XML and Databases

Chapter 7: XML Schema III: Reference

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Winter 2019/20

http://www.informatik.uni-halle.de/~brass/xml19/

Objectives

After completing this chapter, you should be able to:

- write an XML schema for a given application.
- check a given XML schema definition for syntactic correctness.
- check given XML documents for validity according to a given XML schema.

Inhalt

Content Models (1)

- Content models are used to describe the sequence of elements that are nested inside an element (child elements).
- Content models in XML Schema offer basically the same possibilities as content models in DTDs:
  - `sequence`: Corresponds to "`,`" in DTDs.
  - `choice`: Corresponds to "`|`" in DTDs.
  - `all`: Corresponds to "`&`" in (SGML) DTDs.

    `all` means: The elements in the group must occur (unless `minOccurs=0` for that element), but the order is arbitrary (any permutation is permitted). In XML, "`&`" was removed from the SGML DTD syntax.
- The attributes `minOccurs` and `maxOccurs` take the place of "`?`", "`*`", "`+`" in DTDs.

Content Models (2)

`<sequence>`, `<choice>`:

- Possible attributes:
  - `minOccurs`: Minimum number of times the group must occur (`nonNegativeInteger`)

    The default value of both, `minOccurs` and `maxOccurs`, is 1.

  - `maxOccurs`: Maximum number of times the group may occur (`nonNegativeInteger` or "`unbounded`")
- Content model:

  `annotation?, (element|group|choice|sequence|any)*`
- Possible parent elements: `complexType`, `restriction`, `extension`, `group`, `choice`, `sequence`.

Content Models (3)

`<all>`:

- Possible attributes:
  - `minOccurs`: Minimum number of times the group must occur (`0` or `1`, default: `1`)
  - `maxOccurs`: Maximum number of times the group may occur (`1` is the only legal value)
- Content model:

  `annotation?, element*`

  `all` is very restricted compared with `choice` and `sequence`: It can only directly contain elements, no other groups, and it must appear on the outermost level. The elements it contains must have `maxOccurs=1` (which is the default).
- Possible parent elements: `complexType`, `restriction`, `extension`, `group`.

Content Models (4)

- Like XML DTDs, XML Schema requires deterministic content models, e.g. this is not permitted:

```
<!-- Invalid! Corresponds to (A | (A, B)) -->
<xs:complexType name="nondeterministic">
  <xs:choice>
    <xs:element name="A" type="xs:string"/>
    <xs:sequence>
      <xs:element name="A" type="xs:string"/>
      <xs:element name="B" type="xs:string"/>
    </xs:sequence>
  </xs:choice>
</xs:complexType>
```

Content Models (5)

`<element>` (element reference):

- Possible attributes:
  - `ref`: Name of the element being referenced (a `QName`, the element must be declared globally).

    If `element` is used as element reference, this attribute is required. The element may be declared later or (if it does not occur in the data) may be declared not at all.
  - `minOccurs`, `maxOccurs`: (see above)
- Content model: `annotation?`
- Possible parent elements: `all`, `choice`, `sequence`.

Named Model Groups (1)

- It is possible to introduce a name for a model group, and to use this “named model group” as part of other model groups (like macro/parameter entity).

  Thus, if one must declare several element types that have in part equal content models, it suffices to define the common part only once. If one wants to define a common part only once without named model groups, one needs an element as a container for this part (additional nesting in data).

- Advantages:
  - Helps to ensure the consistency of similar content models.

    This especially holds also for later changes (can be done in one place).
  - Makes equal parts obvious in the schema.
  - The schema becomes shorter.
  - Permits reusable components below element / complex type.

Named Model Groups (2)

`<group>` (named model group definition):

- Possible attributes:
  - `name`: Name of the model group being defined (an `NCName`).

    If `group` is used for defining a named model group, this attribute is required.
- Content model:
  ```
  annotation?, (all | choice | sequence)
  ```
- Possible parent element types: `schema`, `redefine`.

Named Model Groups (3)

`<group>` (named model group reference):

- Possible attributes:
  - `ref`: Name of the model group being referenced (a `QName`).

    If `group` is used to refer to a named model group, this attribute is required.

  - `minOccurs`, `maxOccurs`: (see above)
- Content model: `annotation?`
- Possible parent elements: `complexType`, `restriction`, `extension`, `choice`, `sequence`.

