

# INFO-H-509 XML Technologies

## XML Schema Languages Part II

Stijn Vansummeren

March 15, 2012

# Objectives

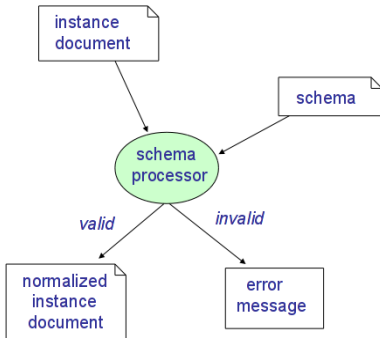
---

1. The essence of **XML Schema Definitions** (XSDs)
2. **Basic features** of XML Schema Definitions
3. **Advanced features** of XML Schema Definitions
4. **Deterministic** Regular Expressions

## Our story so far ...

- An **XML Language** is a set of XML documents that belong to the same “application domain”
- A **schema** is a formal definition of the syntax of an XML language
- A document is either **valid** w.r.t. a schema, or not
- A **schema language** is a notation by which schemas can be defined.

### The idea of validation:



## Our Story so far: Limitations of DTDs

---

1. Cannot constrain character data
2. Specification of attribute values is too limited
3. Element and attribute declarations are context insensitive
4. Character data cannot be combined with the regular expression content model
5. The content models lack an “interleaving” operator
6. The support for modularity, reuse, and evolution is too primitive
7. The normalization features lack content defaults and proper whitespace control
8. Structured embedded self-documentation is not possible
9. The ID/IDREF mechanism is too simple
10. It does not itself use an XML syntax
11. No support for namespaces

# Requirements for XML Schema

---

XML Schema is W3C's proposal for replacing DTDs

## Design Principles:

- More expressive than DTD
- Use XML notation
- Self-describing
- Simplicity

# Requirements for XML Schema

---

XML Schema is W3C's proposal for replacing DTDs

## Design Principles:

- More expressive than DTD
- Use XML notation
- Self-describing → **not really**
- Simplicity → **not really**

# Requirements for XML Schema

---


XML Schema is W3C's proposal for replacing DTDs

## Design Principles:

- More expressive than DTD
- Use XML notation
- Self-describing → **not really**
- Simplicity → **not really**

## Technical Requirements:

- Namespace support
- User-defined datatypes
- Inheritance (OO-like)
- Evolution
- Embedded documentation
- ...

A blackboard with a gold frame. The blackboard has some faint, light-colored chalk marks and scratches. The text "Part I: The Essence" is written in the center in white.

Part I: The Essence



# Let's start simple

---

- XML Schema is a large and complicated standard
- Its syntax in XML is really verbose ...
- ... so it's easy to get lost in the beginning

# Let's start simple

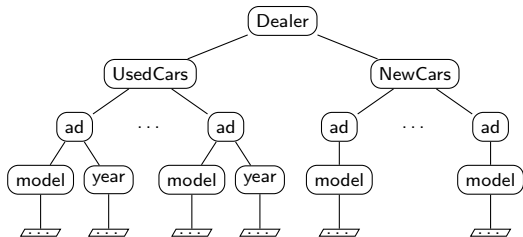
---

- XML Schema is a large and complicated standard
- Its syntax in XML is really verbose ...
- ... so it's easy to get lost in the beginning

- Let us illustrate the **essential ideas** without using the actual XSD syntax
- Let us focus on **elements**

# The Essence of XSDs

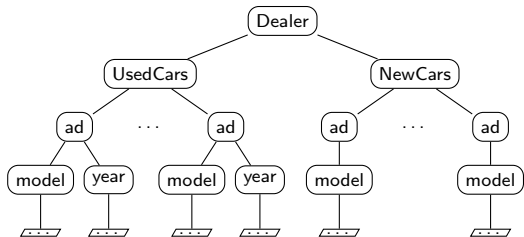
---



- ads under UsedCars must contain **both** model and year
- ads under NewCars must contain **only** model

# The Essence of XSDs

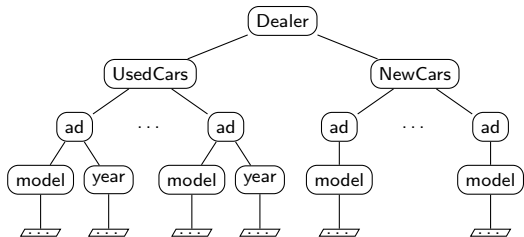
---



- ads under UsedCars must contain **both** model and year
- ads under NewCars must contain **only** model
- This XML language cannot be specified by a DTD

# The Essence of XSDs

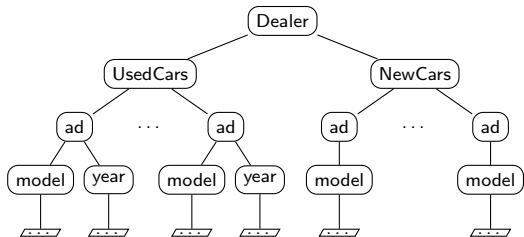
---



- ads under UsedCars must contain **both** model and year
- ads under NewCars must contain **only** model
- This XML language cannot be specified by a DTD

