jInfer BasicIGG 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 modify the way *BasicIGG* creates initial grammar from input files, for example by adding support for a new schema language.

Responsible developer:	Matej Vitásek
Required tokens:	none
Provided tokens:	cz.cuni.mff.ksi.jinfer.base.interfaces.inference.IGGenerator
Module dependencies:	Base
Public packages:	cz.cuni.mff.ksi.jinfer.basicigg.properties

1 Introduction

This is an extensible implementation of the *IGGenerator* inference interface. It is the only IG generator officialy shipped with jInfer.
Make sure you understand the difference between the two types of initial grammar as described in **??**.

2 Structure

The main class implementing `IGGenerator` inference interface and simultaneously registered as its service provider is `IGGeneratorImpl`. In its `start` method it enumerates all files in the input parameter (of type `Input`). For each file, based on its extension the correct *processor* is selected, executed and returned rules are aggregated. After each file has been processed, the resulting grammar is returned by invoking the `finished` method of the callback argument. This process is illustrated in fig. 1.

3 Processors

A *processor* is a class capable of extracting IG from an arbitrary `InputStream` (usually encapsulating a file). Various processors may handle generic XML, schemas like DTD, XSD or Schematron, query languages such as XPath, and so on.
A processor has to be registered as a service provider of the `Processor` interface from *Base* module. Due to the nature of NBP lookups, each processor is internally kept as a singleton, and should not use its inner state (refer to the chapter Lookups in **??**). Note that the factory pattern is not used here.
Each processor declares the class of inputs (documents, schemas, queries) and file extensions it is able to handle by

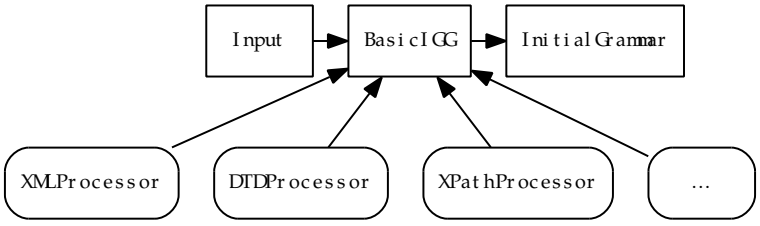

Figure 1: Process view of BasicIGG

implementing methods `getFolder()`, `getExtension()` and `processUndefined()`. Refer to JavaDoc of these methods for further details.

BasicIGG comes bundled with 3 processors: for generic XML documents, DTD schemas and XPath queries. Support for XSD queries is implemented in *XSDImporter* to demonstrate *BasicIGG*'s extension capabilities.

3.1 XML processor

XML processor registers itself into *document* input folder, *xml* file extension and declares that it can process other arbitrary file extensions.
SAX traversal is used to collect the rules; the relevant `ContentHandler` is the class `TrivialHandler`. The way it works is following: every time a start of an element is encountered, a new `Element` is created along with its attributes. This new element representation is then placed on the top of a stack. Every other element or simple data found until its ending tag is created and attached to its subnodes. When an end of an element is encountered, the element currently on the top of the stack is closed and declared to be a rule of the IG.
After reaching the end of the document, all IG rules are returned. Note that this approach creates "simple" initial grammar.

3.2 DTD processor

DTD processor registers itself into *schema* input folder, *dtd* file extension and declares it cannot handle other file extensions.
To parse DTD files a 3rd party library is used: *dtdparser*. Translation from the object model in this library to our own is handled mostly by `DTD2RETranslator` class.
Note that the initial grammar generated in this way is complex. Therefore, this processor checks for the `CAN_HANDLE_COMPLEX_RE` capability of the simplifier following this IG generator: in positive case it simply returns the grammar, if the following module cannot handle complex regular expressions, DTD processor first invokes an *Expander* (see 4).

3.3 XPath processor

This is a rather naïve implementation of an XPath processor. It registers itself into *query* input folder, *xpath* file extension (text file containing one XPath query per line) and declares it cannot handle other file extensions.
To parse XPath queries, standard support present in JDK is used. The relevant `XPathHandler` is the `XPathHandler` class.

4 Expansion

Expansion is used to convert complex regular expressions in initial grammar to simple ones (concatenations of tokens) for simplifiers that can handle only the simple form. The process can be seen as breaking down a regular expression into a set of words (positive examples) that are described by this regexp. Relevant interface encapsulating any implementation of such an expander is `Expander` in *Base*. Reference implementation is `ExpanderImpl` in this module. Note that even though this implementation is retrieved using lookups, jInfer bundles only one such implementation and does not support choice among more of them. Anyone wishing to implement his own expander will thus have to either remove jInfer's implementation, or implement the usual module selection.