Wildcard (1)

- With "`<any>`" it is possible to allow arbitrary elements (one can restrict the namespace).
- E.g., to permit arbitrary XHTML in a product description (without explicitly listing elements):

```
<xs:complexType name="Description" mixed="true">
  <xs:sequence>
    <xs:any minOccurs="0" maxOccurs="unbounded"
        namespace="http://www.w3.org/1999/xhtml"
        processContents="skip"/>
  </xs:sequence>
</xs:complexType>
```

Wildcard (2)

- With the attribute `processContents`, one can select whether the contents of elements inserted for the wildcard should be checked:
  - `skip`: Only the well-formedness is checked.
  - `lax`: If the XML Schema processor can find declarations of the elements, it will validate their contents.

    Otherwise no warning is printed.
  - `strict`: Full validation.
- A wildcard is a “quick&dirty” solution.

  In this case, `processContents` was set to “`skip`”. But even if it were set to “`strict`”, this would not prevent XHTML elements like `meta` (intended for the head). The only safe solution is probably to explicitly list the allowed XHTML elements. With a bit of luck, the schema for XHTML contains a named model group that can be used.

Wildcard (3)

- With the attribute `namespace`, one can restrict the namespace of the elements matched with `any`:
  - `##any`: no restriction (this is the default).
  - `##other`: any namespace except the target namespace of this schema.

    In this case, it is required that the elements have a namespace.
  - List of URIs, "`##local`", "`##targetNamespace`": Only these namespaces are permitted.

    "`##local`" allows elements without a namespace,

    "`##targetNamespace`" allows elements from this schema.

Wildcard (4)

`<any>`:

- Possible attributes:
  - `namespace`: Restrictions for the namespace of the elements inserted for the wildcard.

    See Slide 14. The default is no restriction ("`##any`").
  - `processContents`: Defines whether the contents of elements matched with "`any`" is checked.

    See Slide 13. The default is "`strict`".
  - `minOccurs`, `maxOccurs`: (see above)
- Content model:

  `annotation?`
- Possible parent element types: `choice`, `sequence`.

Inhalt

Complex Types (1)

- Complex types are used to define the characteristics of elements (content model and attributes).
- However, if an element has no attributes and no element content (only a string, number, etc.), a simple type suffices.
- There are two ways to use complex types:
  - Define a named complex type and reference it in the `type`-attribute of `element`.
  - Define an anonymous complex type as a child of `element`.

Complex Types (2)

- The possibilities for defining a complex type are:
  - List the content model (see above), followed by the attributes (see below).

    If the content model is missing (empty), elements of this type have empty content. If the attribute part is not used, elements of this type have no attributes.
  - Use a `simpleContent` child: For type derivation.

    This is used for deriving a complex type from a simple type (by adding attributes), or from another complex type with simple content (by restriction or extension). See last section of this chapter.
  - Use a `complexContent` child: For type derivation.

    This is used for restricting or extending a complex type with element content. See last section of this chapter.

Complex Types (3)

`<complexType>` (anonymous type):

- Possible attributes:
  - `mixed`: Can additional character data appear between the elements of the content model?

    Used for specifying mixed content models. The default is `false`.
- Content model:

  ```
  annotation?, (simpleContent | complexContent |
                   ((all|choice|sequence|group)?,
                    (attribute|attributeGroup)*,
                    anyAttribute?))
  ```
- Possible parent element types: `element`.

Complex Types (4)

`<complexType>` (for defining a named type):

- Possible attributes:
  - `name`: Name of the type to be defined (`NCName`).
  - `mixed`: For mixed content models, see above.
  - `abstract`, `block`, `final`: See following slides.
- Content model:

```
annotation?, (simpleContent | complexContent |
                 ((all|choice|sequence|group)?,
                  (attribute|attributeGroup)*,
                  anyAttribute?))
```

- Possible parent element types: `schema`, `redefine`.

Complex Types (5)

Attribute `final` (forbids type derivation):

- One can forbid that other types are derived from this type. Possible values of the attribute `final` are:
  - `"#all"`: There cannot be any derived type.

    `"extension restriction"` (in either order) is equivalent.
  - `"extension"`: Type derivation by extension is excluded, type derivation by restriction is possible.
  - `"restriction"`: Conversely.
  - `""`: Both forms of type derivation are possible.

    If `final` is not specified, the value of the attribute `finalDefault` of the `schema`-element is used (which in turn defaults to `""`).

Complex Types (6)

Attribute `block` (forbids type substitution):

- If an element type E is declared with a complex type C, and C' is derived from C, elements of type E can state that they are really of type C' (with `xsi:type=`C'), and e.g. use the additional attributes or child elements of type C'.
- The attribute `block` can be used to prevent this.

  Possible values are: `"#all"` (i.e. type substitution is not permitted), `""` (i.e. type substitution is possible), `"restriction"` (i.e. only types defined by extension can be used), `"extension"` (i.e. only types defined by restriction can be used), `"extension restriction"` (in either order: same as `"#all"`). The default is `blockDefault` in the `schema`-element, which in turn defaults to `""` (no restriction).

