jInfer XSDImporter 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 those creating or extending a parser.

Responsible developer	Robert Smetana
Required tokens	none
Provided tokens	none
Module dependencies	listed for each module separately
Public packages	listed for each module separately

1 Introduction

This document focuses on the description of modules involved in the process of importing of XML Schema Definition (XSD) documents, or Schema documents. These modules are responsible for creating the IG rules from input files, therefore they belong in the inference modules category. As a whole, they are a specialized extension of `BasicIGG` module.

None of the modules provide nor require any tokens. Complete dependency tree is described in [KMS+a, section 4] and depicted in figure 9 of that document.

1.1 Naming convention

Note that there are several types of entities called "element" in the context of the described modules. To avoid ambiguity, in this document we shall use notation `DOM.Element` for instances of `org.w3c.dom.Element` and notation `Element` for instances of`cz.cuni.mff.ksi.jinfer.base.objects.nodes.Element`.

To add to this complication, XSD Schemas are XML documents which consist of elements and attributes. We shall refer to these XML elements as *tags* and XML attributes as *tag attributes*. Moreover, all tags and tag attributes that are known to our parsers are specified in section 2.1.4.

For example, the *attribute* tag can have a tag attribute *use*. While former will match `XSDTag.ATTRIBUTE`, the latter will match `XSDAttribute.USE`. Lastly, tag attributes should not be confused with instances of `Attribute`, which is a class for storing information about the tag attributes inside jInfer. Every `Element` has a list of `Attributes`, as can be seen from figure 4 in [KMS+a, section 2.3].

1.2 Namespaces

XSD Schemas allow using different namespace prefixes for tag names. Our parser implementations do not recognize namespaces; all prefixes from tag names are trimmed. For example, there is no difference between `xs:choice` and `xsd:choice` tags, both will match the `XSDTag.CHOICE` constant.

1.3 Rule-trees

Rule-trees are tree data structures, where vertices are instances of `Element` class and edges are the relations between them stored in `Regexp` instances within vertices. Every vertex is a rule by itself, so in order to obtain IG rules, one must extract all Elements from a rule-tree. Rule-trees are created by `DOMHandler`, described in subsection 2.2.2. The extraction is done by `DOMParser`, described in subsection 2.2.1. Or alternatively, when parsing with SAX is selected, `SAXHandler` handles both processes.

Life cycle of rule-trees is apparent from section 3.2.

2 Structure

There are three modules that provide support for importing Schema files bundled with jInfer.

- `XSDImporter` - Main module providing support for actual parsing modules, as well as common configuration options for the parsers.
- `XSDImportDOM` - Dependent module implementing DOM parser method.
- `XSDImportSAX` - Dependent module implementing SAX parser method.

All three modules depend on the `Base` module, as they all use the common data structures of jInfer. `XSDImporter` module also depends on `BasicIGG` module. DOM and SAX parser modules are independent of each other; however, they rely on classes from `XSDImporter`. They provide similar functionality, but use different paradigms.

2.1 XSD Importer module

Module dependencies	Base BasicIGG
Public packages	`cz.cuni.mff.ksi.jinfer.xsdimporter.interfaces` `cz.cuni.mff.ksi.jinfer.xsdimporter.utils`

This module has several key roles in the process of importing XSD documents.

Firstly, it is an extension of the `BasicIGG` module, registering a service provider of a `Processor` class, to handle files with *xsd* extension (see [KMS+e, section 3]). It defines a common interface providing the developers with easy integration of their custom parsers to the rest of the inference process. It also provides a common configuration in the form of a properties panel in project properties for all depending modules. And finally, this module holds common utility classes related to the task of processing data characteristic to the Schema documents.
Functionality is divided between separate packages, in the same order as above:

- `cz.cuni.mff.ksi.jinfer.xsdimporter`
- `cz.cuni.mff.ksi.jinfer.xsdimporter.interfaces`
- `cz.cuni.mff.ksi.jinfer.xsdimporter.properties`
- `cz.cuni.mff.ksi.jinfer.xsdimporter.utils`

Since some of the packages contain only one important class, we structure the documentation in logical blocks based on significance, not package structure.

2.1.1 XSD Processor

Similar to the `Processor` implementations in the `BasicIGG` module, this is the core class of `XSDImporter` logic. `XSDProcessor` registers itself into *schema* input folder, *xsd* file extension and declares it cannot handle other file extensions. Class depends on NBP lookups (see [KMS+a, section 4.1]) to find available parsers, and uses the selected parser to perform the import operation on defined files. Initial grammar generated by this processor is considered *complex* and should be followed by expansion if necessary (see [KMS+e, section 4]).

