jInfer Architecture

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.

*Note: we use the term **inference** for the act of creation of schema throughout this and other jInfer documents.*

The description of jInfer architecture will commence by describing the data structures, namely representations of regular expressions and XML elements, attributes and simple data.
Afterwards the interfaces of basic inference modules - Initial Grammar Gen- erator, Simplifier and Schema Generator - will be explained.
Finally, the process of inference will be described.

1 Package naming conventions

All packages start with `cz.cuni.mff.ksi.jinfer`. Afterwards is the short, normalized name of the module (e.g. `base`) and finally the package structure in this module (e.g. `objects.utils`). All in all, a package in the Base module could look like `cz.cuni.mff.ksi.jinfer.base.objects.utils`

2 Data structures

2.1 Regular expressions

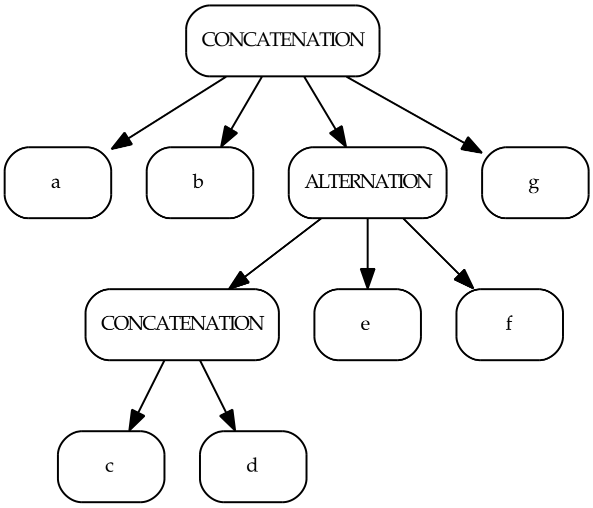
For general information on regular expressions, please refer to [wik], [HMU01]. All classes pertaining to regular expressions can be found in the package `cz.cuni.mff.ksi.jinfer.base.regexp`. In jInfer, we use extended regular expressions as they give us nicer syntax (and easier programming).
Regular expression is implemented as class `Regexp` with supplementing classes `RegexpInterval` and `RegexpType`. Each `Regexp` instance has one of the enum `RegexpType` type:

- Lambda - empty string (also called ϵ in literature),
- Token - a letter of the alphabet,
- Concatenation - one or more regular expression in an ordered sequence. Eg. (a, b, c, d),
- Alternation - a choice between one or more regular expressions. Eg. $(a|b|c|d)$,
- Permutation - shortcut for all possible permutations of regular expressions. Our syntax to write down permutation is $(a\&b\&c\&d)$.

Type of regexp is held in `type` member in class `Regexp` and can be tested by calling methods `isLambda()`, `isToken()` etc.
Each `Regexp` instance has one instance of `RegexpInterval` as member. Class `RegexpInterval` represents POSIX-like intervals for expression:

- $a\{m, n\}$ means a at least m-times, at most n-times,
- $a\{m, \}$ means at least m-times (unbounded).

Figure 1: Example tree for regular expression $(a, b, ((c|d), e), f)$



Interval is either bounded (you have to set both lower and upper bound integers), or unbounded (you have to set only lower bound). Testing interval commonly follows routine:

```
RegexpInterval i = r.getInterval();
if (i.isUnbounded()) {
  print(i.getMin());
} else {
  print(i.getMin(), i.getMax());
}
```

That is, first check interval for being unbounded, only if it is bounded, you can ask for maximum.

Class `Regexp` can represent regular expression over any alphabet. This is done by using java generics, `Regexp` evinceis implemented as `Regexp<T>`. Only token regexps hold instance of type `T` in member `content`.

Regular expression is in fact n-ary tree, for example expression $(a, b, ((c|d), e), f)$ can be viewed as in fig. 1. We implement this tree by member of `Regexp` class called `children`, which is of type `List<Regexp<T>>`. List contains children of regexp in means of regexp tree.

