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, supporting basic features of the language.

2 Structure

The main class implementing `SchemaGenerator` inference interface and simultaneously registered as its service provider is `SchemaGeneratorImpl` in package `cz.cuni.mff.ksi.jinfer.basicxsd`. Process of export consists of two phases described in detail in later sections.

1. Preprocessing.
2. The export to a string representation itself.

Method `start()` first creates an 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 as which elements should be globally defined and which element is the root element.

Afterwards, `start()` method uses instances of classes derived from `AbstractElementsProcessor` class to export elements of input grammar.

2.1 Preprocessing

Code to handle preprocessing resides in package `cz.cuni.mff.ksi.jinfer.basicxsd.preprocessing`.

2.1.1 Purpose

As mentioned above, 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.

2.1.2 How does it work

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

If it is desired to not generate global types, this feature can be turned off in preferences. However, turning it off is not recommended and in some cases it may cause invalidity of resulting XSD output. For more information, see 3.

Minimal number of occurrences of an element to define its type globally can be altered if preferences, as well. See 2.3.

2.1.3 Running preprocessor and obtaining its result

As described above, preprocessing is performed by creating an instance of `Preprocessor` class.

Information discovered by the preprocessing can be obtained by calling `Preprocessor`'s `getResult()` method. This method returns an instance of `PreprocessingResult` class. Purpose of this class is to provide of what the preprocessor has discovered and to provide an easy way to search the input grammar for an element by its name. For details see JavaDoc of `PreprocessingResult`'s public methods.

2.2 Export

XSD export itself is performed using classes derived from `AbstractElementsProcessor` class and a helper class named `Indentator`.

2.2.1 Indentation

To generate a human readable XSD output, it is necessary to apply correct indentation of XSD elements and their content. This is handled by `Indentator` class. This classes also serves as a buffer for string representation of XSD that the exporter is creating.

Instance of this class 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 the resulting XSD.

Number of spaces characters per one level of indentation can be altered in project properties. See 2.3.

2.2.2 Definition of elements

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

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`. This means, that these 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 defines 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 the following one.

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

Exporter supports types of *mixed elements*. *Mixed element* is an element that contains other elements as well as some text.

```
<mixedElement>
  some text
  <anotherElement/>
  another text
</mixedElement>
```

Mixed element type is defined as `complexType` with attribute `mixed="true"`. Definition of the element from the last example may be as following.

```
<xs:element name="mixedElement">
  <xs:complexType mixed="true">
    <xs:sequence>
      <xs:element name="anotherElement" type="..."/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

2.2.3 Definition of attributes

Attributes are defined by XSD element `xs:attribute` with attributes `name`, `type` and optional `use`. Elements `xs:attributes` have to be placed at the end of a `complexType` definition.

```
<xs:element name="Person">
  <xs:complexType>
  ...
  </xs:complexType>
  <xs:attribute name="age" type="xs:positiveInteger"/>
  <xs:attribute name="id" type="xs:string" use="required"/>
</xs:element>
```

Attribute `type` is one of a built-in types. If an attribute is obligatory, this is defined by specifying `use="required"`.

2.2.4 Export of elements

Classes to handle export of elements are in package `cz.cuni.mff.ksi.jinfer.basicxsd.elementsexporters`. Basic common logic is implemented in `AbstractElementsExporter` class. This class is abstract and is supposed to be extended by classes with a particular purpose. Its constructor signature is defined as following.

```
public AbstractElementsExporter(final PreprocessingResult preprocessingResult,
                                final Indentator indentator)
```

Names of parameters are self explanatory. Parameter `preprocessingResult` is a result of preprocessing. Parameter `indentator` is an instance of `Indentator` class to be used to buffer and indent output of exporter. This instance has not to be empty. Output of exporter is appended at the end of text held by the `indentator`. This behaviour is convenient when chaining output of several elements exporters.

There are two classes extending `AbstractElementsExporter`. `GlobalElementsExporter` and `RootElementExporter`. Their constructors has the same signature as constructor of `AbstractElementsExporter` and both have `run()` method to perform their function.

`GlobalElementsExporter` retrieves global elements from the result of preprocessing and creates global definition of their types. These definitions are appended to the `indentator`. The `indentator` should be set to a level of indentation at which it is desired to append the global type definitions (typically, no indentation). After return from a `run()` method call, output of exporter is appended to the `indentator`.

If a global type contains another elements, these are processed as following.

- If a contained element is global or of a built-in type, its type is simply referenced.
- If a contained element is not global nor of a build-in type, it is fully defined in a place of its occurrence inside the global type.

Name of a global type is derived from name of a corresponding global element by prefixing and suffixing it. Default prefix is "T" and default suffix is empty. These values can be changed in project properties.

Examples of definition of global types.

Type generated from element named "GlobalElement", containing elements "Text", which is string, and element "AnotherGlobalElement", which is another global element.

```
<xs:complexType name="TGlobalElement">
  <xs:sequence>
    <xs:element name="Text" type="xs:string"/>
    <xs:element name="AnotherGlobalElement" type="TAnotherGlobalElement"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="TAnotherGlobalElement">
  ...
</xs:complexType>
```

Type generated from element named "X", containing element "Y", which is not global element and contains elements "Text" and "GlobalElement" from the previous example. Element "X" has one mandatory string attribute named "id".