Complex Types (7)

Attribute `abstract` (forbids instantiation):

- If `abstract` is `"true"`, no elements can have this complex type.

  The default value is `"false"`.

- Thus the type is defined only as a basis for type derivation.

  Actually, one can define element types of an abstract complex type, but then type substitution must be used for all elements of this type.

- This corresponds to the notion of abstract superclasses in object-oriented programming.

Derived Complex Types (1)

`<simpleContent>/<complexContent>`:

- Possible attributes:
  - `mixed` (only for `complexContent`): Is character data is allowed between child elements?

    Possible values are `true` (for mixed content models) and `false` (else). The default value is the value in the enclosing `complexType` element, which defaults to `false`. This attribute in `complexContent` is simply an alternative to specifying it in `complexType`.

- Content model:

  ```
  annotation?, (extension | restriction)
  ```

- Possible parent element types: `complexType`.

Derived Complex Types (2)

`<extension>` (inside `<simpleContent>`):

- Possible attributes:
  - `base`: The base type that is extended to define a new type (`QName`, required).

    For an extension inside `simpleContent`, the base type must be a simple type, or a complex type derived from a simple type (i.e. with a simple type as content).

- Content model:

  ```
  annotation?,
  (attribute | attributeGroup)*, anyAttribute?
  ```

- Possible parent element types: `simpleContent`.

Derived Complex Types (3)

`<extension> (inside <complexContent>):`

- Possible attributes:
  - `base`: The base type that is extended to define a new type (`QName`, required).

    For an extension inside `complexContent`, the base type must be a complex type, i.e. it must have element, mixed, or empty content.
- Content model:

  ```
  annotation?,
  (group | all | choice | sequence)?,
  (attribute | attributeGroup)*, anyAttribute?
  ```
- Possible parent element types: `complexContent`.

Inhalt

Attributes (1)

- Elements can have attributes, therefore complex types must specify which attributes are allowed or required, and which data types the attribute values must have.
- Attributes can be declared
  - globally, and then referenced in complex types,
  - locally within a complex type (immediately used, never referenced).

    This is a counterpart to “anonymous types” which are defined when they used (and cannot be reused). However, attributes always have a name.

Attributes (2)

- If a target namespace is declared for the schema, globally declared attributes are in this namespace.
- Thus, they need an explicit namespace prefix in each occurrence in the data file.

  Default namespaces do not apply to attributes.

- For locally declared attributes, one can choose whether they must be qualified with a namespace.

  This is done with the `form` attribute (`"qualified"` or `"unqualified"`). A default can be set with the `attributeFormDefault`-attribute of the `schema`-element. If this is not set, the default is `"unqualified"`, i.e. the attribute is used without namespace prefix.

Attributes (3)

- Since one usually does not want to specify a namespace prefix, global attribute declarations are seldom used.

  Global attributes with a namespace prefix are typically used when many or all elements can have this attribute.

- If several elements/complex types have the same attribute, one can define an attribute group (see below), in order to specify the characteristics of the attribute only once.

  When the attribute group is used, it becomes a local declaration (it works like a parameter entity/macro).

Attributes (4)

- As in DTDs, one can specify a default or fixed value for an attribute.

  Fixed values are mainly interesting for global attributes, see Chapter 1.

- If the attribute does not occur in the start tag of an element, the XML Schema processor automatically adds it with the default/fixed value.

  Thus the application gets this value. Attributes with fixed value can have only this single value and usually do not appear in the data file.

- In XML Schema, default/fixed values are specified with the attributes `default`/`fixed` of `attribute` elements. These attributes are mutually exclusive.

Attributes (5)

- As in DTDs, one can specify whether an attribute value must be given in every start tag or not.

  In XML DTDs, the alternatives are: (1) a default value, (2) `#REQUIRED`, (3) `#IMPLIED` (meaning optional), and (4) `#FIXED` with a value.

- In XML Schema, this is done with the attribute "`use`". It can have three possible values:
  - `"optional"`: Attribute can be left out.
  - `"required"`: Attribute value must be given.

    This cannot be used together with a default value.
  - `"prohibited"`: Attribute value cannot be specified.

    This is only used for restricting complex types, see below.

Attributes (6)

`<attribute>` (attribute reference):

- Possible attributes:
  - `ref`: Name of the attribute (`QName`, required).
  - `use`: `"optional"`, `"required"`, or `"prohibited"`.