**We need a way to distinguish between UsedCar ads and NewCar ads**

# The Essence of XSDs



- ads under UsedCars must contain **both** model and year
- ads under NewCars must contain **only** model
- This XML language cannot be specified by a DTD

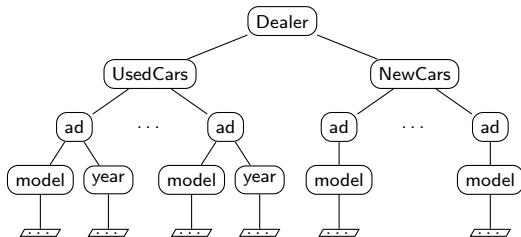
**We need a way to distinguish between UsedCar ads and NewCar ads**

**XML Schema solves this by introducing types**

- **Simple types** describe the legal values of text and attribute nodes (**integer**, **string**, **date**, ...)
- **Complex types** describe the content model for element nodes: each complex type is a regular expression over pairs of the form (*element name*, *type*).
- An XML Schema is a collection of **type definitions**.

## The Essence of XSDs (2)

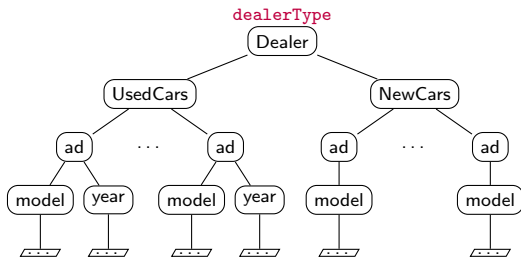
---



### Example:

`dealerType` → (`UsedCars`, `usedType`), (`NewCars`, `newType`)  
`usedType` → (`ad`, `adType1`)\*  
`newType` → (`ad`, `adType2`)\*  
`adType1` → (`model`, `string`), (`year`, `date`)  
`adType2` → (`model`, `string`)

## The Essence of XSDs (2)

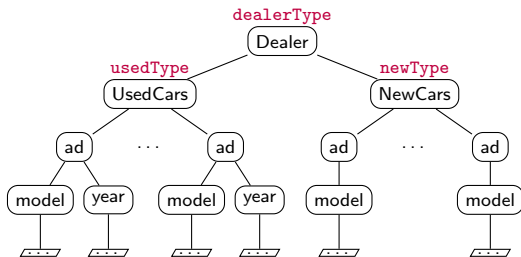


### Example:

`dealerType` → (`UsedCars`, `usedType`), (`NewCars`, `newType`)  
`usedType` → (`ad`, `adType1`)\*  
`newType` → (`ad`, `adType2`)\*  
`adType1` → (`model`, `string`), (`year`, `date`)  
`adType2` → (`model`, `string`)



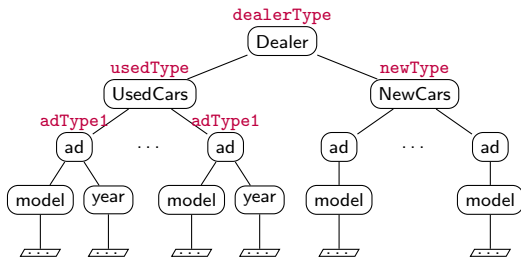
## The Essence of XSDs (2)



### Example:

`dealerType` → (`UsedCars`, `usedType`), (`NewCars`, `newType`)  
`usedType` → (`ad`, `adType1`)\*  
`newType` → (`ad`, `adType2`)\*  
`adType1` → (`model`, `string`), (`year`, `date`)  
`adType2` → (`model`, `string`)

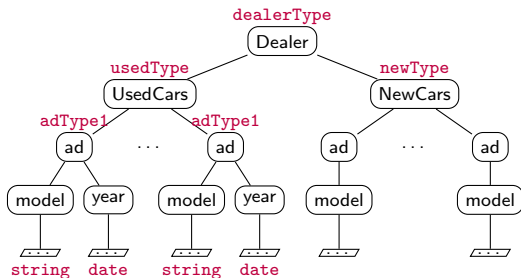
## The Essence of XSDs (2)



### Example:

`dealerType` → (`UsedCars`, `usedType`), (`NewCars`, `newType`)  
`usedType` → (`ad`, `adType1`)\*  
`newType` → (`ad`, `adType2`)\*  
`adType1` → (`model`, `string`), (`year`, `date`)  
`adType2` → (`model`, `string`)

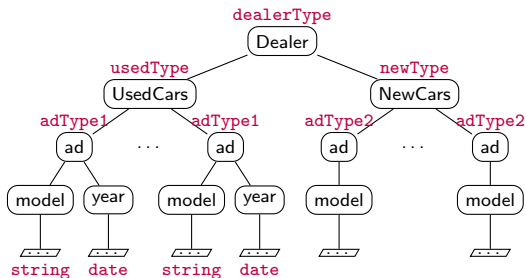
## The Essence of XSDs (2)



### Example:

`dealerType` → (`UsedCars`, `usedType`), (`NewCars`, `newType`)  
`usedType` → (`ad`, `adType1`)\*  
`newType` → (`ad`, `adType2`)\*  
`adType1` → (`model`, `string`), (`year`, `date`)  
`adType2` → (`model`, `string`)

