {
      ./@myAtt, ./a
    }
}
```

Each BonXai schema has up to five blocks, as defined by the following EBNF rule:

```
1: Bonxai   ::= Namespaces Global Groups? Grammar Constraints?
```

The mandantory namespace block specifies the target namespace of the schema and declares any namespace prefixes used in the schema. It is described in Section 8.
The namespace block is followed by the mandantory global block contains the list of global elements and attributes defined in the schema.

```
3: Global   ::= "global" "{" ( NCName | "@"NCName )+ "}"
```

The optional groups block may contain group definitions. Group definitions are covered in Section 7.
The mandantory grammar block is the core of a BonXai schema. It contains a list of rules.

```
6: Grammar  ::= "grammar" "{" Rule* "}"
```

Each rule consists of an ancestor pattern and a child pattern. The ancestor pattern is a regular expression which matches the elements in the XML document to which the child pattern is applicable. The child pattern describes the content model of these elements. For example, the rule

```
/myRootElement = {
  element firstChild,
  element secondChild,
  element thirdChild
}
```

specifies that the children of `myRootElement` should be named `firstChild`, `secondChild` and `thirdChild`. Ancestor patterns are described in Section 3, while child patterns are described in Sections 4 and 5.
A rule in BonXai can also have a type reference instead of a child pattern. This is especially useful for simple types. For example, the rule

```
//firstChild = {xs:string}
```

specifies that `firstChild` should contain a string. Type references are explained in Section 10.
A rule may be prefixed with annotations (explained in Section 13).

```
7: Rule     ::= Annotation* AncestorPattern "=" (ChildPattern | TypeRef)
```

3 Ancestor Patterns

As an XML Schema language, BonXai needs a way to address a group of elements for which the child pattern, also known as the content of each element, is described. BonXai makes use of pattern based languages to match the ancestor path of an element in order to address it. Various approaches exist to formulate such patterns, for example regular expressions or XPath[1].
Two criteria were defined, to be fulfilled by the BonXai language, which affect the ancestor pattern language:

1. BonXai may not exceed the descriptive power of XML Schema
2. The used pattern language should be intuitive to XML users

Another important aspect during the development of the ancestor pattern language was, to have a very simple syntactical subset that allows users to describe about 90% of the XML Schemas in practice (see [5]). All additional syntax constructs to fully support the descriptive power of XML Schema were added on top of this basis. The latter additions do not necessary fulfill the criteria of simplicity, but are still intuitive to users who are familiar with regular expressions.
A rule in a BonXai schema that defines the children of a certain element typically has the form

```
<ancestor pattern> = <child pattern>
```

The ancestor pattern (left of the equality sign) describes the context of the rule and should be matched against paths in the tree that start from the root.
Here, we describe the syntax and semantics of ancestor patterns.

3.1 Simple Ancestor Patterns: Linear XPath

The base of the BonXai ancestor pattern language is built on linear XPath ([1]). Linear XPath is a subset of the XPath language, which is restricted to the child- and descendant-axis and only allows to use XML element node names for matching. The pattern expressions only consist of element names (including namespaces whenever relevant), the special element name `*` to indicate any element name and the two XPath operators `/` and `//`. Simple Linear XPath expressions without predicates are defined by the following grammar:

```
LinearXPath ::= LPath | "/" LPath | "//" LPath
LPath       ::= Step | Step "/" LPath | Step "//" LPath
Step        ::= NCName | "*"
```

where NCName stands for "non colon name" as it is defined in XML Schema ([], Section XXX). An NCName is an element name without a namespace specifier. Examples for simple linear XPath expressions are:
`/root/childelement`

[1] `http://www.w3.org/TR/xpath`

```
//descendantelement
descendantelement
/root//descendantelement
/root/*/grandchildelement
//a/b//c/*/d
a/b//c/*/d
```

3.2 Full Ancestor Patterns: Regular Expression Dialect

Ancestor patterns extend the expressiveness of linear XPath expressions to regular languages over strings. The following EBNF is almost the final version.

```
 8: AncestorPattern ::= FAPattern | APattern
 9: APattern        ::= ("@"? (NCName | "*")) FAPattern?
                      | "(" APattern ("|" APattern)* ")"
10: FAPattern       ::= ( "/" | "//" ) APattern
                      | "(" FAPattern ("|" FAPattern)* ")" RepetitionOp? FAPattern?
11: RepetitionOp    ::= "*" | "+" | "?"
```

Intuitively, ancestor patterns (`FullAPattern`) are similar to regular expressions in DTD syntax, but they inherit the two forms of concatenation from XPath:

- `/` for the "child"-relation
- `//` for the "descendant"-relation

When viewing a path in a tree as a word, the semantics of `/` in APatterns is therefore the same as a standard concatenation in regular expressions.
Note that we require all disjunctions to be surrounded with brackets and there are two types of disjunctions, one where the concatenation symbols are outside of the brackets and one where they are inside.
Repetition operators (`*`, `+`, `?`) are also only allowed for bracketed subexpressions. Whether `*` is used as wildcard or for specifying a repetition is always clear from context.
For convenience reasons, it is possible to use ancestor patterns without any `/` ot `//` prefix. These patterns are implicitly assumed to use a `//` prefix. This allows to write `elementname` to match all elements with name `elementname` in a document.
Every linear XPath expression is also an ancestor pattern. Beyond linear XPath, the following expressions are valid ancestor patterns:

- `(/c(/b)+)`: This ancestor pattern matches paths that have c at the root, followed by one or more b's.
- `/c(/b)?`: This ancestor pattern matches paths that have c at the root, followed by zero or one b.
- `(a/b//c/*/d)`: This pattern is equivalent to `//a/b//c/*/d`.
- `(/a(/b/a)*/c/b | /a(/b/a)*/d/c)`: This ancestor pattern allows occurence of a as a root and zero or more occurences of a pattern consisting of /b/a (descendant b and a as b's descendant) then either /c/b or /d/c (immediately below c and d or d and c). For instance

Operator	Arity	Semantics
*	unary	repetition; zero or more
+	unary	repetition; one or more
?	unary	optionality; zero or one
|	binary	disjunction
/	binary	concatenation (child relation)
//	binary	descendant relation

Table 3.1: Regular expression operators for ancestor patterns in BonXai

string acb and abacb matches the pattern since a occurs as root followed by a zero or more occurence of b followed by a (/b/a) , then the descendant c followed by b(/c/b). Also string ababadc is accepted since a occurs as root followed by a zero or more occurence of b followed by a (/b/a) , then the descendant d followed by c (/d/c).While abcb and aacb are not accepted since every b needs the descendant a and every second occurence of a can only occur with b as its ancestor.

To summarize, the semantics of the various operators in ancestor patterns can be found in Table 3.1.

3.3 Example

Code-example 3.1: Ancestor patterns in BonXai