    The default is `"optional"`, i.e. the attribute can be left out.
  - `default`: Default value for the attribute.
  - `fixed`: Fixed value for the attribute.
- Content model: `annotation?`
- Possible parent element types:
  `complexType`, `restriction`, `extension`, `attributeGroup`.

Attributes (7)

`<attribute>` (global attribute declaration):

- Possible attributes:
  - `name`: Name of the declared attribute (`NCName`).

    This attribute is required.
  - `type`: Data type of the attribute (`QName`).

    This attribute is mutually exclusive with the `simpleType` child.

    If neither is used, the default is `anySimpleType` (no restriction).
  - `default`: Default value for the attribute.
  - `fixed`: Fixed value for the attribute.
- Content model: `annotation?, simpleType?`
- Possible parent element types: `schema`.

Attributes (8)

`<attribute>` (local attribute declaration):

- Possible attributes:
  - `name`: Name of the attribute (`NCName`, required).
  - `type`: Data type of the attribute (`QName`).
  - `form`: `"qualified"` or `"unqualified"` ($\rightarrow$ 29).
  - `use`: `"optional"`, `"required"`, or `"prohibited"`.
  - `default`, `fixed`: see above.
- Content model: `annotation?, simpleType?`
- Possible parent element types:
  `complexType`, `restriction`, `extension`, `attributeGroup`.

Attributes (9)

Constraint on Attributes within a Complex Type:

- A complex type cannot have more than one attribute with the same name.

  This is not surprising, because the XML standard requires this already for well-formed XML. Note that the qualified name counts: One could have attributes with the same name in different namespaces.

- A complex type cannot have more than one attribute of type `ID`.

  Also this is a restriction given by the XML standard (although only for DTDs, maybe one could have removed it in XML Schema, but XML Schema anyway has more powerful identification mechanisms). Note also that attributes of type `ID` cannot have default or fixed values.

Attributes (10)

Attribute Wildcard:

- One can permit that the start tags of an element type can contain additional attributes besides the attributes declared for that element type.

  Actually, certain attributes such as namespace declarations, and `xsi:*` are always allowed, and do not have to be explicitly declared.

- This is done by including the attribute wildcard "`<anyAttribute>`" in the complex type definition.
- The wildcard matches any number of attributes.

  This is a difference to the element wildcard `<any>`. Thus, it makes no sense to specify `<anyAttribute>` more than once in a complex type.

Attributes (11)

`<anyAttribute>`:

- Possible attributes:
  - `namespace`: Restrictions for the namespace of the attributes inserted for the wildcard.

    See Slide 14. The default is no restriction ("`##any`").
  - `processContents`: Defines whether the value of the additional attributes is type-checked.

    See Slide 13. The default is "`strict`".
- Content model: `annotation?`
- Possible parent element types: `complexType`, `restriction`, `extension`, `attributeGroup`.

Attributes (12)

Attribute Groups:

- If several complex types have attributes in common, one can define these attributes only once in an attribute group (example see next slide).

  Since elements / complex types cannot have two attributes with the same name, also attribute groups cannot contain attributes with the same name. In the same way, multiple `ID`-attributes are forbidden.

- This attribute group can then be referenced in a complex type, or in other attribute groups.
- Like model groups, attribute groups are similar to a special kind of parameter entity.

Attributes (13)

- Example for attribute group definition (`CAT`, `ENO`):

```
<xs:attributeGroup name="exIdent">
  <xs:attribute name="CAT" use="required">
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="H"/>
        <xs:enumeration value="M"/>
        <xs:enumeration value="F"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="ENO" use="required"
      type="xs:positiveInteger"/>
</xs:attributeGroup>
```

Attributes (14)

- A reference to the attribute group "`exIdent`" (see previous slide) looks as follows:

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

- The attributes of the attribute group (e.g., `CAT` and `ENO`) are inserted in place of the group reference.

  This is basically done like the expansion of a macro/entity. However, a complex type can contain only one attribute wildcard. In XML Schema, it was decided that referencing two attribute groups that both contain wildcards in the same complex type is no error. In this case, the namespace constraints are intersected, and the `processContents`-value of the first group is chosen (a wildcard directly in the complex type counts as first).

Attributes (15)

`<attributeGroup>` (attribute group definition):

- Possible attributes:
  - `name`: Name of the attribute group (`NCName`).

    This attribute is required.
- Content model:

  ```
  annotation?,
  (attribute|attributeGroup)*, anyAttribute?
  ```
- Possible parent element types: `schema`, `redefine`.

Attributes (16)

`<attributeGroup>` (attribute group reference):

- Possible attributes:
  - `ref`: Name of the attribute group (`QName`).