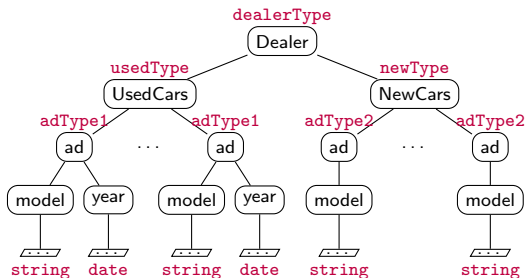
## The Essence of XSDs (2)



### Example:

`dealerType` → (`UsedCars`, `usedType`), (`NewCars`, `newType`)  
`usedType` → (`ad`, `adType1`)\*  
`newType` → (`ad`, `adType2`)\*  
`adType1` → (`model`, `string`), (`year`, `date`)  
`adType2` → (`model`, `string`)

## The Essence of XSDs (2)



### Example:

`dealerType` → (`UsedCars`, `usedType`), (`NewCars`, `newType`)  
`usedType` → (`ad`, `adType1`)\*  
`newType` → (`ad`, `adType2`)\*  
`adType1` → (`model`, `string`), (`year`, `date`)  
`adType2` → (`model`, `string`)

# The Element Declaration Consistent Constraint (EDC)

---

## Definition

XML Schema requires that **within the same type definition** the same element name must occur with the same type. This is called the **element declaration consistent** constraint (EDC).

## Example

# The Element Declaration Consistent Constraint (EDC)

## Definition

XML Schema requires that **within the same type definition** the same element name must occur with the same type. This is called the **element declaration consistent** constraint (EDC).

## Example

- Legal:

`dealerType` → (UsedCars, `usedType`), (NewCars, `newType`)

# The Element Declaration Consistent Constraint (EDC)

## Definition

XML Schema requires that **within the same type definition** the same element name must occur with the same type. This is called the **element declaration consistent** constraint (EDC).

## Example

- Legal:

`dealerType`  $\rightarrow$  (`UsedCars`, `usedType`), (`NewCars`, `newType`)

- Illegal:

`dealerType`  $\rightarrow$  (`Cars`, `usedType`), (`Cars`, `newType`)



# The Element Declaration Consistent Constraint (EDC)

## Definition

XML Schema requires that **within the same type definition** the same element name must occur with the same type. This is called the **element declaration consistent** constraint (EDC).

## Example

- Legal:

`dealerType` → (UsedCars, `usedType`), (NewCars, `newType`)

- Illegal:

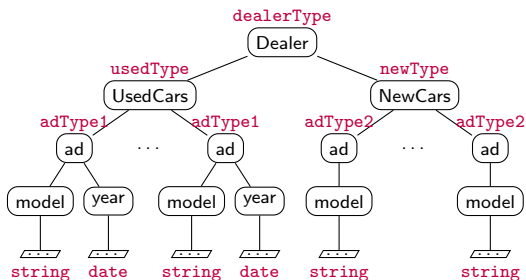
`dealerType` → (Cars, `usedType`), (Cars, `newType`)

- Legal:

`dealerType` → (Cars, `usedType`), (Cars, `usedType`)

# The Element Declaration Consistent Constraint (EDC) (2)

---

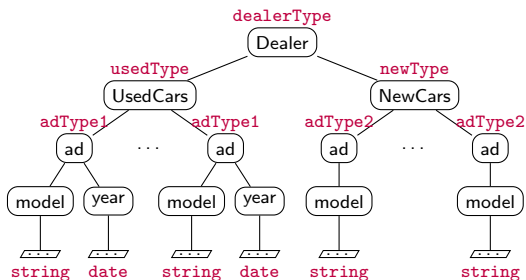


**Theorem** [Martens, Neven, Schwentick, 2006]

The type assignment is determined by path from element to root

# The Element Declaration Consistent Constraint (EDC) (2)

---



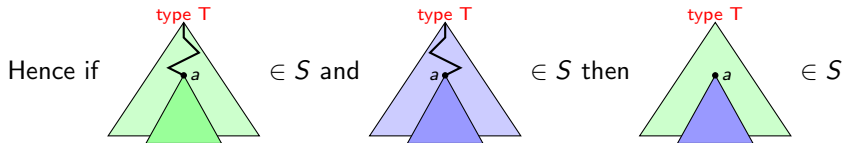
**Theorem** [Martens, Neven, Schwentick, 2006]

The type assignment is determined by path from element to root

In other words: paths determine types!

# What XML languages can we express with an XSD?

---



We can use this to show that an XML language is not definable in XML Schema

A dark blue chalkboard with a gold-colored frame. The text "Part II: Basic Syntax" is written in white in the center. There are faint, light-colored scribbles on the board, suggesting it was previously used for writing.