Regexp has to obey constraits:

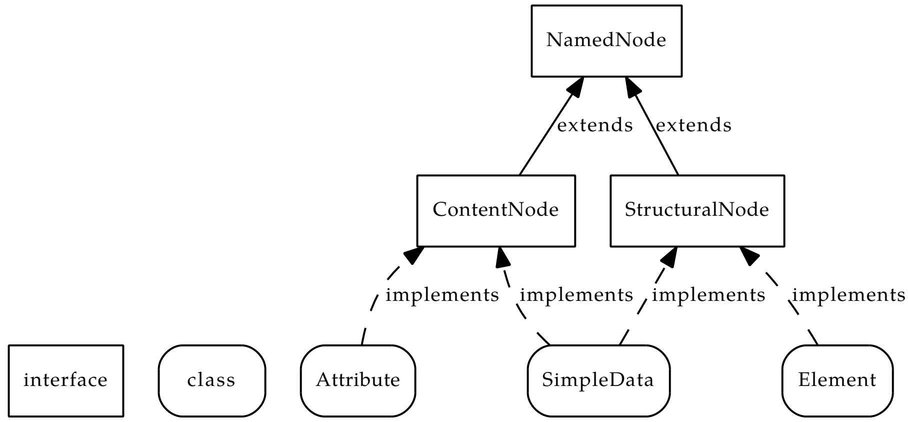
- type, children and interval have to be non-null references,
- when type is lamba, content and interval has to be null,
- when type is token, content has to be non-null,
- when type concatenation, alternation or permutation, content has to be null.

These constraits are checked by constructors, so the best way to construct new regexps is by using methods `getToken()`, `getConcatenation()` etc.

Regexp instance is by default created as immutable, that is, once instantiated, you cannot add more children to list of children, cannot change type, content etc. It is to prevent missuse. In special circumstances, one does not know future children of regexp in time of creation. This occurs mainly in input modules, where by parsing XML data sequentially, one does not know contents of element in time of handling start element event. For these cases, special `getMutable()` method is implemented to obtain regexp with none of members set. One has to fill in all properties carefully and call `setImmutable()` aftewards. Proper usage should be one of following:

```
Regexp<T> r = Regexp.<T>getMutable();
r.setInterval(...);
r.setType(RegexpType.LAMBDA);
r.setImmutable();cz.cuni.mff.ksi.jinfer.base
```

Figure 2: How should interfaces and classes for XML representation look like in theory




```
Regexp<T> r = Regexp.<T>getMutable();
r.setInterval(...);
r.setType(RegexpType.TOKEN);
r.setContent(...)
r.setImmutable();

Regexp<T> r = Regexp.<T>getMutable();
r.setInterval(...);
r.setType(RegexpType.CONCATENATION);
r.addChild(...);
r.addChild(...);
r.addChild(...);
r.setImmutable();
```

Finally, regexp contain one useful method for obtaining all leaves in the regexp tree, it is called `getTokens()` and it recursively traverses tree returning list of leaves (token type regexps).

2.2 XML representation

XML data basically contains elements, text nodes (characters inside elements) and attributes. For maximum generality, we decided to break apart theese objects. We define three basic interfaces: `NamedNode`, `StructuralNode` and `ContentNode` (see package `cz.cuni.mff.ksi.jinfer.base.interfaces.nodes`).

The first stands for bare node in XML document tree, it has its name and context withing the tree (path from root). The latter two extends `NamedNode` interface. `StructuralNode` is for nodes, which form structure of XML document tree: elements and text nodes. `ContentNode` is for nodes, that have content in XML documents: text nodes and attributes. We have three classes: `Element` for elements, `SimpleData` for text nodes, `Attribute` for attributes (see package `cz.cuni.mff.ksi.jinfer.base.objects.nodes`). In theory, the classes and interfaces would be layed out as on fig. 2

For even more generality in design, we decided to implement abstract classes in midlevel:

- `AbstractNamedNode`, which implements methods from `NamedNode` interface to handle context, name and metadata (will discuss later),
- `AbstractStructuralNode`, which implements only task of deciding if instance is `Element` or `SimpleData` actually.

As practice showed, for methods handling and infering structural properties, it is important to recognize whether structural node on input is element or text node. However methods for content devising don't need to know, if they are working on infering model for content of attribute or text node.

Finally, our interface/class model for representing XML nodes is drafted on fig. 3. Those, who are brave enough, can look on fig. 4.