    This attribute is required.
- Content model:

  ```
  annotation?
  ```
- Possible parent element types:
  `complexType`, `restriction`, `extension`, `attributeGroup`.

Inhalt

Elements (1)

- The main purpose of an element declaration is to introduce an element type name and associate it with a (simple or) complex type.

  In addition, they can define a default/fixed value for the content, permit or forbid a nil value, define keys or foreign keys, block type substitution, and define substitution groups. See below.

- Simple and complex types together are called data types (to distinguish them from “element types”).

  At least in the book “Definitive XML Schema”. The Standard uses simply “type” (for simple and complex type) and avoids the word “element type”. On my slides, I sometimes incorrectly use “element” instead of “element type”. Maybe, “element name” would be good.

Elements (2)

- The association of the declared element type with the simple/complex type can be done in two ways:
  - By including a `simpleType` or `complexType` child element (anonymous type definition).
  - By referencing a named (globally declared) simple or complex type with the `type`-attribute.

  The two possibilities are mutually exclusive.

- If none of the two is used, the element type is associated with `anyType`, and permits arbitrary (well-formed) content and arbitrary attributes.

  Unless the element type is part of a substitution group, see below.

Elements (3)

- Element declarations can be
  - global (later referenced by the element name),

    For element references, see above ("Content Models": 8).
  - local inside a complex type declaration (immediately used and never referenced again).
- As with attributes,
  - globally declared element types always belong to the target namespace of the schema,
  - whereas one can choose whether locally declared element types belong to the target namespace or remain unqualified (no namespace).

Elements (4)

- The namespace decision for local element declarations is done with the attribute `form`. It can be
  - `"qualified"`: The element type name belongs to the target namespace of the schema.
  - `"unqualified"`: The element type name has no namespace.

If a local element type declaration does not contain the `form`-attribute, the default is defined with `elementFormDefault` in the `schema`-element. This in turn defaults to `"unqualified"`. The possibility to switch nearly all element types between unqualified and qualified form with a single attribute setting is one aspect of the "Venetian Blind" design.

Elements (5)

- The namespace of elements can be defined implicitly with a default namespace declaration.

  Important difference to attributes: For elements, it is no problem if every element belongs to a namespace (if it is the same namespace).

- However, the user of a schema must know which elements belong to a namespace and which not.
- One should use a simple rule, e.g.
  - The root element belongs to the target namespace of the schema, the others not.
  - All elements belong to the target namespace.
  - The schema has no target namespace.

Elements (6)

- Global declarations must be used:
  - for the possible root element type(s),
  - for element types that participate in substitution groups (see below).
- Local declarations must be used:
  - if the same element type has different attributes or content models depending on the context.

    It might be better to say if there are different element types with the same name.
  - if the element type name should be unqualified.

    And at least one name in the schema needs a namespace.

Elements (7)

Default and Fixed Values:

- Whereas in DTDs, one can specify default and fixed values only for attributes, in XML Schema, this is possible for attributes and elements.
- However,
  - for an attribute, the default/fixed value is automatically added if the attribute is missing,
  - for an element, the element must still be present, but with empty content.

    In both cases, the validation adds data to the data explicitly given in the input document. This might simplify the application.

Elements (8)

- Only values of simple types can be specified as default/fixed values.

  This is a technical restriction, because default/fixed values are specified in an attribute. But probably default/fixed values for elements were mainly added to make attributes and elements with simple content more similar/interchangable.

- Of course, the default/fixed value must be legal for the declared element content.

  Thus, default/fixed values can be used only for elements with simple content, or mixed content when all child elements are optional.

Elements (9)

- If a default value is declared, there is no way to enter the empty string as element content.

  Then the element is empty, and the default value is added. If the `whitespace`-facet is `collapse`, the default value is added even if there are spaces between start and end tag. But see `xsi:nil` below.

- Note that empty elements can have attributes.

  The default value added as long as the contents is empty.

- A fixed value is very similar to a default value, with the additional constraint that if a value is explicitly specified, it can be only this value.

  Possibly a different lexical representation of the same value.

Elements (10)

Nil:

- Also “nil values” are possible for element content if the element type declaration contains

  ```
  nillable="true"
  ```

  The default value is false.

- This is probably similar to a null value in databases.

  The specific meaning of the nil value depends on the application (i.e. is not defined by XML Schema). The nil value is different from the empty string (and from the missing element).

- Fixed values cannot be combined with `nillable`.

Elements (11)

- In the input document, elements with nil content are marked with `xsi:nil="true"`.

  Where `xsi` is mapped to `http://www.w3.org/2001/XMLSchema-instance`. Note that the attribute `xsi:nil` can be used even if it is not declared for the element type (if the element type is `nillable`).