```
<xs:complexType name="TX">
  <xs:sequence>
    <xs:element name="Y">
      <xs:complexType>
        <xs:element name="Text" type="xs:string"/>
        <xs:element name="GlobalElement" type="TGlobalElement"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="id" type="xs:string"/>
</xs:complexType>
```

`RootElementExporter` retrieves the root element from the result of preprocessing and creates its definition. This definition is also appended to the `indentator` (supplied to constructor) at a level of indentation, the `indentator` is set. Way of handling the root element's subelements depends on if these are global elements or not.

Global elements (retrieves from the result of preprocessing) are defined by referencing its type, which is supposed to be defined globally. This applies also to elements which are not global and their type is one of a built-in types. For example, root element "Root" contains one the following two elements. Global element "A" and string element "B".

```
<xs:complexType name="TA">
  ...
</xs:complexType>

<xs:element name="Root">
  <xs:complexType>
    <xs:choice>
      <xs:element name="A" type="TA"/>
      <xs:element name="B" type="xs:string"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

Elements which are not global and are not of a build-in type are defined recursively, at the place of their occurrence. For example, root element "Root" contains element "A", which contains two another string elements.

```
<xs:element name="Root">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="A">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="text1" type="xs:string"/>
```

```
            <xs:element name="text2" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      <xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

2.2.5 Export of attributes

Export of attributes is done along with export of elements, as attributes are exported as XSD element `xs:attribute` in XSD element `xs:complexType`.

```
<xs:complexType name="...">
  <xs:sequence>
    ...
  </xs:sequence>
  <xs:attribute name="id" type="xs:string"/>
</xs:complexType>
```

2.3 Preferences

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

- Turn off generation of global element types. Turning off this feature is not recommended as it may cause certain problems with validity of resulting XSD. See 3.
- Minimal number of occurrences of element to define its type globally. (Only if generation of global elements is active.)
- Number of spaces in output per one level of indentation.
- Global type name prefix. It is a string which will be inserted before a name of a type, which is derived from element's name. Can be also an empty string. (Only if generation of global elements is active.)
- Global type name suffix. It is a string which will be appended after a name of a type, which is derived from element's name. Can be also an empty string. (Only if generation of global elements is active.)

3 Known problems and limitations

This section describes some limitations of XSD export and some known problems, which occur in some specific cases of input grammar.

3.1 Namespaces

3.1.1 Description

This basic implementation of XSD export does not support and does not handle namespaces. That means that no namespace definitions are generated in a result, namespace definitions in input are processed as regular elements and attributes and namespace usages (namespace:element) are considered as a part of name of an element.

3.1.2 Workaround

To generate a valid XSD output, it is necessary to remove all namespace definitions and usages from input. If a presence of namespaces in a resulting XSD is necessary, they need to be inserted there manually.

3.2 XSD invalidity if generation of global element types is disabled

3.2.1 Error conditions

Following conditions have to be met to cause a problem.

- Generation of global element types if turned off (in project properties).
- Input grammar contains a concatenation or an alternation regexp with several (two or more) same elements.

3.2.2 Description

Example of a part of generated XSD.

```
...
  <xs:sequence>
    <xs:element name="A">
      ...
    </xs:element>
    <xs:element name="A">
      ...
    </xs:element>
  <xs:sequence>
...
```

In XSD, it is not allowed to define elements with the same name in one sequence (and other XSD constructs).

3.2.3 Workaround

Do not turn the generation of global element types off.

3.3 "Unique Particle Attribution" problem

3.3.1 Error conditions

This problem appears if input grammar contains an alternation of two or more concatenations, which begin with at least one same element. Part of a problematic XSD is for example following.

```
...
  <xs:choice>
    <xs:sequence>
      <xs:element name="A" type="TA"/>
      <xs:element name="B" type="TB"/>
    </xs:sequence>
    <xs:sequence>
      <xs:element name="A" type="TA"/>
      <xs:element name="C" type="TC"/>
    </xs:sequence>
  </choice>
...
```

3.3.2 Description

If error conditions are met, input XMLs are not valid against a generated XSD.

4 Data flow

Flow of data in this module is following.

1. `Preprocessor` topologically sorts elements (rules) it got on input.
2. For each element, its occurrence count is computed.
3. Unused elements (occurrence count equals 0) are removed.
4. For each element, it is determined if it type will be defined as a global type or not.
5. For each global element, its type is exported to a XSD representation by `GlobalElementsExporter`.
6. Root element and recursively all remaining elements are exported to a XSD representation by `RootElementExporter`.
7. String representation of the schema is returned along with the information that file extension should be "xsd".

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.

[Bou] Ronald Bourret. Dtd parser, version 2.0. `http://www.rpbourret.com/dtdparser/index.htm`.

[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+a] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. *jInfer Architecture*.

[KMS+b] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. *jInfer AutoEditor automaton visualization and editor module*.

[KMS+c] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. *jInfer Base Module Description*.

[KMS+d] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. *jInfer BasicDTDExporter Module Description*.

[KMS+e] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. *jInfer BasicIGG Module Description*.

[KMS+f] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. *jInfer BasicRuleDisplayer Module Description*.

[KMS+g] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. jinfer javadoc. `http://jinfer.sourceforge.net/javadoc`.

[KMS+h] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. *jInfer TwoStep simplifier design and implementation*.

[log] Apache log4j™. `http://logging.apache.org/log4j/`.

[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 *DASFAA'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`.