Figure 3: How are interfaces and classes for XML representation arranged in practice

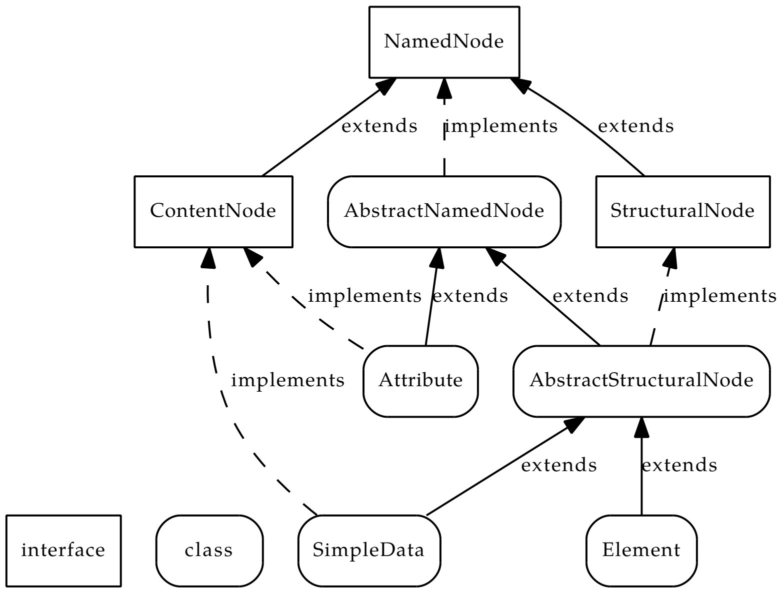
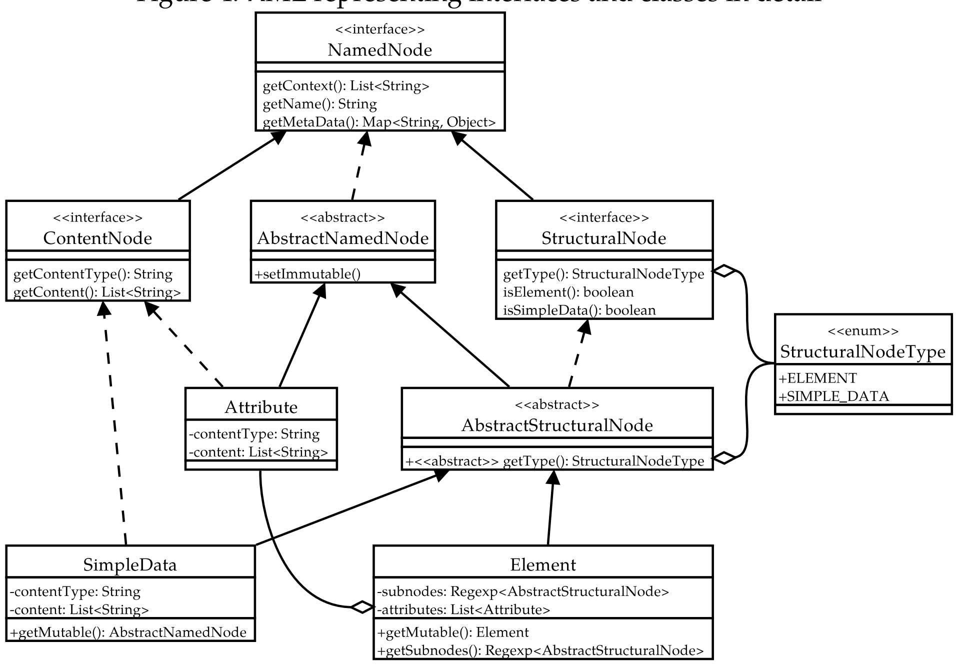


Figure 4: XML representing interfaces and classes in detail



In result, `Element` and `SimpleData` have method `getType()` to devise type of `AbstractStructuralNode` variables. And `SimpleData` and `Attribute` have methods `getContentType()` and `getContent()` to work with content model.

Class `Element` has two important members of course:

- `Regexp<AbstractStructuralNode> subnodes` - for representing right side of grammar rule in resulting infered schema,
- `List<Attribute> attributes` - for representing all attributes in resulting infered schema.

Theese two are filled by import modules, processed further by infering (simplifying) modules and finally exported by exporter modules. We will look at proper interfaces later.

Let's take an example, the following XML document would be represented as on fig. 5.

```
<person name="john" surname="smith">
  <info>
    Some text
    <note/>
  </info>
  <more/>
</person>
```

As in regular expressions, classes pertaining XML nodes are by default immutable. For elements, it means no adding of attributes and changing regexp reference (regexp instance itself is immutable). Same `getMutable()` principles and good usage practises hold for theese classes.