- In this case, the element content must be empty (but the element can still have attributes).
- It is not required that the element type permits an empty content (but it must be `nillable`).
- If an element is nil, a default value is not added, although the contents looks empty (it is nil).

Elements (12)

`<element>` (global element type declaration):

- Possible attributes:
  - `name`: Element type name (`NCName`, required).
  - `type`: Name of simple or complex type (`QName`).
  - `default`, `fixed`, `nillable`: see above.
  - `abstract`, `substitutionGroup`, `block`, `final`: see below.
- Content model:
  ```
  annotation?, (simpleType | complexType)?
               (key | keyref | unique)*
  ```
- Possible parent element types: `schema`.

Elements (13)

`<element>` (local element type declaration):

- Possible attributes:
  - `name`: Element type name (`NCName`, required).
  - `form`: `"qualified"` or `"unqualified"` (see above).
  - `type`: Name of simple or complex type (`QName`).
  - `minOccurs`, `maxOccurs`: see above.
  - `default`, `fixed`, `nillable`: see above.
  - `block`: see below.
- Content model:

```
annotation?, (simpleType | complexType)?
             (key | keyref | unique)*
```

- Possible parent elements: `all`, `choice`, `sequence`.

Elements (14)

- The scope of a local element type declaration is the enclosing complex type definition.

  One can have two completely different local element type declarations inside different complex types.

- Within the same complex type, one can declare the same element type more than once, if the associated data type is identical.

  Only the types must be identical. Other properties (like default values) can be different. Anonymous types are never identical, even if they have the same content model and attributes.
  This double declaration might be necessary if the element type appears more than once in a content model and one wants a local declaration.

Elements (15)

Attribute `substitutionGroup`:

- It is possible to define a hierarchy on element types, again similar to subclasses.
- The name of the "superclass" (called the "head of the substitution group" in XML Schema) is defined in the attribute `substitutionGroup` (a `QName`).
- If the declaration of element type `E` contains

  ```
  substitutionGroup="S"
  ```

  then `E` is permitted everywhere where `S` is permitted, i.e. `E` can be substituted for `S`.

Elements (16)

- This is also possible over several levels (if `X` defines `E` as the head of its substitution group, `X` can be substituted for `E` and for `S`).
- Of course, the data types of these element types must be compatible, e.g. the data type of `E` must be derived from the data type of `S` (maybe indirectly) (it can also be the same).
- Alternatives to substitution groups are:
  - choice model group with all "subclass elements",
  - "superclass element" with type substitution.

Elements (17)

Attribute `abstract`:

- If this is `"true"`, the element type cannot be used in input documents (i.e. it cannot be instantiated).
- It can only be used as head of a substitution group (“superclass”).

  It appears of course in model groups of the schema, but only as placeholder for one of the element types that can be substituted for this element type. The element type substitution is required in this case.

- The default is `"false"`.

Elements (18)

Attribute `final`:

- With `final="#all"`, one can prevent that the current type can be used as head of a substitution group.

  The default is the value of the `finalDefault`-attribute of the `schema`-element, which defaults to `""`, i.e. no restriction.

- One can also specify restrictions on the data types of the element types that can be substituted for the current element type.

  E.g. `final="restriction"` means that the current element type can be head of a substitution group, but the data type of the substituted element type must be derived by restriction.

Elements (19)

Attribute `block`:

- The attribute `block` can be used to forbid type substitution or usage of substitution groups in the instance (input document, data file).

  As mentioned on Slide 22, one can use `xsi:type` in the input document (data file) to state that an element type E has not its normal data type C, but a data type C' that is derived from C.

  With the attribute `block`, certain forms of type derivation (`restriction` or `extension`) can be excluded from this possibility.

  `block="restriction extension"` completely excludes type substitution.

  The list can also contain `substitution`, which forbids element type substitution (via substitution groups). This is basically the same as `final="#all"`, but now only the concrete occurrence in the input document is false, not the schema.

Inhalt

Unique/Key Constraints (1)

`<unique>`/`<key>`:

- Possible attributes:
  - `name`: Name of the key constraint (`NCName`).

    This attribute is required. The value must be unique in the schema among all `unique`, `key`, and `keyref`-constraints.
- Content model:

  ```
  annotation?, selector, field+
  ```
- Possible parent element types: `element`.

Unique/Key Constraints (2)

`<selector>`:

- Possible attributes:
  - `xpath`: Defines the nodes that are to be identified by the key (restricted XPath expression).

    It is required. The XPath subset is explained below.

- Content model:

  `annotation?`

- Possible parent element types: `unique`, `key`, `keyref`.