```
target namespace http://example.org/namespace
namespace xs = http://www.w3.org/2001/XMLSchema

global { root, foo }

grammar {
    /root = {
            element first ,
            element second ,
            element third
    }
    /root(/first|/second|/third) = {
         element alpha  { type xs:string } ,
         element beta   { type xs:string }
    }
    /foo(/foo)* = {
        (element foo)+  |  element bar { type xs:string }
    }
}
```

The ancestor patterns for above bonxai schema are listed below.

- `/root(/first|/second|/third)`: This ancestor pattern matches children named `first`, `second`, or `third` below the root element `root`.
- `/foo(/foo)*`: This ancestor pattern matches all nodes named `foo` for which the path to the root only contains elements with name `foo`.

An XML Schema equivalent to the above BonXai schema can be defined as follows.

Code-example 3.2: XML Schema to be translated to BonXai

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://example.org/namespace"
   xmlns:my="http://example.org/namespace"
   elementFormDefault="qualified">

   <element name="root">
      <complexType>
         <sequence>
            <element name="first"  type="my:sequenceExample" />
            <element name="second" type="my:sequenceExample" />
            <element name="third"  type="my:sequenceExample" />
         </sequence>
      </complexType>
   </element>

   <complexType name="sequenceExample">
      <sequence>
         <element name="alpha" type="string" />
         <element name="beta"  type="string" />
      </sequence>
   </complexType>

   <element name="foo" type="my:fooType" />

   <complexType name="fooType">
      <choice>
         <sequence>
            <element name="foo" type="my:fooType" maxOccurs="unbounded" />
         </sequence>
         <element name="bar" type="string" />
      </choice>
   </complexType>
</schema>
```

3.4 Attributes in Ancestor Patterns

In order to succinctly define the structure of certain attributes, it is also possible to end an ancestor pattern with an attribute name, preceded by `@`. For example, it is possible to include a rule of the form

```
//lotsofattributes//@a = { type xs:string }
```

that says that, all attributes with name `a` that occur in a subtree rooted with an element `lotsofattributes` are of type string.
One could also state that all `a`- or `b`-attributes below `lotsofattributes` are of type string, by writing

```
//lotsofattributes//(@a | @b) = { type xs:string }
```

or, simply that *all* attributes in subtrees rooted by `lotsofattributes` are of type string, by writing

```
//lotsofattributes//@* = { type xs:string }
```

Perhaps it comes in handy to write a single rule

```
//@* = { type xs:string }
```

in a BonXai file, stating that all attributes have type string.

4 Content Models (Child Patterns)

BonXai allows to define content models through *child patterns*. A rule in a BonXai schema that defines the children of a certain element typically has the form

```
<ancestor pattern> = <child pattern>
```

where the child pattern can be found on the right of the equality sign. Here, we describe the syntax and semantics of ancestor patterns.

4.1 Basic Structure of Child Patterns

The basic structure of a child pattern in BonXai is summarized by the following grammar. A child pattern is produced by `ChildPattern`:

```
12: ChildPattern   ::= MixedNillable? "{" AttributePattern? ElementPattern "}"
```

We explain the meaning of `MixedNillable` in Chapter 6. The attribute pattern defines which attributes are allowed for an element and is explained in the next chapter. The core part of a child pattern is the element pattern, which describe which elements can occur as children of some element.
Before giving the full syntax for element patterns, we give a simplified suntax excluding some corner cases. Element patterns are usually produced by `regex`:

```
Regex ::= Regex Counter?
        | Regex "," Regex
        | Regex "|" Regex
        | Regex "&" Regex
        | "(" Regex ")"
        | "element" NCName
```

Element patterns use regular expression operators from DTD such as `,` (concatenation), `|` (disjunction), `*` (repetition; zero or more), `+` (repetition; one or more), and `?` (optionality). We also use the additional operators which XML Schema uses to define content models: `&` for interleaving and the variants of `[x,y]` for bounded iteration. (More precisely, for two natural numbers `x` and `y`, the operator `[x,y]` corresponds to XML Schema's `minOccurs="x" maxOccurs="y"`.) The semantics of the various operators in ancestor patterns is summarized in Table 4.1.

Operator	Arity	Semantics
`*`	unary	repetition; zero or more
`+`	unary	repetition; one or more
`?`	unary	optionality; zero or one
`[n]`	unary	repetition; exactly `n` times
`[n,m]`	unary	repetition; `n` times minimum, `m` times maximum
`[n,*]`	unary	repetition; `n` times minimum
`,`	binary	concatenation
`|`	binary	disjunction
`&`	binary	XML Schema "all" (interleave operator)

Table 4.1: Regular expression operators for child patterns in BonXai.

4.2 Example (Basic Child Patterns)

The following BonXai schema defines documents that have an element `myroot` at their root. The children of the root are elements `a`, `b`, and, optionally, `c`. Furthermore, from left to right, the string of children contains 5 to 10 elements called `a` or `b`, followed by an optional `c`.

Code-example 4.1: Basic Child Patterns in BonXai

```
target namespace http://example.org/namespace
namespace xs = http://www.w3.org/2001/XMLSchema

global { myroot }

grammar {
    /myroot = { (element a, element b)[5,10], element c? }
    /myroot/* = { type xs:string }
}
```

An XML Schema equivalent to the above BonXai schema can be defined as follows.

Code-example 4.2: XML Schema to be translated to BonXai

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://example.org/namespace"
   elementFormDefault="qualified">

   <element name="myroot">
      <complexType>
         <sequence>
           <sequence minOccurs="5" maxOccurs="10">
             <element name="a"  type="string" />
             <element name="b"  type="string" />
           </sequence>
           <element name="c" minOccurs="0" type="string" />
        </sequence>
      </complexType>
   </element>
</schema>
```

4.3 Restrictions on Element Patterns

For compatibility with XML Schema, we impose the same restrictions on element patterns as XML Schema does for its complex types. This means that the regular expressions for element patterns should be *deterministic* in the same way as the XML Schema specification requires of complex types definitions. In the XML Schema specification, these restrictions can be found under the name *Unique Particle Attribution* (see Chapter 3.8.6 of [6]).
Furthermore, we syntactically restrict the use of the interleave operator `&` in the same way as XML Schema does for its `all` operator (see Section 3.8.2 in [6]).

4.4 Full syntax for Element Patterns

We now give the full syntax for element patterns. This syntax restricts the use of the interleave operator (`&` as described above and introduces the special element pattern `empty`, meaning, that there are no child elements allowed.