2.1.2 XSD Parser interface

Key interface which needs to be implemented by any parser handling the import of Schema documents. Lookups of available parsing methods work by searching for service providers based on this interface. In order to support the lookup functionality, this interface extends `NamedModule`. It declares method `parse()` which takes an `InputStream` that contains the input document and returns a complete list of rules from that document. In other words, this method does all the necessary work. Method has the following signature:

```
List<Element> parse(InputStream stream) throws XSDException;
```

Diagram showing the inheritance of this interface, as well as both parsers bundled with jInfer, which implement it is in figure 1.

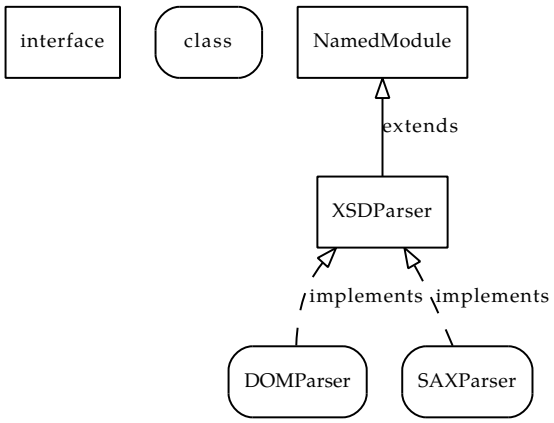

Figure 1: Inheritance of XSDParser interface.

2.1.3 Properties

Settings provided by the package `cz.cuni.mff.ksi.jinfer.xsdimporter.properties` are project-wide. They are meant to be common for all parsers handling the import of Schema documents. Access to these settings is provided by utility class `XSDImportSettings` described in section 2.1.4. For description of each of the settings, please see section 5.

2.1.4 Utils

Package `cz.cuni.mff.ksi.jinfer.xsdimporter.utils` contains classes available to all parsing methods.

Built-in data types A special class is designed to contain all built-in data types supported by XSD language. Both built-in primitive types and built-in derived types are recognized. For a complete list, please see our JavaDoc documentation for `XSDBuiltInDataTypes` class. These types are contained using instances of `SimpleData` class, as described in [KMS+a, section 2.2].

XSD Import Settings Simple utility class that provides convenient access methods to all the settings, hiding unnecesary implementation details. It holds the following methods.

- `getParser()` returns selected instance of `XSDParser`.
- `logLevel()` returns current log priority, messages with lower priority are not displayed.
- `isVerbose()` returns `true` or `false`, depending on state of the checkbox.
- `stopOnError()` returns `true` or `false`, depending on state of the checkbox.

XSD Occurences Helper class for creating intervals from string representation of occurrences found in `minOccurs` and `maxOccurs` attributes of schema tags. Default values for both attributes are 1. Methods can create all types of intervals. If optional flag `minHasPriority` is set, method `createInterval` will prefer the value of lower bound, if parsing of upper bound fails, which can result in an interval with equal values of lower and upper limit (`min, min`).

XSD Attribude Enumeration of tag attributes currently recognized by our parsers. Others are simply skipped. They are the following.

- `name`
- `ref`
- `type`
- `use`

- `minOccurs`
- `maxOccurs`

List should be extended according to the parser, to reflect its functionality.

XSD Tag Enumeration of tags currently known by our parsers. Others are simply skipped. They are the following.

- `schema`
- `element`
- `complexType`
- `sequence`
- `choice`
- `all`
- `attribute`
- `redefine` - this tag is not yet supported, but allows parsing of a certain types of documents

This list should also be updated in case the parser functionality is extended.

XSD Utility This is basically only a library of short, useful and frequently used functions for parsing, please refer to the JavaDoc.

XSD Exception `XSDException` is a simple class extending `RuntimeException` to announce errors during import operation. It is catched by `XSDProcessor` which checks if *Stop on error* setting is enabled and stops the inference.

2.2 XSD Import DOM module

Module dependencies	Base XSDImporter
Public packages	none

This module implements the import functionality using API from Xerces DOM parser. Its structure is very simple, all classes are grouped in one main package, relying on utility classes from `XSDImporter` module mentioned before.

2.2.1 DOM Parser

Class implementing `XSDParser` interface. It is responsible for creation of DOM tree, extraction of contained rule-trees and aggregation of rules from them. First task is delegated to the underlying Xerces DOM parser API. Conversion between DOM tree and the rule-trees is then passed to `DOMHandler`.

```
parser = new com.sun.org.apache.xerces.internal.parsers.DOMParser();
parser.parse(new InputSource(stream));
Document doc = parser.getDocument();
DOMHandler handler = new DOMHandler();
List<Element> ruleTrees = handler.createRuleTrees(doc.getDocumentElement());
```

2.2.2 DOM Handler

When DOM parsing method is selected, most of the logic is implemented in `DOMHandler` class. This class is responsible for creating so called rule-trees from an existing Document Object Model.