Unique/Key Constraints (3)

`<field>`:

- Possible attributes:
  - `xpath`: Defines a component of the tuple of values that uniquely identifies the nodes.

    This attribute is required. The value must again be a restricted XPath expression, see below.
- Content model:

  `annotation?`
- Possible parent element types: `unique`, `key`, `keyref`.

XPath Subset (1)

- The standard states: "In order to reduce the burden on implementers, in particular implementers of streaming processors, only restricted subsets of XPath expressions are allowed in {selector} and {fields}."
- Indeed, the subset of XPath that can be used to define the components of keys is quite simple.
- The purpose of XPath is to select a set of nodes in the XML tree, given a context node as a starting point. In the XPath subset, one can navigate only downward in the tree (in full XPath, also upward).
- The XPath subset that can be used in `selector` and the subset that can be used in `field` differ slightly.

XPath Subset (2)

- A selector XPath expression consists of one or more "Paths", separated by "`|`":

```
Selector ::= Path ('|' Path)*
```

  The set of nodes that are selected by this expression is the union of the nodes selected by the single paths (as usual, "|" means disjunction).

- A Path
  - can optionally start with "`.//`".
  - After that, it is a sequence of steps, separated with "`/`":

```
Path ::= ('.//')? Step ('/' Step)*
```

- Let the start node set be:
  - If "`.//`" is present: The context node and all its descendants.
  - Otherwise: Only the context node.

XPath Subset (3)

- Each step defines a new set of nodes, given the resulting nodes from the previous step (initialized with the start node set).

  Formally, a step defines a set of selected nodes for a single given node. If the current node set consists of several nodes, take the union of the selected nodes given each element in the current node set.

- A step can be: `Step ::= '.' | NameTest`
  - "`.`": Selects the current node (nothing changed).
  - A "name test": This selects those children of the current node that are element nodes with an element type name satisfying the "name test" (see next slide).

XPath Subset (4)

- A "name test" is:
  - An element type name (a `QName`).

    Default namespace declarations do not affect XPath expressions. If the element type is in a namespace, one must use the prefix.
  - A wildcard "`*`" (satisfied by all element nodes).
  - A namespace with a wildcard (satisfied by all element nodes that belong to that namespace).

  ```
  NameTest ::= QName | '*' | NCName ':' '*'
  ```

  A name test can also be used for attribute nodes (see below).
- Between any two tokens, whitespace is allowed.
- That completes the definition of XPath expressions that can be used in the attribute `xpath` of `selector`.

XPath Subset (5)

- The XPath expressions in `field` permit in addition to select an attribute node as last step in `Path`:

  ```
  Path ::= ('.//')? (Step '/')* (Step | '@' NameTest)
  ```

  Although one can use "`|`" (disjunction) and wildcards, this is probably seldom applied because the XPath expression in `field` must select a single node. The node contents/value is taken implicitly at the end.

- A name test for attributes offers the same three possibilities as explained for element nodes above:
  - "`Name`": Attribute with that qualified name.
  - "`*`": Any attribute.
  - "`Prefix:*`": All attributes in that namespace.

Key References (1)

`<keyref>`:

- Possible attributes:
  - `name`: Name of the foreign key constraint (`NCName`).

    This attribute is required. The value must be unique in the schema among all `unique`, `key`, and `keyref`-constraints.

  - `refer`: Name of a `unique`/`key`-constraint (`NCName`).

    This attribute is required: By linking the foreign key to the referenced key, it defines which values are possible.

- Content model:

  `annotation?, selector, field+`

- Possible parent element types: `element`.

Inhalt

The Schema Element (1)

`<schema>`:

- Possible attributes:
  - `targetNamespace`: Namespace for defined schema components.

    At least the global types, elements, and attributes belong to this namespace. Whether local components belong to it depends on the `elementFormDefault`, `attributeFormDefault` and the `form` attribute of the element or attribute declaration.

  - `elementFormDefault`: `qualified` or `unqualified`.

    Default is `unqualified`. I.e. unless something else is specified in the local element declaration, it has no namespace.

  - `attributeFormDefault`: `qualified` or `unqualified`.

    Default is `unqualified`.

The Schema Element (2)

`<schema>`, continued:

- Possible attributes, continued:
  - `blockDefault`: Permission of type substitution (with `xsi:type`) and substitution groups.

    Possible values of this attribute are: `#all` or a list of `substitution`, `extension`, `restriction`. Default is the empty list, i.e. no restriction.
  - `finalDefault`: Permission of type derivation.