4.1 ExpanderImpl internals

`ExpanderImpl` works recursively, and it will probably be the best to explain it from bottom to up.

At this point, let's denote a *word* over an *alphabet* typed (in Java generics sense) T as a `List<T>`. Following the same logic, let's denote a set of such words as `List<List<T>>`. Our aim while expanding will be to convert a regular expression representing some node's subnodes to a list of words that are described by this regexp. In this case, T will be `AbstractStructuralNode`.

The first interesting function is `ExpansionHelper.applyInterval()`. It takes a list of words extracted from a regexp (thus in the List<List<AbstractStructuralNode>> form) and a `RegexpInterval` and creates a new list of words: applying the interval on each of the words in a heuristic sense. The behaviour depends on the type of the interval.

- *Once*: the resulting set of words is exactly the original set of words.
- *Optional*: same as above, but an empty word is added to the resulting set (each word from input *may* or *may not* occur - but one empty word is enough).
- *General interval*: has a minimum and a maximum number of occurences. Theoretically, all counts in interval $[min, max]$ should be considered, but for practical reasons the following heuristic is employed: each word from the input is duplicated[1] first min, then max times and added to the result.
- *Kleene star*: similar to the general interval and *optional*, the result contains an empty word. Then for each word on input: this word and its $3\times$ duplication is added to the result. Obviously, this is a heuristic approach again - theoretically each word should be duplicated between 0 and ∞ times in the result.
- *Kleene cross*: except for not including empty word, this is the same case as *Kleene star*: each input word is duplicated 1 and 3 times in the result.

Understanding how intervals affect the expansion, it is time to move a level up: how a regular expression is expanded. This is done in `ExpanderImpl.unpackRE()` performing a recursive descent into the regexp tree and returning a set of words. Based on the regexp type, one of the following is performed.

- *LAMBDA*: resulting word set contains one empty word.
- *TOKEN*: resulting set contains a word with one letter: the content (an `AbstractStructuralNode`) of this token; the interval of the regexp is applied on this word before returning it.
- *ALTERNATION*: first, all children of this alternation are recursively evaluated. Their respective words are appended in a list, this list has the regexp interval applied and is returned. Reasoning is the following: we are trying to get all the words a regexp can produce. If the regexp is an alternation, any word that can be produced by one of its children can be a word produced by the regexp itself.
- *PERMUTATION*: a heuristic is applied again: because there are $n!$ ways to order children of a permutation, only two are actually picked: the original order in which they appear, and reversal of this order. In both cases the children are treated as if they we concatenation (and thus the following case is invoked) and appended in a list. On this list the regexp interval is applied and the list is returned.
- *CONCATENATION*: this is by far the most complicated case. Let's denote word sets produced by the N children of the concatenation $w_1 \dots w_N$. Now, all words that could be produced by this concatenations correspond to concatenations of form $w_{a_1} w_{a_2} \dots w_{a_N}$ for all possible combinations of indices $a_i, i \in \{1 \dots N\}$. Number of such words grows exponentially with N, therefore a heuristic approach has to be employed again.
  In this case, consider $m = \max_{i \in 1 \dots N} |w_i|$. Now, m words will be constructed in the following way: in a for-loop enumerating $i \in \{1 \dots m\}$ the j-th part of this i-th word will be $w_{imod|w_j|}$. The method `ExpanderImp.unpackConcat()` illustrates this.

Now for the next part, description of `Element` expansion. This is the content of `ExpanderImpl.expandElement()` method. First, element's subnodes are expanded as a regexp. For each word from this expansion a new `Element` is created: letters of this word are encapsulated in tokens, these tokens are put in a concatenation that constitutes subnodes of this new element. If the letter is an element itself, it is not recursively expanded - it is just declared to be a sentinel.

Finally, expanding a grammar means expanding each of its rules (elements) and returning the gathered rules (elements).

[1] Duplicating a word abc N times results in a word $\underbrace{abcabc \dots abc}_{N\times}$.

5 Data flow

Flow of data in this module is following.

1. IGGeneratorImpl walks over input files in a loop.
2. Each file gets processed by a processor based on its folder and extension.
3. Rules from all files are gathered in a single list (IG) and returned via a callback.

6 Extensibility

BasicIGG can be easily extended to support a new input type: just create a class implementing `Processor`, annotate it as a service provider and implement any logic needed. *XSDImporter* is an example of this.

It is possible to replace the default `Expander` implementation: but as mentioned in 4, either the old implementation must be removed or module selection must be introduced.