

# jInfer BasicXSDExporter Module Description

Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, Matej Vitásek  
Advisors: RNDr. Irena Mlýnková, Ph.D., Martin Nečaský, Ph.D.

Praha, 2011

Target audience: developers willing to extend jInfer, specifically hack the XSD export.

Responsible developer:	Mário Mikula
Required tokens:	none
Provided tokens:	cz.cuni.mff.ksi.jinfer.base.interfaces.inference.SchemaGenerator
Module dependencies:	Base
Public packages:	none

## 1 Introduction

This is an implementation of a *SchemaGenerator* exporting the inferred schema to XSD, providing basic features of the language.

## 2 Structure

The main class implementing *SchemaGenerator* inference interface and simultaneously registered as its service provider is *SchemaGeneratorImpl*. Process of export consists of two phases described in detail in later sections:

1. Preprocessing
2. Own export to a string representation of XSD

Method `start` first creates instance of *Preprocessor* class supplied by rules (elements) it got in the simplified grammar on input. Phase of preprocessing is done by creating that instance (calling its constructor) and its purpose is to discover information such which elements should be globally defined and which element is the root element. This instance is kept as a member variable of module to be used during the whole export process.

Afterwards, `start` method recursively traverses global elements followed by other elements starting at the root element and for each it creates element's XSD string representation.

### 2.1 Preprocessing

As mentioned before, preprocessing is implemented in *Preprocessor* class and its functions are following.

- Decide which elements should be defined globally.
- Remove unused elements.
- Find the top level element.
- Find an instance of element by its name.

Constructor of *Preprocessor* class gets elements and a number, defining minimal number of occurrences of an element to be defined. It first topologically sorts input elements to decide which one is the root element. Afterwards, it counts occurrences of the elements and removes unused ones (those which did not occurred). Finally, for each element it decides whether mark it as a global one or not. An element is considered global if its occurrence count is

greater than or equal number of occurrences supplied on input.

Results of preprocessing are provided by public methods of Preprocessor class. For details see their JavaDoc.

## 2.2 Own export

Own XSD export is performed in module's start function right after the preprocessing.

Useful helper class to handle indentation of text in a resulting XSD is named Indentator. Instance of this class is a member variable of the module (alike instance of Preprocessor), it holds text appended to it and keeps indentation level state. Text can be appended without indentation (method append) or indented (method indent). Level of indentation can be incremented or decremented by methods increaseIndentation and decreaseIndentation. At the end of export, when textual representation of each element has been appended to the Indentator, Indentator's method toString will return string representation of resulting XSD.

Before describing export of elements, let's take a look on how we define elements and their attributes using XSD language.

### 2.2.1 Definition of elements

Element is defined by XSD element element, specifying its name and type. Let xs be XMLSchema namespace.

```
<xs:element name="Person" type="..."
```

Type of an element is one of following.

- XSD built-in type. One of types like xs:string, xs:integer, xs:positiveInteger, etc.

```
<xs:element name="Person" type="xs:string"/>
```

- simpleType. Actually, exporter does not support any XSD features, which are defined in simpleType. That means, this types will not occur in result XSDs.

```
<xs:element name="Person">
  <xs:simpleType>
    ...
  </xs:simpleType>
</xs:element>
```

- complexType. This type can contain XSD element xs:sequence or xs:choice. Each of these elements can contain definitions of elements, xs:sequences and xs:choices again.

```
<xs:element name="Person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Name" type="xs:string"/>
      <xs:element name="Surname" type="xs:string"/>
    <xs:choice>
      ...
    </xs:choice>
  </xs:sequence>
</xs:complexType>
</xs:element>
```

An empty element is defined as empty complexType

```

<xs:element name="EmptyElement">
  <xs:complexType>
  </xs:complexType>
</xs:element>

```

Named type of an element is defined by XSD element `simpleType` (lack of features mentioned above) or `complexType` with specified attribute name. Its content is exactly the same as described above. There is, of course, no need to define built-in types.

```

<xs:complexType name="PersonType">
...
</xs:complexType>

```

Element of this type can be then defined by specifying name of the type.

```

<xs:element name="Person" type="PersonType"/>

```

XSD elements `xs:element`, `xs:sequence` and `xs:choice` can have attributes `minOccurs` and `maxOccurs`. These attributes define interval of number of instances of a particular element. Legal values of these attributes are non-negative integers.

```

<xs:element name="Person" type="PersonType" minOccurs="1" maxOccurs="3"/>

```

Default values for `minOccurs` and `maxOccurs` attributes are "1", if they are not specified. So the example above has the same meaning as following.

```

<xs:element name="Person" type="PersonType" maxOccurs="3"/>

```

First, global elements are exported. For each element, its type is defined as a global type. TODO rio example. This is done by passing global elements to `processGlobalElement` method.