2.3 Rules, Grammars, etc

jInfer and its documentation uses extended context-free grammars[ext]. *Rules* in such grammar are in the form

$$\text{Left Hand Side (LHS)} \rightarrow \text{Right Hand Side (RHS)}$$

where LHS is a letter of the alphabet (token), RHS is a regular expression over this alphabet. Example would be

$$a \rightarrow b, (c|d)*$$

In jInfer each such rule is represented with an `Element` instance. In this representation, the `Element` itself is the LHS, its `subnodes` are the RHS.

Another important notion is a *grammar*. A grammar consists of its rules, so in jInfer a grammar is just a collection of `Elements`. Closely related term is *Initial Grammar*, which for us is a grammar consisting of rules with *simple* right hand sides, i.e. just concatenations of tokens. Initial Grammar is produced by Initial Grammar Generator.

2.4 Nondeterministic Finite Automaton

For all inference algorithms based on merging states of NFA's, our implementation of nondeterministic finite automaton might be interesting. Implementation consists of 4 classes: `Automaton<T>`, `Step<T>`, `State<T>` and `AutomatonCloner<A, B>` (see package `cz.cuni.mff.ksi.jinfer.base.automaton`). Whole implementation uses java generics for representation of symbol of alphabet, we denote by `T` the java type of symbol.

Let's begin by smalles of the classes, the `State<T>`. It represents automaton state, it has its `name` member (integer)

3 Inference process

The process by which jInfer infers the resulting schema from various inputs (inference process) is summarized by fig. 6. From the high-level viewpoint, it consists of three consecutive steps carried out by three different modules:

1. Initial Grammar (IG) generation: done by the *Initial Grammar Generator (IGG)* module, this is the process of converting all of the inputs to IG representation. All documents, schemas and queries selected as input are evaluated, simple rules are extracted and in the end sent to the next step.
   For example, a trivial XML document