```
17: ElementPattern ::= "empty" | AllPattern | Regex
18: AllPattern     ::= ElementDecl "?"? ( "&" ElementDecl "?"? )+
19: Regex          ::= Regex Counter?
                     | Regex "," Regex
                     | Regex "|" Regex
                     | "(" Regex ")"
                     | ElementDecl
                     | "any" WildcardDecl?
```

`Counter` is defined as above. We replaced `"element" NCName` with `ElementDecl` as there are different possibilities for declaring elements (see below). The additional possibility of choosing `"any" WildcardDecl?` accounts for the possibility of adding wildcards to the content models as described in Section 9.
Element declarations can be of three different kinds:

```
21: ElementDecl   ::= "element" NCName InlineTypeRef?
                    | "elementref" QName
                    | "group" NCName
```

In its simplest case, an element declaration just specifies an element name. Additionally, a type reference can be specified. The other two kinds for an element declaration are `elementref` which refers to a global element in some namespace[1] and `group` which includes the content model of the referenced group. For type and element references see Section 10 and for groups see Section 7.

[1] This namespace can also be the target namespace of the schema.

5 Attributes (Child Patterns)

The specification of attributes for XML elements also happens in child patterns. More particularly, attributes in BonXai are specified in the beginning of child patterns. Attributes can be declared optional. For example, the rule

```
/myroot = {
  attribute att1 { type xs:integer } ?,
  attribute att2 { type xs:string },
  element a*, element b, element c
}
```

specifies that the root element `myroot` can have two attributes: `att1` (which is optional) and `att2`. Attribute `att1` is an integer and attribute `att2` is a string. We use the question mark `?` to specify that an attribute is optional.

5.1 Example

Code-example 5.1

```
target namespace http://example.org/namespace
namespace xs = http://www.w3.org/2001/XMLSchema

global { address }

grammar {
  /address = {
        attribute id           { type  xs:integer },
        attribute private      { type  xs:boolean }?,
        element   name         { type  xs:string } ,
        element   street       { type  xs:string } ,
        element   city,
        element   zip
  }
  /address/zip = { type xs:integer }
  /address/city = { type xs:string }
}
```

This example shows a simple address element containing the attributes "id" and, optionally, "private". Furthermore address instances have to contain elements name, street, city and zip in this exact order. The simple types of id, private, name and street are specified inside the child-pattern, whereas the simple types of city and zip are specified as seperate BonXai expressions. In comparison to the BonXai example the same address definition in XML Schema looks as follows:

Code-example 5.2

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://example.org/namespace">
   <element name="address">
      <complexType>
         <sequence>
            <element name="name" type="string"/>
            <element name="street" type="string"/>
            <element name="city" type="string"/>
            <element name="zip" type="integer"/>
         </sequence>
         <attribute name="id" type="integer" use="required"/>
         <attribute name="private" type="boolean"/>
      </complexType>
   </element>
</schema>
```

6 Mixed and Nillable Content Models

XML Schema provides the author with an easy way to declare elements which can have mixed content or no content at all.
An essential XML feature is to provide text and element content together inside one element node. This so called *mixed mode* is commonly used in text annotation formats like XHTML and is supported by XML Schema.
XML Schema uses the attribute `mixed` on the `complexType` element to define that the content of an XML element may consist of the elements given in the complex type definition and free form text content in addition.
A different feature are nillable elements. Nillable elements are specified in XML Schema using the attribute nillable set to true in the element definition. Using this technique the instance can provide either an element filled with the defined type or an element which is empty and carries the attribute `nil` from the namespace of the XML Schema instance called `http://www.w3.org/2001/XMLSchema-instance` set to true. This feature is mostly used to express that some values have yet to be inserted into the instance.
Both features can be combined. The BonXai language allows to define the contents of an element to be mixed and/or nillable by adding the keywords `mixed` and/or `nillable` before its describing child pattern as specified in the corresponding EBNF rules:

```
12: ChildPattern  ::= MixedNillable? "{" AttributePattern? ElementPattern "}"
13: MixedNillable ::= "mixed" "nillable"? | "nillable" "mixed"?
```

6.1 Example

The following Example demonstrate the usage of mixed and nillable in XML Schema. The element `someElement` may not be filled in the XML Schema instance if instead the attribute `xsi:nil` is set to `true`. The element `anotherelement` may contain text and

Code-example 6.1: XML Schema using nillable and mixed

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://example.org/namespace">
  <element name="someElement" nillable="true">
    <complexType>
      <sequence>
        <element name="anOtherElement">
          <complexType mixed="true">
            <sequence>
              <element name="first"  type="xs:integer" />
              <element name="second" type="xs:integer" />
            </sequence>
          </complexType>
        </element>
      </sequence>
```

```
    </complexType>
  </element>
</schema>
```

The provided example shows a XML Schema which uses nillable to allow the element `someElement` to be nil in the instance document. If `someElement` is not nil, then it needs to have a child element `anotherElement`, which contains elements `first` and `second` and may contain additional text. The following example shows exactly the same definition made in BonXai.

Code-example 6.2: BonXai using nillable and mixed

```
target namespace http://example.org/namespace
namespace xs = http://www.w3.org/2001/XMLSchema

global { someElement }

grammar {
   /someElement = nillable { element anotherElement }
   //anotherElement = mixed { element first, element second }
   //(first | second) = { type xs:integer }
}
```

Code-example 6.3: An example where someElement is nil

```
<?xml version="1.0"?>
<someElement xmlns="http://example.org/namespace"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:nil="true"/>
```

Code-example 6.4: An example where someElement is not nil