After global elements, others are exported by calling of method `processElement`, supplied by the top level element as its argument. Reference to a global element type is done simply by declaring element's name and type.

```

<xs:element name="Person" type="PersonType"/>

```

To sa deje rekurzivnym priechodom so startom v korenovom elemente. Elementy, ktorých typy su definovane globalne a elementy so vstavanyimi typmi, su exportovane jednoducho, uvedením ich mena a typu. Ostatne elementy su definovane na mieste. TODO rio priklady. Na toto je potrebne zavolať metódu `processElement` s korenovým elementom, ako argumentom.

### 2.2.2 Definition of elements

Spracovava sa vo funkcii `processElementContent`. Obsah elementu je definovany pomocou elementu `element` a bud pomocou jedného zo vstavanych typov ako napríklad `string`, `integer`, `positiveInteger`, etc alebo je definovany vlastny.

Vlastny obsah elementu je definovany bud ako `simpleType` alebo `complexType`. TODO rio priklady. Actually exporter nepodporuje konstrukcie XSD, ktore sa definuju cez `simpleType`, takže tento typ sa vo výsledku neobjavi. `complexType` moze obsahovat konstrukcie `sequence` a `choice`, v ktorých sa mozu vyskytovat dalsie elementy alebo zas tieto konstrukcie. `complexType` moze na konci obsahovat definicie atributov (popisane nizšie). Prazdny element je vyjadreny prazdnym `complexType`.

Konstrukcie `element`, `sequence` a `choice` moze obsahovat atributy `minOccurs` a `maxOccurs`, ktore udavaju interval poctu instancií danej konstrukcie v mieste definovania. TODO priklad. Moze byt zadane aj jednotlivy. Pri nezadani plati podľa pravidiel XSD default hodnota 1. Tieto hodnoty sa ziskavaju z metadat pomocou funkcie `getInterval` definovanej na triede `regexp` (TODO odkaz niekam).

Exporter podporuje zmiesany obsah elementov definovany atributom `mixed="true"`. TODO priklad.

## 2.3 Attributes

Atribut elementu sa definuju pomocou elementu attribute and takes place at the end of complexType element. Spracovavaju sa vo funkcii `processElementAttributes`.

- Built-in types like `string`, `integer`, `positiveInteger`, etc.
- Attribute requiredness defined by attribute `use="required"`.

Code exporting attributes is in `attributeToString()`. First thing this method does is to assess the domain of a particular attribute: this is a map indexed by attribute values containing number of occurrences for each such attribute. Type definition of an attribute is generated in the `DomainUtils.getAttributeType()` method. Based on a user setting, this might decide to enumerate all possible values of this attribute using the `(a|b|c)` notation, otherwise it just returns `#CDATA`.

Attribute requiredness is assessed based on required metadata presence. If an attribute is not deemed required, it might have a default value: if a certain value is prominent in the attribute domain (based on user setting again), it is declared default.

## 2.4 Preferences

All settings provided by *BasicDTDExporter* are project-wide, the preferences panel is in `cz.cuni.mff.ksi.jinfer.basicdtd.properties` package. As mentioned before, it is possible to set the following.

- Maximum attribute domain size which is exported as a list of all values `((a|b|c)` notation).
- Minimal ratio an attribute value in the domain needs to have in order to be declared default.

## 3 Data flow

Flow of data in this module is following.

1. `SchemaGeneratorImpl` topologically sorts elements (rules) it got on input.
2. For each element, relevant portion of DTD schema is generated.
3. String representation of the schema is returned along with the information that file extension should be `"dtd"`.

## References

- [Aho96] H. Ahonen. *Generating grammars for structured documents using grammatical inference methods*. PhD thesis, Department of Computer Science, University of Helsinki, Series of Publications A, Report A-1996-4, 1996.
- [HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison-Wesley, 2001.
- [HW07] Yo-Sub Han and Derick Wood. Obtaining shorter regular expressions from finite-state automata. *Theor. Comput. Sci.*, 370(1-3):110–120, 2007.
- [jun] Java universal network/graph framework. <http://jung.sourceforge.net/>.
- [KMS<sup>+</sup>a] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. *jInfer Architecture*.
- [KMS<sup>+</sup>b] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. *jInfer AutoEditor automaton visualization and editor module*.
- [KMS<sup>+</sup>c] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. *jInfer TwoStep simplifier design and implementation*.
- [loo] org.openide.util.class lookup. <http://bits.netbeans.org/dev/javadoc/org-openide-modules/org-openide/modules/doc-files/api.html>.
- [mod] Module system api. <http://bits.netbeans.org/dev/javadoc/org-openide-modules/org-openide/modules/doc-files/api.html>.
- [Nor] Theodore Norvell. A short introduction to regular expressions and context free grammars. <http://www.engr.mun.ca/~theo/Courses/fm/pub/context-free.pdf>.
- [VMP08] Ondřej Vošta, Irena Mlýnková, and Jaroslav Pokorný. Even an ant can create an xsd. In *DASEAA'08: Proceedings of the 13th international conference on Database systems for advanced applications*, pages 35–50, Berlin, Heidelberg, 2008. Springer-Verlag.
- [wik] Regular expression. [http://en.wikipedia.org/wiki/Regular\\_expression](http://en.wikipedia.org/wiki/Regular_expression).