Its only public method iterates through children of the root node, given as parameter, in two passes. This root node should always contain `schema` tag, according to XSD specification. Firstly, it registers `complexType` and `element` tags, as these can be referenced from within the tree. And then launches the algorithm for creating rule-trees for relevant nodes. It can be summarized as follows:

```
public List<Element> createRuleTrees(org.w3c.dom.Element root) {
  for each (root.getChildNodes() as tag) {
    if (tag = XSDTag.ELEMENT)     addToReferenced(tag);
    if (tag = XSDTag.COMPLEXTYPE) addNamedCType(tag);
  }
  for each (root.getChildNodes() as tag) {
    subtree = buildRuleSubtree(tag);
    ruleTrees.add(subtree);
  }
  return ruleTrees;
}
```

The most important method is `buildRuleSubtree()`. It is divided into four logical parts.

1. Creates an `Element` instance with appropriate parameters (occurences, metadata, etc.).
2. Inspects the given tag and sets parameters for subnodes `Regexp`.
3. Recursively creates rule-subtrees for its children.
4. Finalizes the instance by checking for extra conditions.

Signature of the method is:

```
private Pair<Element, RegexpInterval> buildRuleSubtree(
          org.w3c.dom.Element currentNode,
          List<String> context,
          List<org.w3c.dom.Element> visited) throws XSDException
```

Method takes current node, which will be returned as root `Element` node (first value of the returned pair) of this subtree, and creates an entire rule-tree below it by recursively calling itself, as mentioned in step 3. Context parameter holds the list of names of nodes that create unique path from actual root of the schema document to the current node. Visited parameter lists references to instances of `Element` that were visited on this path. When handling *attribute* tags, context needs to be passed along, but list of visited nodes does not.

2.2.3 DOM Helper

Helper class for `DOMHandler`, containing ulility functions specific for DOM parsing method. Two most notable of them are `createSentinel()` and `finalizeElement()`.

CreateSentinel() Creates an instance of `Element` class, which is always a leaf in the rule-tree. This is useful when there are cyclic definitions in an input Schema document, or an `Element` with this name was defined previously and it is unnecessary to build a rule subtree for it again (it is a reference in this case). More information about sentinels can be found in [KMS+a, section 3.2], or [KMS+h, section 4].

FinalizeElement() Checks if `Element` is properly defined. Redefine it to `LAMBDA` when it was empty, or to `TOKEN` if it only contained a simple data type. This method allows for built-in data types of tags to be retained for export.

2.3 XSD Import SAX module

Module dependencies	Base XSDImporter
Public packages	none

This module implements the import functionality using API of a standard SAX parser. Its structure is divided in two packages. Package `cz.cuni.mff.ksi.jinfer.xsdimportsax` holds the main parsing logic, while package `cz.cuni.mff.ksi.jinfer.xsdimportsax.utils` contains two utility classes. Both `SAXAttributeData` and `SAXDocumentElement` serve as wrappers for tag attributes and tags of the input Schema file during parsing, to ease the process of extracting data from them.

Classes `SAXParser` and `SAXHelper` are very similar to their counterparts in `XSDImportDOM` module, for information about their methods please refer to the JavaDoc. Implementation is also relying on utility classes from `XSDImporter` module mentioned before.

2.3.1 SAX Handler

Sax handler implements methods in response to the events generated by the underlying SAX parser API. Similar to its counterpart `DOMHandler`, this class holds most on the logic for importing a Schema. It works by keeping current tree branch, which is being parsed, on a stack. Adding current node to the parent on top of the stack as current tag closes.

3 Data Flow

Parsing of XSD begins when `IGGeneratorImpl` calls `XSDProcessor` with an input file. The overall data flow while parsing a Schema document is simple, it can be summarized into following steps.

1. `XSDProcessor` finds out the selected `XSDParser` from available service providers. This also indicates which module will be handling the actual parsing.
2. `XSDProcessor` passes the input document to a selected module by calling `parse()` method on the selected `XSDParser`.
3. Depending on the module, the corresponding API is used to extract IG rules from the input document.
4. List of rules that were extracted from the document are returned to `XSDProcessor`.
5. If no errors occured during this process, list of extracted rules is returned to `IGGeneratorImpl` for expansion (if needed). Otherwise, depending on the *Stop on error* setting, the inference is stopped, or an empty list of rules for this input document is returned.

3.1 Data Flow using DOM parser

When using the DOM parsing method, extraction of rules from a Schema document, mentioned in step 3 above, proceeds as follows.

1. Document Object Model tree data structure is created from the input file by *Xerces DOM parser*.
2. Root node of the DOM tree is handed over to `DOMHandler` class.
3. `DOMHandler` recursively passes the DOM tree by a depth-first algorithm and creates complete subtrees of rules (rule-trees) for each of the direct children of root node.
4. Rule-trees are returned from `DOMHandler`.
5. `DOMParser` traverses the returned rule-trees, depth-first, to create a list of final rules. Each of the rules now contains a full path, thanks to the way in which they were created.