## Part II: Basic Syntax

# XSDs: XML Schema Definitions - Syntax

---

## Definition

Syntactically, an **XSDs** is a collection of:

- **complex type definitions**: defines content and attributes
- **simple type definitions**: defines a family of legal Unicode text strings
- **element declarations**: associate an element name with a simple or complex type
- **attribute declarations**: associate an attribute name with a simple type

## XSDs by Example

---

- XSDs are written in XML
- All definitions and declarations are put inside an **schema** element

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://cardealers.org"
        xmlns:b="http://cardealers.org">
```

*the definitions go here ...*

```
</schema>
```

# XSDs by Example

---

Example in our made-up syntax:

```
dealerType → (UsedCars, usedType), (NewCars, newType)
```

Real XSD syntax:

```
<complexType name="dealerType">  
  <sequence>  
    <element name="UsedCars" type="b:usedType"/>  
    <element name="NewCars" type="b:newType" />  
  </sequence>  
</complexType>
```



# XSDs by Example

---

## Example in our made-up syntax:

```
dealerType → (UsedCars, usedType), (NewCars, newType)
usedType  → (ad, adType1)*
```

## Real XSD syntax:

```
<complexType name="usedType">
  <sequence>
    <element name="ad" type="b:adType1"
      minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
</complexType>
```

# XSDs by Example

---

## Example in our made-up syntax:

```
dealerType → (UsedCars, usedType), (NewCars, newType)
usedType  → (ad, adType1)*
newType   → (ad, adType2)*
```

## Real XSD syntax:

```
<complexType name="newType">
  <sequence>
    <element name="ad" type="b:adType2"
      minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
</complexType>
```

# XSDs by Example

## Example in our made-up syntax:

```
dealerType → (UsedCars, usedType), (NewCars, newType)
usedType  → (ad, adType1)*
newType   → (ad, adType2)*
adType1   → (model, string), (year, date)
```

## Real XSD syntax:

```
<complexType name="addType1">
  <sequence>
    <element name="model" type="string" />
    <element name="year" type="date" />
  </sequence>
</complexType>
```

# XSDs by Example

---

## Example in our made-up syntax:

```
dealerType → (UsedCars, usedType), (NewCars, newType)
usedType  → (ad, adType1)*
newType   → (ad, adType2)*
adType1   → (model, string), (year, date)
adType2   → (model, string)
```

## Real XSD syntax:

```
<complexType name="addType2">
  <sequence>
    <element name="model" type="string" />
  </sequence>
</complexType>
```

# XSDs by Example

---

## Example in our made-up syntax:

```
dealerType → (UsedCars, usedType), (NewCars, newType)
usedType  → (ad, adType1)*
newType   → (ad, adType2)*
adType1   → (model, string), (year, date)
adType2   → (model, string)
```

Finally we need to declare globally that Dealer elements have type `dealerType`:

```
<!-- Global element declaration -->
<element name="Dealer" type="b:dealerType"/>
```

# The complete XSD

---



By means of  
**Online**  
demonstration

# Element and attribute declarations

---

## Definition

### Declarations (global and local):

- `<element name="elem name" type="simple or complex type name"/>`
- `<attribute name="attr name" type="simple type name ..."/>`

### References:

- `<element ref="elem name"/>`
- `<attribute ref="attr name"/>`

# Element and attribute declarations

---

## Definition

### Declarations (global and local):

- `<element name="elem name" type="simple or complex type name"/>`
- `<attribute name="attr name" type="simple type name ..."/>`

### References:

- `<element ref="elem name"/>`
- `<attribute ref="attr name"/>`



**Example**  
**By means of**  
**Online**  
**demonstration**



# Global versus local type definitions

---

## Global style:

```
<complexType name="newType">
  <sequence>
    <element name="ad" type="b:adType2"
      minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
</complexType>

<complexType name="addType2">
  <sequence>
    <element name="model" type="string" />
  </sequence>
</complexType>
```

# Global versus local type definitions

---

## Local style: in-line anonymous type definitions

```
<complexType name="newType">
  <sequence>
    <element name="ad" minOccurs="0" maxOccurs="unbounded">
      <complexType>
        <sequence>
          <element name="model" type="string" />
        </sequence>
      </complexType>
    </element>
  </sequence>
</complexType>
```

# Global versus local declarations

## An example with global and local declarations:

```
<element name="ad" type="adType1">

<complexType name="usedType">
  <sequence>
    <element ref="b:ad" minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
</complexType>

<complexType name="newType">
  <sequence>
    <element name="ad" type="b:adType2"
      minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
</complexType>
```

Note that we need at least one local element declaration to distinguish between used car ads and new car ads. This is called **overloading**.

# Complex Type Definitions

---

## Definition

- Syntax:

```
<complexType name="..."> content model/attributes </complexType>
```

# Complex Type Definitions

---

## Definition

- Syntax:

```
<complexType name="..."> content model/attributes </complexType>
```

- Content models are **regular expressions** with a peculiar syntax

# Complex Type Definitions

---

## Definition

- Syntax:

```
<complexType name="..."> content model/attributes </complexType>
```

- Content models are **regular expressions** with a peculiar syntax

Element declaration → `<element name="..." type="...">`

Element reference → `<element ref="...">`

Concatenation → `<sequence> ...</sequence>`

Union → `<choice> ...</choice>`

All (unordered) → `<all> ...</all>`

Element Wildcard → `<any namespace="..." processContents="...">`

Cardinalities (\*,+) → `attributes minOccurs, maxOccurs`

Mixed content → `attribute mixed="true"`

# Complex Type Definitions

## Definition

- Syntax:

```
<complexType name="..."> content model/attributes </complexType>
```

- Content models are **regular expressions** with a peculiar syntax

Element declaration → `<element name="..." type="...">`

Element reference → `<element ref="...">`

Concatenation → `<sequence> ...</sequence>`

Union → `<choice> ...</choice>`

All (unordered) → `<all> ...</all>`

Element Wildcard → `<any namespace="..." processContents="...">`

Cardinalities (\*,+) → `attributes minOccurs, maxOccurs`

Mixed content → `attribute mixed="true"`

- Attributes:

Attribute declaration → `<attribute name="..." type="..." ...>`

Attribute reference → `<attribute ref="..." ...>`

Attribute wildcard → `<attribute namespace="..."  
processContents="...">`

## Complex Type Definitions (2)

---

### Example:

```
<element name="order" type="n:order_type"/>

<complexType name="order_type" mixed="true">
  <choice>
    <element ref="n:address"/>
    <sequence>
      <element ref="n:email" minOccurs="0" maxOccurs="4"/>
      <element ref="n:phone"/>
    </sequence>
  </choice>

  <attribute ref="n:id" use="required"/>
  <attribute ref="n:email" default="no email address available"/>
  <attribute ref="n:method" fixed="some fixed value"/>
</complexType>
```



## Simple types

- XML Schema has a myriad of built-in simple types ...

<code>string</code>	any Unicode string
<code>boolean</code>	true, false, 1, 0
<code>decimal</code>	3.1415
<code>float</code>	6.02214199E23
<code>double</code>	42E970
<code>dateTime</code>	2004-09-26T16:29:00-05:00
<code>time</code>	16:29:00-05:00
<code>date</code>	2004-09-26
<code>hexBinary</code>	48656c6c6f0a
<code>base64Binary</code>	SGVsbG8K
<code>anyURI</code>	<a href="http://www.brics.dk/ixwt/">http://www.brics.dk/ixwt/</a>
<code>QName</code>	rcp:recipe, recipe
...	

# Simple types

- XML Schema has a myriad of built-in simple types ...
- ... but it is **also possible to define your own simple types.**

<code>string</code>	any Unicode string
<code>boolean</code>	true, false, 1, 0
<code>decimal</code>	3.1415
<code>float</code>	6.02214199E23
<code>double</code>	42E970
<code>dateTime</code>	2004-09-26T16:29:00-05:00
<code>time</code>	16:29:00-05:00
<code>date</code>	2004-09-26
<code>hexBinary</code>	48656c6c6f0a
<code>base64Binary</code>	SGVsbG8K
<code>anyURI</code>	http://www.brics.dk/ixwt/
<code>QName</code>	rcp:recipe, recipe
...	

## Simple type definition by example

---

- New simple types are defined by **restricting existing simple types**

### Example:

- To match integers between 0 and 100:

```
<simpleType name="score_from_0_to_100">  
  <restriction base="integer">  
    <minInclusive value="0"/>  
    <maxInclusive value="100"/>  
  </restriction>  
</simpleType>
```

## Simple type definition by example

---

- New simple types are defined by **restricting existing simple types**

### Example:

- To match strings of the form  $N\%$  with  $0 \leq N \leq 100$ :

```
<simpleType name="percentage">
  <restriction base="string">
    <pattern value="([0-9] | [1-9] [0-9] | 100)%"/>
  </restriction>
</simpleType>
```

## Simple type definition by example

---

- New simple types are defined by **restricting existing simple types**
- By defining **lists of simple types**

### Example:

- To match a whitespace-separated list of integers like 1 55 399:

```
<simpleType name="score_from_0_to_100">  
  <list itemType="integer"/>  
</simpleType>
```

# Simple type definition by example

---

- New simple types are defined by **restricting existing simple types**
- By defining **lists of simple types**
- Or taking **unions of simple types**

## Example:

- To match all booleans and decimals:

```
<simpleType name="boolean_or_decimal">
  <union>
    <simpleType>
      <restriction base="boolean"/>
    </simpleType>
    <simpleType>
      <restriction base="decimal"/>
    </simpleType>
  </union>
</simpleType>
```

# Deriving simple types by restriction

---

The things that we restrict by are called **facets**

## Available constraining facets:

- length
- minLength
- maxLength
- pattern
- enumeration
- whiteSpace
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive
- totalDigits
- fractionDigits

## Built-in derived simple types

---

XML Schema has a myriad of built-in simple types that are defined by derivation from the primitive simple types

- normalizedString
- token
- language
- Name
- NCName
- ID
- IDREF
- integer
- nonNegativeInteger
- unsignedLong
- long
- int
- short
- byte
- ...



## Complex types with simple content

---

- Sometimes we want to specify that the content of an element should be of some simple type, but that it can also have some attribute.
- This requires a peculiar syntax

### Example:

```
<element name="category" type="n:category"/>
<attribute name="class" type="string"/>


<complexType name="category">
  <simpleContent>
    <extension base="integer">
      <attribute ref="n:class"/>
    </extension>
  </simpleContent>
</complexType>
```

# Connecting Schemas and Instances

## Example instance document:

```
<b:card xmlns:b="http://businesscard.org"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://businesscard.org
                            business_card.xsd">
  <b:name>John Doe</b:name>
  <b:title>CEO, Widget Inc.</b:title>
  <b:email>john.doe@widget.com</b:email>
  <b:phone>(202) 555-1414</b:phone>
  <b:logo b:uri="widget.gif"/>
</b:card>
```

- **Only globally declared elements can be starting points for validation!**
- **The targetNamespace of the Schema, and the namespace of the elements in the instance document must match!**
- **If the XSD does not have a target namespace, use noNamespaceSchemaLocation instead of schemaLocation**

A dark blue chalkboard with a gold-colored frame. The text "Part III: Advanced Features" is written in white, centered on the board. There are some faint, light-colored scribbles on the chalkboard surface.

## Part III: Advanced Features

# Complex Type Derivation

- Also complex types can be derived by **restricting or extending existing complex types**
- This is similar to inheritance in object-oriented programming languages

## Example:

- Assume given the following complex type:

```
<complexType name="basic_card_type">
  <sequence> <element name="name" type="string"/> </sequence>
</complexType>
```

# Complex Type Derivation

- Also complex types can be derived by **restricting or extending existing complex types**
- This is similar to inheritance in object-oriented programming languages

## Example:

- We can extend this type with a title element and optional email elements as follows:

```
<complexType name="extended_type">
  <complexContent>
    <extension base="b:basic_card_type">
      <sequence>
        <element ref="b:title"/>
        <element ref="b:email" minOccurs="0"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

# Complex Type Derivation

- Also complex types can be derived by **restricting or extending existing complex types**
- This is similar to inheritance in object-oriented programming languages

## Example:

- We can subsequently restrict this type such that email becomes required:

```
<complexType name="restricted_type">
  <complexContent>
    <restriction base="b:extended_type">
      <sequence>
        <element name="name" type="string"/>
        <element ref="b:title"/>
        <element ref="b:email"/>
      </sequence>
    </restriction>
  </complexContent>
</complexType>
```

# Complex Type Derivation: subsumption

---

## Assume that:

- $T$  is some (complex or simple) type
- $T^-$  is derived from  $T$  by **restriction**
- $T^+$  is derived from  $T$  by **extension**

## Definition

**Subsumption** is the principle that whenever an instance of type  $T$  is required,

- an instance of type  $T^-$  may be used instead (every instance of type  $T^-$  is also of type  $T$ )
- an instance of type  $T^+$  may be used instead if the instance has attribute `xsi:type="T+"` with

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

# Complex Type Derivation: subsumption

---

## Assume that:

- $T$  is some (complex or simple) type
- $T^-$  is derived from  $T$  by **restriction**
- $T^+$  is derived from  $T$  by **extension**

## Definition

**Subsumption** is the principle that whenever an instance of type  $T$  is required,

- an instance of type  $T^-$  may be used instead (every instance of type  $T^-$  is also of type  $T$ )
- an instance of type  $T^+$  may be used instead if the instance has attribute `xsi:type="T+"` with

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

**Note:** Derivation, instantiation, and subsumption can be constrained using `final`, `abstract`, and `block`



# Namespaces

---

`<schema targetNamespace="..." ...>`

- Prefixes are also used in certain attribute values!
- Unqualified Locals:
  - if enabled, the name of a locally declared element or attribute in the instance document must have no namespace prefix (i.e. the empty namespace URI)
  - such an attribute or element “belongs to” the element declared in the surrounding global definition
  - always change the default behavior using `elementFormDefault="qualified"`

# Uniqueness, Keys, references

- Keys can be defined by means of key
- Keys can be referred to by means of keyref
- unique functions as key, but fields may be absent

## Example:

```
<element name="w:widget" xmlns:w="http://www.widget.org">
  <complexType> ... </complexType>

  <key name="my_widget_key">
    <selector xpath="w:components/w:part"/>
    <field xpath="@manufacturer"/>
    <field xpath="w:info/@productid"/>
  </key>

  <keyref name="annotation_references" refer="w:my_widget_key">
    <selector xpath="./w:annotation"/>
    <field xpath="@manu"/>
    <field xpath="@prod"/>
  </keyref>
</element>
```

# Other Features

---

- Groups
- Substitution groups (essentially subsumption based on element names, not types)
- Nil values
- Annotations
- Defaults and whitespace
- Modularization

Read the book chapter

# Specifying RecipyML with XML Schema

---



By means of  
**Online**  
demonstration

# Problems with the XML Schema Description

---

## We had the following problems with the DTD description:

- `calories` should contain a non-negative number; → **FIXED!**
- `protein` should contain a value on the form  $N\%$  where  $N$  is between 0 and 100; → **FIXED!**
- `comment` should be allowed to appear anywhere in the contents of `recipe`; → **NOT FIXED!**
- `unit` should only be allowed in an elements where `amount` is also present; → **NOT FIXED!**
- nested `ingredient` elements should only be allowed when `amount` is absent; → **NOT FIXED!**