    Possible values are `#all` or a list of `extension`, `restriction`, `list`, `union`. Default is the empty list, i.e. no restriction.
  - `version`: Version of the schema (type `token`)
  - `xml:lang`: Language of the schema document.

The Schema Element (3)

`<schema>`, continued:

- Content model:

```
((include | import | redefine | annotation)*,
 ((simpleType | complexType |
   group | attributeGroup |
   element | attribute | notation),
  annotation*)*)
```

  I.e. if `include`, `import` and `redefine` are used, this must be done before defining the schema components of the current schema document. Note that if one would simply add `annotation` to the `choice` starting with `simpleType`, the content model would not be deterministic.

- Possible parent element types: None.

Including Schema Files

`<include>`:

- Possible attributes:
  - `schemaLocation`: The URI of the schema document to include.

    The included schema document cannot have a different target namespace than the including schema document. It is ok if it has no namespace.

- Content model:

  `annotation?`

- Possible parent element types: `schema`.

Importing Schema Files

`<import>`:

- Possible attributes:
  - `namespace`: The namespace of types, elements, etc. that will be used in this schema.

    This cannot be the same as the target namespace of the current schema document.
  - `schemaLocation`: URI of a schema document that defines the schema components of `namespace`.
- Content model: `annotation?`
- Possible parent element types: `schema`.

Include with Redefinition (1)

`<redefine>`:

- Possible attributes:
  - `schemaLocation`: The URI of the schema document to include.

    The included schema document must have the same target namespace as the current schema.
- Content model:

  ```
  (annotation | simpleType | complexType |
   attributeGroup | group)*
  ```
- Possible parent element types: `schema`.

Include with Redefinition (2)

- `redefine` automatically includes all schema components from the referenced schema, not only the redefined ones.
- Not arbitrary redefinitions are possible: Types or groups must restrict or extend the original version, it cannot be something entirely new.
- The new, redefined versions also apply to all elements (subtypes etc.) in the included schema.

Documentation, App. Info (1)

`<annotation>`:

- Possible attributes: (only `id`)
- Content model:

```
(documentation | appinfo)*
```

- Possible parent element types:

  `all`, `any`, `anyAttribute`, `attribute`, `attributeGroup`, `choice`, `complexContent`, `complexType`, `element`, `enumeration`, `extension`, `field`, `fractionDigits`, `group`, `import`, `include`, `key`, `keyref`, `length`, `list`, `maxExclusive`, `maxInclusive`, `maxLength`, `minExclusive`, `minInclusive`, `minLength`, `notation`, `pattern`, `redefine`, `restriction`, `schema`, `selector`, `sequence`, `simpleContent`, `simpleType`, `totalDigits`, `union`, `unique`, `whitespace`.

Documentation, App. Info (2)

`<documentation>`:

- Possible attributes:
  - `source`: URI pointing to further documentation
  - `xml:lang`: natural language of the documentation

    E.g. `de`, `en`, `en-US` (it has type `xs:language`).
- Content model: `ANY`

  In XML schema, this is the `any` wildcard, together with `mixed="true"`. It is processed using lax validation, i.e. one can specify a schema location with `xsi:schemaLocation` (e.g. in the root `xs:schema` element of the schema). Otherwise only the well-formedness is checked.
- Possible parent element types: `annotation`

Documentation, App. Info (3)

`<appinfo>`:

- Possible attributes:
  - `source`: URI pointing to further documentation
- Content model: `ANY`

  I.e. `any` wildcard with mixed content. Processed using lax validation. So `appinfo` has the same declaration as `documentation`, only without the `xml:lang` attribute.
- Possible parent element types: `annotation`

References

- Harald Schöning, Walter Waterfeld: XML Schema.
  In: Erhard Rahm, Gottfried Vossen: Web & Datenbanken, Seiten 33-64.
  dpunkt.verlag, 2003, ISBN 3-89864-189-9.
- Priscilla Walmsley: Definitive XML Schema.
  Prentice Hall, 2001, ISBN 0130655678, 560 pages.
- W3C Architecture Domain: XML Schema.
  [http://www.w3.org/XML/Schema]
- David C. Fallside, Priscilla Walmsley: XML Schema Part 0: Primer.
  W3C, 28. October 2004, Second Edition.
  [http://www.w3.org/TR/xmlschema-0/]
- Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelsohn:
  XML Schema Part 1: Structures.
  W3C, 28. October 2004, Second Edition
  [http://www.w3.org/TR/xmlschema-1/]
- Paul V. Biron, Ashok Malhotra: XML Schema Part 2: Datatypes.
  W3C, 28. October 2004, Second Edition
  [http://www.w3.org/TR/xmlschema-2/]
- [http://www.w3schools.com/schema/]