```
<?xml version="1.0"?>
<someElement xmlns="http://example.org/namespace">
  <anotherElement>
    We can <first>23</first> mix text with <second>42</second> child elements.
  </anotherElement>
</someElement>
```

7 Element and Attribute Groups

The group feature allows the user to define a group of elements or attributes and use them in any location of the schema. It makes the schemas better to write and read because instead of using many elements/attributes you can group these elements/attributes into an element/attribute group and use them in any declaration and definition of the schema.

7.1 Element Groups

Element groups in XML Schema are intended to allow the user to specify and a regular expression so that it can be used to define multiple content models in the schema. To this end, one can give the regular expression a name and then refer to it from elsewhere in the schema.
BonXai supports element groups via keyword the `group` and allows all of the features in groups which are possible with XML Schema. Groups are declared in a `groups` block before the `grammar` block.

7.2 Attribute Groups

XML Schema allows to define attribute groups with the same use case like element groups, just for attributes. In XML Schema an attribute group can be included in an attribute group or complex type and an attribute group reference must appear at the end of complex type definitions. BonXai supports the attribute groups and all of their features in XML Schema. In BonXai one can define an attribute group via the keyword `attribute-group`.

7.3 Difference between BonXai and XML Schema

Note that there is one big difference for groups between BonXai and XML Schema. In BonXai an element group is just a regular expression. In XML Schema an element group additionally specifies the types for the elements specified in the regular expression. Therefore it might be necessary to translate one BonXai element group to several XML Schema element groups, as the elements inside the group may have different content models if the group is used in different places of the BonXai schema.

7.4 BonXai Syntax

Groups are dealt with in two different places in BonXai. The first place is the group block, where groups are defined. The syntax is given by the following EBNF fragment:

```
4: Groups ::= "groups" "{" Group* "}"
5: Group  ::= "group" NCName "=" "{" Regex "}"
            | "attribute-group" NCName "=" "{" AttributePattern "}"
```

Every group needs to have a name to allow references. Besides this, attribute groups contain an attribute pattern and element groups contain a regular expression.
The second place is inside child patterns, where groups can be referenced. For convenience, we reproduce the corresponding EBNF rules here.

```
15: AttributeDecl       ::= "attribute" NCName ("{" TypeRef "}")? "?"?
                          | "attributeref" QName DefaultOrFixedValue?
                          | "attribute-group" NCName

21: ElementDecl         ::= "element" NCName InlineTypeRef?
                          | "elementref" QName
                          | "group" NCName
```

Note that instead of an attribute or element declaration, it is possible to reference a group using the `attribute-group` or `group` keyword and the name of the referenced group.

7.5 Examples

7.5.1 Example of an Element Group

The following example demonstrates the usage of element groups in XML Schema:
With the next XML Schema example we want to show the definition of the element group `A` including the elements `A1`, `A2` and `A3`. This element group `A` is used in the definition of the element `ABC`.

Code-example 7.1: An Example of using group in XML Schema

```
<?xml version="1.0" ?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://example.org/namespace">
   <group name="A">
      <sequence>
         <element name="A1" type="string"/>
         <element name="A2" type="string"/>
         <element name="A3" type="string"/>
      </sequence>
   </group>

   <complexType name="ABC">
         <group ref="A"/>
         <element name="B" type="string"/>
         <element name="C" type="string"/>
   </complexType>
</schema>
```

The following example shows the same description above in BonXai.

Code-example 7.2: Declaration of the group in BonXai

```
target namespace http://example.org/namespace
namespace xs = http://www.w3.org/2001/XMLSchema

global { ABC }
```

```
groups {
   group A = {
      element A1 {type xs:string},
      element A2 {type xs:string},
      element A3 {type xs:string}
   }
}

grammar {
   /ABC = {
      group A,
      element B {type xs:string},
      element C {type xs:string}
   }
}
```

7.5.2 Attribute Group Example

The following example demonstrates the usage of an attribute group in XML Schema. This example represents the entities of a book into attributegroup `bookattributes` which consists of three attributes `attributeA`, `attributeB` and `attribute1` and uses it in the definition of the `book`

Code-example 7.3: Example of using attribute group in XML Schema

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://example.org/namespace">
   <attributeGroup name="bookattributes">
      <attribute name="attributeA" type="string"/>
      <attribute name="attributeB" type="string"/>
      <attribute name="attribute1" type="decimal"/>
   </attributeGroup>

   <complexType name "book">
      <attributeGroup "ref="bookattributes"/>
   </complexType>
</schema>
```

The following example shows the same description above in BonXai

Code-example 7.4: Declaration of the group in BonXai

```
target namespace http://example.org/namespace
namespace xs = http://www.w3.org/2001/XMLSchema

global { book }

groups {
  attribute-group bookattributes = {
    attribute attributeA {type xs:string},
    attribute attributeB {type xs:string},
    attribute attribute1 {type xs:decimal}
```

```
  }
}

grammar {
   /book = {
      attribute-group bookattributes
   }
}
```

8 Namespace Declaration

The main goal of namespaces in XML [1] is to allow different schemas to use the same element names, while having different semantics. As such, namespaces are simply a tool for avoiding name conflicts.

8.1 Target Namespace

The namespace for which the current schema is providing definitions is called the *target namespace*. A single schema file contains one target namespace.
It should be noted, that XML Schema distinguishes between target namespace and default namespace[2], while BonXai only uses the target namespace.
In BonXai schemas all unprefixed element names are always the target namespace, i.e. target namespace and default namespace are always the same. Note that unprefixed attribute names are in no namespace at all, resembling the default of XML. If one wants to declare attributes to be in the target namespace, the target namespace has to be bound to some prefix and the attribute name has to use this prefix. Usually one want to declare these attribute as global in the `global` section in the schema, such that they can be referenced by other schemas.
The following XSD specifies "http://example.org/ns" as its target namespace. This means that, if some other schema wants to import fragments of this XSD, it should refer to "http://example.org/ns".