# Limitations of XML Schema

---

- The details are extremely complicated (and the spec is unreadable)
- Declarations are (mostly) context insensitive
- It is impossible to write an XML Schema description of XML Schema
- With mixed content, character data cannot be constrained
- Unqualified local elements are bad practice
- Cannot require specific root element
- Element defaults cannot contain markup
- The type system is overly complicated
- `xsi:type` is problematic
- Simple type definitions are inflexible

## An important comment (not in book!)

---

The regular expressions used as element content models must be **deterministic**, sometimes also called **one-unambiguous**

### Intuitively:

- Intuitively, a regular expression is deterministic if, when processing the input sequence from left to right, it is always determined which symbol in the expression matches the next input symbol **without looking ahead**.
- This is supposed to make implementations faster

## An important comment (not in book!)

---

The regular expressions used as element content models must be **deterministic**, sometimes also called **one-unambiguous**

### Intuitively:

- Intuitively, a regular expression is deterministic if, when processing the input sequence from left to right, it is always determined which symbol in the expression matches the next input symbol **without looking ahead**.
- This is supposed to make implementations faster

### Example:

- $a, a \mid a, b$  is **not** deterministic (consider input  $aa$ )



## An important comment (not in book!)

---

The regular expressions used as element content models must be **deterministic**, sometimes also called **one-unambiguous**

### Intuitively:

- Intuitively, a regular expression is deterministic if, when processing the input sequence from left to right, it is always determined which symbol in the expression matches the next input symbol **without looking ahead**.
- This is supposed to make implementations faster

### Example:

- $a, a \mid a, b$  is **not** deterministic (consider input  $aa$ )
- $a, (a \mid b)$  **is** deterministic

## An important comment (not in book!)

---

The regular expressions used as element content models must be **deterministic**, sometimes also called **one-unambiguous**

### Intuitively:

- Intuitively, a regular expression is deterministic if, when processing the input sequence from left to right, it is always determined which symbol in the expression matches the next input symbol **without looking ahead**.
- This is supposed to make implementations faster

### Example:

- $a, a \mid a, b$  is **not** deterministic (consider input  $aa$ )
- $a, (a \mid b)$  **is** deterministic
- Note that these two expressions match the same sequences!

## An important comment (not in book!)

---

### Definition

- For a regular expression  $\alpha$ , define  $\bar{\alpha}$  to be the regular expression obtained from  $\alpha$  by replacing, for each  $i$ , the  $i$ -th occurrence of symbol  $\sigma$  in  $\alpha$  (counting from left to right) by  $\sigma_i$

### Example:

- $a, a \mid a, b \rightarrow a_1, a_2 \mid a_3, b_1$
- $a, (a \mid b) \rightarrow a_1, (a_2 \mid b_1)$
- $(a \mid b)^*, a, c, (b \mid c)^* \rightarrow (a_1 \mid b_1)^*, a_2, c_1, (b_2 \mid c_2)^*$

## An important comment (not in book!)

---

### Definition

- For a regular expression  $\alpha$ , define  $\bar{\alpha}$  to be the regular expression obtained from  $\alpha$  by replacing, for each  $i$ , the  $i$ -th occurrence of symbol  $\sigma$  in  $\alpha$  (counting from left to right) by  $\sigma_i$
- So if  $\alpha$  is over the alphabet  $\{a, b, c, \dots\}$  then  $\bar{\alpha}$  is over the alphabet  $\{a_1, a_2, \dots, b_1, b_2, \dots, c_1, c_2, \dots\}$ .

### Example:

- $a, a \mid a, b \rightarrow a_1, a_2 \mid a_3, b_1$
- $a, (a \mid b) \rightarrow a_1, (a_2 \mid b_1)$
- $(a \mid b)^*, a, c, (b \mid c)^* \rightarrow (a_1 \mid b_1)^*, a_2, c_1, (b_2 \mid c_2)^*$

## An important comment (not in book!)

---

### Definition

- For a regular expression  $\alpha$ , define  $\bar{\alpha}$  to be the regular expression obtained from  $\alpha$  by replacing, for each  $i$ , the  $i$ -th occurrence of symbol  $\sigma$  in  $\alpha$  (counting from left to right) by  $\sigma_i$
- So if  $\alpha$  is over the alphabet  $\{a, b, c, \dots\}$  then  $\bar{\alpha}$  is over the alphabet  $\{a_1, a_2, \dots, b_1, b_2, \dots, c_1, c_2, \dots\}$ .
- A regular expression  $\alpha$  is **deterministic** if there are no sequences  $wa_jv$  and  $wa_jv'$  in  $\mathcal{L}(\bar{\alpha})$  with  $i \neq j$  ( $w$  may be empty).