3.2 Data Flow using SAX parser

Due to the nature of SAX parser paradigm, it is not possible to create the tree data structure in the top-to-bottom fashion, as with DOM parser. However, using stack data structures, analogous rule-trees are created from bottom to top, leading to a similar data flow.

1. A `SAXParserFactory` is created and our implementation of handler class `SAXHandler` is passed to it, along with `InputStream` containing an input document.
2. Our handler then sequentially responds to events generated by the SAX parser, creating `Element` nodes when appropriate, and keeping them on a stack. This way, a relation can be created by adding an `Element` as a child to the `Element` on top of the stack.
3. When parsing is finished, the rule-trees are kept in the data structures of `SAXHandler`.
4. `SAXParser` calls `getRules()` method of `SAXHandler` to traverse the rule-trees to create the list of final rules in a similar manner as in DOM case. These rules also contain a full path.

4 Known issues and limitations

In this section we describe known problems that can occur while importing Schema documents. Supported structures are discussed only; tags and tag attributes that are not supported are not in scope of this section.

4.1 SAX parser limitations

Due to the inherent difficulty of SAX parsing method to cope with cross-referencing definitions, there is a known issue when schema defines *complexType* tags in the following manner.

```
<xsd:complexType name="folderType">
  <xsd:sequence>
    <xsd:element name="file" type="fileType"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="fileType">
  <xsd:attribute name="ID" type="xsd:ID"/>
</xsd:complexType>

<xsd:element name="folder" type="folderType"/>
```

In this example definition, the `folderType` type is dependent on `fileType` type, which has not been defined when a new Element, named `file`, is trying to resolve its type. This results in the parser assigning empty children to the `file` Element, adding the Element to a list of unresolved Elements and continuing with the parsing process. When parser reads the complete definition of `folderType` type, the type is only added to a list of known types, as there are no data structures for listing type cross-dependencies.

When the complete definition of `fileType` is read, this type is also only added to a list of known types, no dependencies are retrospectively resolved. The reason for it lies in one-pass approach of SAX parser. In future versions, we are planning to resolve this issue by creating dependency trees of `complexType` tag definitions. However, it will make the parsing algorithm more time and memory expensive.

Known fix is to place definitions of dependent types after the types they depend on have been defined, or simply using DOM parser, which is devoid of such complications by design. Using either will result in a desired output:

```
<xs:complexType name="Tfile">
  <xs:attribute name="ID" type="xs:ID"/>
</xs:complexType>

<xs:complexType name="Tfolder">
```

```
  <xs:sequence>
    <xs:element name="file" type="Tfile"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="folder" type="Tfolder"/>
```

5 Settings

Common settings of modules can be found in project properties under *XSD Import* category. They are project specific. Following list explains their purpose.

Parsing method Setting selects whether SAX or DOM parser is used. This selection is based on the number of registered service providers for the `XSDParser` class, so that both the jInfer bundled parsers as well as any custom parsers will be automatically displayed here. DOM parser is set as the default parsing method.

Log level Setting determines the minimal priority for a message from the selected parser module to be logged. For example, if the module provides extensive debug information about elements being currently parsed, these messages can be filtered by setting the log level higher than `DEBUG`.

Stop on error Checkbox determines if parsing halts the inference on the first error. If it is not set, then the file that caused the error is simply skipped and not imported.

Print verbose info Setting enables logging of some additional information, for example, complete list of imported rules, or extended warning messages.

6 Extensibility

Module `XSDImporter` itself is an example of an extension of `BasicIGG`, for which it provides a specific funcionality. Other modules that handle different definition languages can be created in a similar fashion.

The modular design of the parser implementations is also an example of extensibility which can be achieved using the NetBeans platform. It allows developers to add, or remove, custom parsers just by creating a module with a new parser implementation; there is no need to change any code within `XSDImporter` module. Only exception is the redefinition of the default parser (currently DOM), which is not needed even if the module of default parser is not present.

Parsers with extended or custom functionality can be created using `XSDImportDOM` module as example (recommended).

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`.

[gra] Graph visualization software. `http://www.graphviz.org/`.

[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.

[JAX] Java architecture for xml binding. `http://jaxb.java.net/`.

[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`.

[pro] Project sample tutorial. `http://platform.netbeans.org/tutorials/nbm-projectsamples.html`.

[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.

[Vyh] Julie Vyhnanovská. Automatic construction of an xml schema for a given set of xml documents.

[wik] Regular expression. `http://en.wikipedia.org/wiki/Regular_expression`.

[xml] Xml validation api. `http://download.oracle.com/docs/cd/E17802_01/webservices/webservices/docs/1.6/api/javax/xml/validation/package-summary.html`.