Code-example 8.1: XML Schema with target namespace "http://example.org/ns"

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://example.org/ns"
   elementFormDefault="qualified">

   <element name="myroot">
      <complexType>
         <sequence>
           <sequence minOccurs="2" maxOccurs="3">
             <element name="a"/>
             <element name="b"/>
           </sequence>
           <element name="c" minOccurs="0"/>
        </sequence>
      </complexType>
   </element>
</schema>
```

As such, an XML document that specifies valid content with respect to the above schema, could look like this:

[1] `http://www.w3.org/TR/2006/REC-xml-names11-20060816/`

[2] The default namespace is the default namespace of the XML file defining the schema. In our examples this is always the XML Schema namespace, such that we do not need to prefix all nodes in the schema definition with the `xs:` prefix.

Code-example 8.2: XML Document

```
<?xml version="1.0"?>
<ns:myroot xmlns:ns="http://example.org/ns">
  <ns:a/><ns:b/><ns:a/><ns:b/><ns:c/>
</ns:myroot>
```

The code fragment `xmlns:ns="http://example.org/ns"` binds the namespace http://example.org/ns to the `ns` namespace prefix. All nodes in the document use this prefix to declare that they are inside the http://example.org/ns naemspace. Alternatively one could define a default namespace using the code fragment `xmlns="http://example.org/ns"` as in the following example:

Code-example 8.3: XML Document

```
<?xml version="1.0"?>
<myroot xmlns="http://example.org/ns">
  <a/><b/><a/><b/><c/>
<myroot>
```

The declaration of a default namespace places all element nodes without an explicit namespace prefix in the http://example.org/ns namespace. Both documents are equivalent.
In BonXai all used namespaces are declared at the beginning of the schema. The syntax is given by the following EBNF rule:

```
2: Namespaces ::= "target namespace" NamespaceUriLiteral
                  ("namespace" NCName "=" NamespaceUriLiteral)*
```

At first always a target namespace needs to be declared using the `target namespace` keyword. Afterwards any number of foreign namespaces can be bound to namespace prefixes using the `namespace` keyword.
The following code fragment shows how to define the above XSD in BonXai.

Code-example 8.4: BonXai schema with target namespace "http://example.org/ns"

```
target namespace http://example.org/ns

global { myroot }

grammar {
    /myroot = { (element a, element b)[2,3], element c? }
}
```

The attentive reader undoubtedly noticed that the BonXai schema does not mention an `elementFormDefault` as the XML Schema does. The `elementFormDefault="qualified"` fragment says that all elements declared in the schema should belong to the target namespace. Otherwise, that is, with `elementFormDefault="unqualified"`, only the global elements would belong to the target namespace and all local elements would belong to no namespace. Since we believe that using `elementFormDefault="qualified"` leads to more useable schemas and since it is difficult to come up with use cases that make sensible use of `elementFormDefault="unqualified"`, we chose to always adopt the `elementFormDefault="qualified"` for BonXai schemas which, by consequence, does not need to be stated anymore in the BonXai schema.

9 Any, AnyAttribute

XML Schema provides the possibility to declare the content of a complex type to be arbitrary using the keyword `any`. One use for example could be to embed a piece of xhtml (or an element from another schema) from a certain namespace. It is possible to constrain the source of the elements that are allowed to be inserted: any content from

- the target (i.e., local) namespace;
- any namespace except the target (local) one;
- any conceivable namespace;
- a certain, exactly declared namespace.

The same functionality applies to `anyAttribute`.

9.1 Any / AnyAttribute by Example

Code-example 9.1: XML Schema fragment illustrating `any`

```
[...]
<element name="book">
   <complexType>
      <sequence>
         <element name="title" type="xsd:string" />
         <element name="author" type="xsd:string" />
         <any processConents="skip"/>
      </sequence>
   </complexType>
</element>
[...]
```

The above XSD fragment specifies a `book` element for which the content is a title, an author, followed by a single, arbitrary XML tree. (The tag "skip" for processContents states that the XML tree is not validated in any way; it should merely be well-formed XML.
A rule with the same functionality in BonXai could be written as follows.

Code-example 9.2: BonXai fragment illustrating `any`

```
/book = {element title {xsd:string}, element author {xsd:string}, any {skip}}
```

The following XML Schema fragment allows a book to have a title and an author element, together with an arbitrary set of attributes.

Code-example 9.3: XML Schema fragment illustrating `anyAttribute`

```
[...]
<element name="book">
   <complexType>
      <sequence>
         <element name="title" type="xsd:string" />
         <element name="author" type="xsd:string" />
    </sequence>
    <anyAttribute processContents="skip"/>
   </complexType>
</element>
[...]
```

An equivalent way of stating this in BonXai would be as follows:

Code-example 9.4: BonXai fragment illustrating `any`

```
/book = {anyAttribute {skip}, element title {xsd:string}, element author {xsd:string}}
```

Instead of `skip`, it is also possible to use `strict` or `lax` with the same semantics as in XML Schema.

9.2 Any / AnyAttribute together with Namespaces

XML Schema allows the use of `any` and `anyAttribute` together with an explicit list of namespaces, or together with the pre-defined keywords `##other`, `##local`, or `##any`. Here we simply provide a few examples of the syntax of these constructs is in BonXai. The semantics is exactly the same as in XML Schema.
The following fragment specifies a `mysvg` element with a title and an author, followed by any content coming from the SVG or XHTML namespace.

Code-example 9.5: BonXai fragment illustrating the use of `any` with namespaces

```
/mysvg = {element title {xsd:string}, element author {xsd:string}, any
  {strict namespace {http://www.w3.org/2000/svg http://www.w3.org/1999/xhtml}}}
```

Instead of the list of namespaces mentioned above, we could also write
`namespace { ##local}`
`namespace { ##other}`
`namespace { ##any}`
with the same meaning as in XML Schema. The syntax for `anyAttribute` is analogous.

10 Foreign References

Element-, attribute-, and type references can be used in BonXai to import content that is defined elsewhere.

10.1 Element- and Attribute References

The following XSD imports the namespace http://www.w3.org/2000/svg for scalable vector graphics and declares that it will prefix all content from this namespace with `svg:`.

Code-example 10.1

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://example.org/ns"
   xmlns:svg="http://www.w3.org/2000/svg"
   elementFormDefault="qualified">

   <element name="aHundredSVGImages">
      <complexType>
         <sequence minOccurs="100" maxOccurs="100">
           <element ref="svg:svg"/>
         </sequence>
      </complexType>
   </element>
</schema>
```

The XSD defines documents with root "aHundredSVGImages", below which we have a sequence of one hundred elements, named `svg` and conforming to the type of the `svg` root element in the http://www.w3.org/2000/svg namespace. A BonXai schema equivalent to the above XSD can be defined as follows:

Code-example 10.2

```
target namespace http://example.org/ns
namespace svg = http://www.w3.org/2000/svg

global { aHundredSVGImages }

grammar {
   /aHundredSVGImages = {
      elementref svg:svg[100]
  }
}
```

Elements and attributes are referenced in BonXai using `elementref` and `attributeref` keywords, respectively.