### Example:

- $a, a \mid a, b \rightarrow a_1, a_2 \mid a_3, b_1$
- $a, (a \mid b) \rightarrow a_1, (a_2 \mid b_1)$
- $(a \mid b)^*, a, c, (b \mid c)^* \rightarrow (a_1 \mid b_1)^*, a_2, c_1, (b_2 \mid c_2)^*$

## An important comment (not in book!)

---

### Definition

- For a regular expression  $\alpha$ , define  $\bar{\alpha}$  to be the regular expression obtained from  $\alpha$  by replacing, for each  $i$ , the  $i$ -th occurrence of symbol  $\sigma$  in  $\alpha$  (counting from left to right) by  $\sigma_i$
- So if  $\alpha$  is over the alphabet  $\{a, b, c, \dots\}$  then  $\bar{\alpha}$  is over the alphabet  $\{a_1, a_2, \dots, b_1, b_2, \dots, c_1, c_2, \dots\}$ .
- A regular expression  $\alpha$  is **deterministic** if there are no sequences  $wa_jv$  and  $wa_jv'$  in  $\mathcal{L}(\bar{\alpha})$  with  $i \neq j$  ( $w$  may be empty).

### Example:

- $a, a \mid a, b \rightarrow a_1, a_2 \mid a_3, b_1$  **not deterministic: consider  $a_1a_2$  and  $a_3b_1$**
- $a, (a \mid b) \rightarrow a_1, (a_2 \mid b_1)$
- $(a \mid b)^*, a, c, (b \mid c)^* \rightarrow (a_1 \mid b_1)^*, a_2, c_1, (b_2 \mid c_2)^*$

## An important comment (not in book!)

---

### Definition

- For a regular expression  $\alpha$ , define  $\bar{\alpha}$  to be the regular expression obtained from  $\alpha$  by replacing, for each  $i$ , the  $i$ -th occurrence of symbol  $\sigma$  in  $\alpha$  (counting from left to right) by  $\sigma_i$
- So if  $\alpha$  is over the alphabet  $\{a, b, c, \dots\}$  then  $\bar{\alpha}$  is over the alphabet  $\{a_1, a_2, \dots, b_1, b_2, \dots, c_1, c_2, \dots\}$ .
- A regular expression  $\alpha$  is **deterministic** if there are no sequences  $wa_jv$  and  $wa_jv'$  in  $\mathcal{L}(\bar{\alpha})$  with  $i \neq j$  ( $w$  may be empty).

### Example:

- $a, a \mid a, b \rightarrow a_1, a_2 \mid a_3, b_1$
- $a, (a \mid b) \rightarrow a_1, (a_2 \mid b_1)$  **deterministic**
- $(a \mid b)^*, a, c, (b \mid c)^* \rightarrow (a_1 \mid b_1)^*, a_2, c_1, (b_2 \mid c_2)^*$

## An important comment (not in book!)

---

### Definition

- For a regular expression  $\alpha$ , define  $\bar{\alpha}$  to be the regular expression obtained from  $\alpha$  by replacing, for each  $i$ , the  $i$ -th occurrence of symbol  $\sigma$  in  $\alpha$  (counting from left to right) by  $\sigma_i$
- So if  $\alpha$  is over the alphabet  $\{a, b, c, \dots\}$  then  $\bar{\alpha}$  is over the alphabet  $\{a_1, a_2, \dots, b_1, b_2, \dots, c_1, c_2, \dots\}$ .
- A regular expression  $\alpha$  is **deterministic** if there are no sequences  $wa_jv$  and  $wa_jv'$  in  $\mathcal{L}(\bar{\alpha})$  with  $i \neq j$  ( $w$  may be empty).

### Example:

- $a, a \mid a, b \rightarrow a_1, a_2 \mid a_3, b_1$
- $a, (a \mid b) \rightarrow a_1, (a_2 \mid b_1)$
- $(a \mid b)^*, a, c, (b \mid c)^* \rightarrow (a_1 \mid b_1)^*, a_2, c_1, (b_2 \mid c_2)^*$   
**deterministic? or not?**



# An important comment (not in book!)

## Definition

- For a regular expression  $\alpha$ , define  $\bar{\alpha}$  to be the regular expression obtained from  $\alpha$  by replacing, for each  $i$ , the  $i$ -th occurrence of symbol  $\sigma$  in  $\alpha$  (counting from left to right) by  $\sigma_i$
- So if  $\alpha$  is over the alphabet  $\{a, b, c, \dots\}$  then  $\bar{\alpha}$  is over the alphabet  $\{a_1, a_2, \dots, b_1, b_2, \dots, c_1, c_2, \dots\}$ .
- A regular expression  $\alpha$  is **deterministic** if there are no sequences  $wa_jv$  and  $wa_jv'$  in  $\mathcal{L}(\bar{\alpha})$  with  $i \neq j$  ( $w$  may be empty).

## Example:

- $a, a \mid a, b \rightarrow a_1, a_2 \mid a_3, b_1$
- $a, (a \mid b) \rightarrow a_1, (a_2 \mid b_1)$
- $(a \mid b)^*, a, c, (b \mid c)^* \rightarrow (a_1 \mid b_1)^*, a_2, c_1, (b_2 \mid c_2)^*$

**not**

# An important comment (not in book!)

---

Can we make every regular expression deterministic?

- $a, a \mid a, b \longrightarrow a, (a \mid b)$

## An important comment (not in book!)

---

Can we make every regular expression deterministic?

- $a, a \mid a, b \longrightarrow a, (a \mid b)$

### Theorem

**No** there exists regular expressions for which no equivalent deterministic regular expression exists.

**So we are limited to a subset of the regular expressions in DTDs and XSDs**