It should be noted that referencing elements is a way to allow complete XML documents to be inserted into an XML tree. Therefore the referenced elements are always validated as if they were the root of the tree. This holds also true for element references, where the declaration of the foreign element is done in a BonXai schema.
It is also possible to reference the global elements in the target namespace. Note that even in this case, the referenced element is considered to be the root of the subdocument. We have a look at the following BonXai example.

Code-example 10.3

```
arget namespace http://www.example.com/namespace

global { a }

grammar {
  //b = { element a }
  //a = { empty }
  /a  = { elementref a?, element b? }
}
```

In this schema, `a`-elements, which are childs of `b`-elements are always empty, because the ancestor pattern `//a` matches and the ancestor pattern `/a` does not match. However, `a`-elements, which are childs of `a`-elements are allowed to have `a`- and `b`-childs, as they are the root of a referenced subdocument. Therefore their effective ancestor path is only `a`, which is matched by the `/a` ancestor pattern. Remember that lower rules have a higher priority in BonXai.

10.2 Type References

Instead of referring to elements or attributes of a foreign schema, it is also possible to refer to types that are defined elsewhere. The most common use of this feature is to refer to simple types from the "http://www.w3.org/2001/XMLSchema" namespace.
For example, the following BonXai schema uses the simple types "string" and "integer" from the "http://www.w3.org/2001/XMLSchema".

Code-example 10.4

```
target namespace http://example.org/ns
namespace xsd = http://www.w3.org/2001/XMLSchema

global { myRoot }

grammar {
   /myRoot = {
      element firstChild,
      element secondChild
  }

  //firstChild = {type xsd:string}
  //secondChild = {type xsd:integer}
}
```

There are two ways of referring to foreign types. The first one is to use a type reference instead of a child pattern. This is described by the following EBNF rule:

```
26: TypeRef ::= "nillable"? "{" AttributePattern? "type" QName DefaultOrFixedValue? "}"
```

Note that it is possible to define additional attributes for an existing type. The attributes specified in the attribute pattern are merged with already defined attributes. For the description of how to specify an default or fixed value for a simple type, have a look at Chapter 11.
For example the anchor tag (`<a href="link">text</a>`>) of html could be defined with rules like:

```
//a = { attribute href, type xsd:string }
//a/@href = { type xsd:string }
```

The second variant of type references are inlined into child or attribute patterns. Especially for attribute definitions it may be handy to not specify their simple types as seperate rules.
Inline type references are described by the following EBNF rule:

```
22: InlineTypeRef ::= "{" QName DefaultOrFixedValue "}"
```

Inline type references can be used after element and attribute declarations, as can be seen from the EBNF rules for declaring elements and attributes, which we reproduce for convenience:

```
15: AttributeDecl ::= "attribute" NCName InlineTypeRef? "?"? | [...]
21: ElementDecl   ::= "element" NCName InlineTypeRef? | [...]
```

Using an inline type reference, the definintion of the anchor tag could be rewritten using just a single rule like this:

```
  //a = { attribute href {xsd:string}, type xsd:string }
```

Note that the rule uses an inlined type reference for the href attribute and a type reference instead of a child pattern for the anchor tag itself.

11 Default and Fixed

XML Schema allows the definition of default and fixed values for elements and attributes. Default values ensure that the a certain value, which is defined inside the schema, is used if no other data is supplied in the processed instance.
BonXai simply allows the keywords `default` and `fixed` to reproduce this feature set of XML Schema. These keywords may be used when using attribute references and inside type references. Have a look at Chapter 10 for defintions of these features.
The EBNF rule for default and fixed values is as follows:

```
16: DefaultOrFixedValue ::= ("fixed" | "default") """ arbitraryLetterExceptQuote* """
```

11.1 Examples

Code-example 11.1: XML Schema showing default and fixed

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://example.org/namespace"
        elementFormDefault="qualified">

   <element name="booking" type="bookingtype" />

   <complexType name="bookingtype">
      <sequence>
         <element name="name" type="string" />
         <element name="customerID" type="decimal" />
         <element name="land" type="string" fixed="Germany" />
         <element name="address" type="string"/>
         <element name="paymentmethod"  type="string" default="CreditCard"/>
      </sequence>
   </complexType>
</schema>
```

Code-example 11.2: BonXai definition using default and fixed

```
target namespace http://example.org/namespace
namespace xs = http://www.w3.org/2001/XMLSchema

global { booking }


grammar {
 /booking = {
      element name {type xs:string},
      element customerID {type xs:decimal},
      element state { type xs:string fixed "Germany"},
```

```
      element address {type xs:string},
      element paymentmethod {type xs:string default "CreditCard"}
   }
}
```

This element definition of booking has five elements and in one of them, the element `land`, has a fixed value `Germany` and another one, the element `paymentmethod`, has default value `CreditCard`. It means the fixed value of the element `land` is Germany and it cannot be modified another value, the default value of the element `paymentmethod` is here `CreditCard` and it appears if no another value is specified. The following example shows a possible booking instance. In this example the value of element `state` is `Germany` because the value of element `land` is fixed to the value `Germany` and the value of element assigns here as `PayPal` but if no value is specified, the would be automatically set to CreditCard.

Code-example 11.3: A possible instance of the example above

```
<?xml version="1.0?">
<booking xmlns="http://example.org/namespace">
   <name> Max Mustermann </name>
   <customerID> 001 </customerID>
   <land> Germany </land>
   <address> mystreet 1, 44339/Dortmund </address>
   <paymentmethod> PayPal </paymentmethod>
</booking>
```

12 Identity Constraints: Key, Keyref and Unique

BonXai allows to express the same integrity constraints as XML Schema (i.e., unique, key, and keyref). All three keywords "unique", "key", and "keyref" are taken from XML Schema and have the same intended meaning. As in XML Schema, keys should have a name, so that keyrefs can refer to them. Note that in BonXai unique constraints and keyrefs do not have names.
The general syntax of key constraints is

```
key <name> <ancestor pattern> { <selector> { <fields> } },
```

where the ancestor pattern is used to select the elements for which the key should be defined and selector and fields have the same meaning as in XML Schema. The syntax for unique constraints is the same, except that unique constraints do not have a name. The syntax for keyref contains the name of the referenced key instead of its own name.

```
27: Constraints         ::= "constraints" "{" Constraint+ "}"
28: Constraint          ::= UniqueConstraint | KeyConstraint | KeyRefConstraint
29: UniqueConstraint    ::= "unique" KeyPattern
30: KeyConstraint       ::= "key" NCName KeyPattern
31: KeyRefConstraint    ::= "keyref" NCName KeyPattern
32: KeyPattern          ::= APattern "{" Selector "{" Field ("," Field)* "}" "}"
```

The definitions of `Selector` and `Field` are taken from XML Schema.

12.1 Namespace Usage in Identity Constraint Definitions

In XML Schema definitions, neither the default nor the target namespace is applied to selector and field expressions, that is all unprefixed element names are in no namespace. As a consequence, all elements in the target namespace needs to have a namespace prefix, which needs to be bound to the target namespace. Different to this, in BonXai, the target namespace is used for all unprefixed element names reflecting our default choice of qualifying all element in the schema.

12.2 Example for Key and Keyref

XML Schema

An XML Schema example for defining keys and keyrefs. We want to represent a small shop which of course consists of a set of articles belonging to different (predefined) categories.

Code-example 12.1

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://example.org/namespace"
   xmlns:ns="http://example.org/namespace"
```

```
   elementFormDefault="qualified">

   <element name="shop">
      <complexType>
         <sequence>
            <element name="categories">
               <element name="category" type="string" maxOccurs="unbounded" />
            </element>

            <element name="articles">
               <complexType>
                  <sequence>
                     <element name="article" maxOccurs="unbounded">
                        <complexType>
                           <sequence>
                              <element name="category" type="string" />
                              <element name="name" type="string" />
                           </sequence>
                        </complexType>
                     </element>
                  </sequence>
               </complexType>
            </element>
         </sequence>
      </complexType>

      <key name="categoryKey">
         <selector xpath="./ns:categories/ns:category" />
         <field xpath="." />
      </key>
   [...]
```

The categories defined in the first part of this schema-definition (addressed with an XPath expression) are now used as key and can thus be referenced from somewhere else.

Code-example 12.2

```
   [...]
      <keyref name="categoryKeyRef" refer="categoryKey">
         <selector xpath="./ns:articles/ns:article/ns:category" />
         <field xpath="." />
      </keyref>
   </element>
</schema>
```

`keyref` finally connects an article's category (marked by `selector` and also adressed with an XPath expression) to the key `categoryKey`.
We provide BonXai schema for the same shop. Categories and articles are defined in the grammar-block at the beginning.

Code-example 12.3

```
target namespace http://example.org/namespace

global {shop}
```

```
grammar {
  /shop = {
      element categories,
      element articles
   }
   /shop/categories = {
      element (category)+
   }
   /shop/articles = {
      element (article)+
   }

   category = {type string }
   article = {
      element category,
      element name
   }

   article/name = {type string }
}
[...]
```

The following `contraints-block` contains the information about `key`/`keyref`-pairs (and could also contain information about unique-constraints as the following example will show). The structure of the definition is the same as in XML Schema, just translated into the BonXai syntax.

Code-example 12.4: BonXai key example

```
[...]
constraints {
   key categoryKey /shop {
      ./categories/category {
         .
      }
   }
[...]
```

The ancestor path behind the `key` keyword and the name of the key defines the location of the key. In the XML Schema example above the key was located in the global schema element, so the key is bound to the whole schema, indicated by the `/shop` ancestor pattern in BonXai.
The key section itself consists of a single selector, a simplified XPath expression, just like in XML Schema, and a comma separated list of fields, since multiple fields are allowed to express keys spanning multiple attributes. In this example only the element content is used as a key value, but the field definition could also look like `., @id` to additionally include an `id` attribute.
The key reference is defined in nearly the same way:

Code-example 12.5: BonXai keyref example

```
[...]
   keyref categoryKey /shop {
      ./articles/article/category {
         .
      }
```

```
   }
}
```

The name behind the `keyref` keyword now defines the referenced key, and thus contains the same name like the key defined before. The structure of the selector and the fields is just the same like in the key example.
As mentioned before the name attribute of the `keyref` is omitted, since it serves no use.

12.3 Example for Unique

In this section, we will describe the usage of `unique` like described above in XML Schema and BonXai.

XML Schema

Code-example 12.6

```
   [...]
      <element name="category" maxOccurs="unbounded">
         <complexType>
            <simpleContent>
               <extension base="string">
                  <attribute name="id" type="string" />
               </extension>
            </simpleContent>
         </complexType>
      </element>
```

This definition of `category` replaces the short one of our first example, simply adding an `id` of type string. This will be forced to be `unique` with the following expression:

Code-example 12.7

```
      <unique name="myUnique">
         <selector xpath="./ns:articles/ns:article/ns:category" />
         <field xpath="@id" />
      </unique>
```

In XML Schema, a name for the unique-specification must be given, in this example `myUnique`. As in `key` constraints, the selector XPath expression defines the path to the element(s) that should be unique according to the fields specified by the field expressions.

BonXai

Code-example 12.8

```
target namespace http://example.org/namespace

global { shop }

grammar {
   /shop = {
      element categories
```

```
   }

   /shop/categories = {
      element (category)+
   }

   category = {
      attribute id {type string},
      string
   }
}

constraints {
   unique /shop {
      ./articles/article/category {
         @id
      }
   }
}
```

The structural definition is just the same as for the `key` and `keyref` definitions.

13 Annotations

One can make annotations in BonXai schemas with the syntax

```
Annotation ::= "@"QName ("=" AnnotationValue)?
```

Annotations are different from comments (which are marked by using the character `#` at the beginning of a line) in the sense that comments are ignored by the parser whereas annotations are not. (The difference between annotations and comments in XML Schema is similar.)
Annotations in BonXai can be used for providing rules with names, which can be used to generate type names when converting BonXai to XML Schema.
BonXai does not use complex type definitions with names, as XML Schema does. To ensure that type names do not get lost when converting BonXai to XML Schema, we use BonXai annotations to store them. As such, the user can identify complex types in XML Schema with the corresponding BonXai rules very quickly.
In the other direction, it is possible to provide BonXai rules with type names (if desired) which can be used to name the corresponding complex types in the conversion to XML Schema.

13.1 Examples

Code-example 13.1: XML Schema containing a complex type called BookingType

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://example.org/namespace"
        elementFormDefault="qualified">

   <element name="booking" type="BookingType" />

   <complexType name="BookingType">
      <sequence>
         <element name="name" type="string" />
         <element name="customerID" type="decimal" />
         <element name="land" type="string"/>
         <element name="address" type="string"/>
         <element name="paymentmethod"  type="string"/>
      </sequence>
   </complexType>
</schema>
```

When converting the above XML Schema to BonXai, one can store the name “BookingType” of the complex type in an annotation of the corresponding BonXai rule. The name of this complex type has no semantic meaning whatsoever in BonXai, but if it is stored in an annotation it simply does not get lost.

Code-example 13.2: BonXai definition containing an annotation

```
target namespace http://example.org/namespace
namespace xs = http://www.w3.org/2001/XMLSchema
namespace ann = http://myannotationsnamespace

global { booking }

grammar {
  @ann:typename = BookingType
  /booking = {
      element name {type xs:string},
      element customerID {type xs:decimal},
      element state { type xs:string},
      element address {type xs:string},
      element paymentmethod {type xs:string}
   }
}
```

Therefore, when converting the above mentioned BonXai schema back to XML Schema, it is possible to use the name `BookingType` again for the corresponding complex type definition.

14 BonXai Syntax

We provide syntax of BonXai as EBNF.

```
 1: Bonxai              ::= Namespaces Global Groups? Grammar Constraints?

 2: Namespaces          ::= "target namespace" NamespaceUriLiteral
                            ("namespace" NCName "=" NamespaceUriLiteral)*

 3: Global              ::= "global" "{" ( NCName | "@"NCName )* "}"

 4: Groups              ::= "groups" "{" Group* "}"
 5: Group               ::= "group" NCName "=" "{" Regex "}"
                          | "attribute-group" NCName "=" "{" AttributePattern "}"

 6: Grammar             ::= "grammar" "{" Rule* "}"
 7: Rule                ::= Annotation* AncestorPattern "=" ( ChildPattern | TypeRef )

 8: AncestorPattern     ::= FAPattern | APattern
 9: APattern            ::= ("@"? (NCName | "*")) FAPattern?
                          | "(" APattern ("|" APattern)* ")"
10: FAPattern           ::= ( "/" | "//" ) APattern
                          | "(" FAPattern ("|" FAPattern)* ")" RepetitionOp? FAPattern?
11: RepetitionOp        ::= "*" | "+" | "?"

12: ChildPattern        ::= MixedNillable? "{" AttributePattern? ElementPattern "}"
13: MixedNillable       ::= "mixed" "nillable"? | "nillable" "mixed"?

14: AttributePattern    ::= ("anyAttribute" WildcardDecl? ",")? (AttributeDecl ",")*
15: AttributeDecl       ::= "attribute" NCName InlineTypeRef? "?"?
                          | "attributeref" QName DefaultOrFixedValue?
                          | "attribute-group" NCName
16: DefaultOrFixedValue ::= ("fixed" | "default") """ arbitraryLetterExceptQuote* """

17: ElementPattern      ::= "empty" | AllPattern | Regex

18: AllPattern          ::= ElementDecl "?"? ( "&" ElementDecl "?"? )+
19: Regex               ::= Regex Counter?
                          | Regex "," Regex
                          | Regex "|" Regex
                          | "(" Regex ")"
                          | ElementDecl
                          | "any" WildcardDecl?
```

```
20: Counter             ::= "*" | "+" | "?"
                          | "[" Number "]"
                          | "[" Number "," Number "]"
                          | "[" Number "," "*" "]"

21: ElementDecl         ::= "element" NCName InlineTypeRef?
                          | "elementref" QName
                          | "group" NCName

22: InlineTypeRef       ::= "{" QName "nillable"? DefaultOrFixedValue "}"

23: WildcardDecl        ::= "{" Policy? ("namespace" "not"? NamespaceList)?
                                    ("name" "not" Namelist)?  "}"
24: NamespaceList       ::= "{" (NamespaceUriLiteral | ##other | ##local | ##any)* "}"
25: Policy              ::= "strict" | "lax" | "skip"

26: TypeRef             ::= "nillable"? "{" AttributePattern?
                            "type" QName DefaultOrFixedValue? "}"

27: Constraints         ::= "constraints" "{" Constraint+ "}"
28: Constraint          ::= UniqueConstraint | KeyConstraint | KeyRefConstraint
29: UniqueConstraint    ::= "unique" KeyPattern
30: KeyConstraint       ::= "key" NCName KeyPattern
31: KeyRefConstraint    ::= "keyref" NCName KeyPattern
32: KeyPattern          ::= APattern "{" Selector "{" Field ("," Field)* "}" "}"

33: Annotation          ::= AnnotationName ("=" AnnotationValue)?
34: AnnotationName      ::= "@"["a"-"z", "A"-"Z"]+
35: AnnotationValue     ::= ~["\r", "\n"]+

36: Number              ::= Digit+
37: QName               ::= /* see: XMLSchema */
38: NCName              ::= /* see: XMLSchema */
39: NamespaceUriLiteral ::= /* see: XMLSchema */
40: Selector            ::= /* see: XMLSchema */
41: Field               ::= /* see: XMLSchema */

################################################
# The following definitions are taken directly form the according w3c
# specification which can be found at: http://www.w3.org/TR/REC-xml/#NT-Letter
################################################

42: Letter              ::= /* http://www.w3.org/TR/REC-xml/#NT-Letter */
43: CombiningChar       ::= /* http://www.w3.org/TR/REC-xml/#NT-Letter */
44: Extender            ::= /* http://www.w3.org/TR/REC-xml/#NT-Letter */
45: Digit               ::= /* http://www.w3.org/TR/REC-xml/#NT-Letter */
46: URL                 ::= /* RFC 1738 - Uniform Resource Locators (URL) */
```

Bibliography

[1] W. Gelade and F. Neven. Succinctness of pattern-based schema languages for XML. In M. Arenas and M. I. Schwartzbach, editors, *DBPL*, volume 4797 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 2007.

[2] G. Kasneci and T. Schwentick. The complexity of reasoning about pattern-based XML schemas. In *PODS*, pages 155–164, 2007.

[3] W. Martens, F. Neven, M. Niewerth, and T. Schwentick. BonXai: Combining the simplicity of DTD with the expressiveness of XML Schema. Manuscript, 2014.

[4] W. Martens, F. Neven, and T. Schwentick. Simple off the shelf abstractions for xml schema. *SIGMOD Record*, 36(3):15–22, 2007.

[5] W. Martens, F. Neven, T. Schwentick, and G. J. Bex. Expressiveness and complexity of XML schema. *ACM Transactions on Database Systems*, 31(3):770–813, 2006.

[6] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema part 1: Structures second edition. Technical report, World Wide Web Consortium (W3C), October 2004. http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.