Figure 5: XML document representation

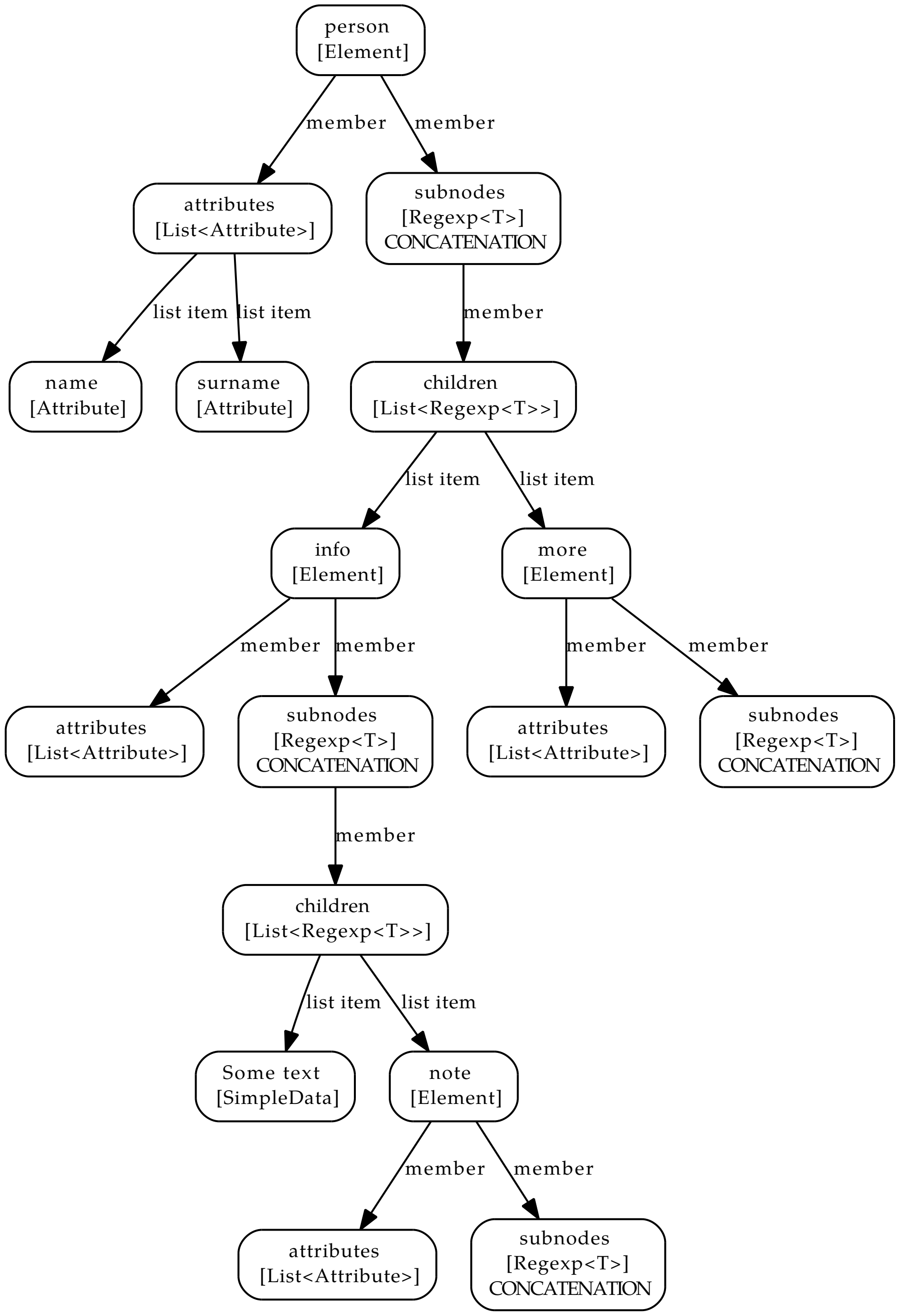
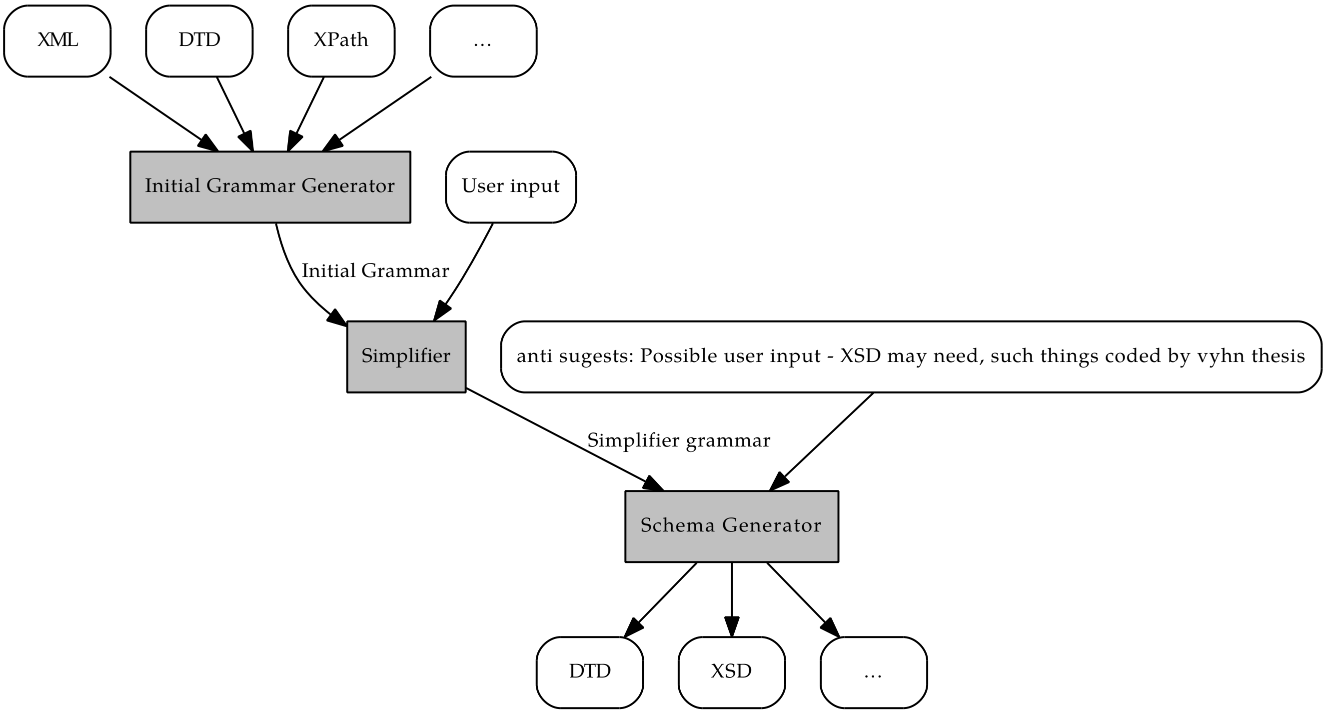


Figure 6: High-level view of the inference process




```
<a>
  <b>
    <c/>
  </b>
  <d/>
</a>
```

would translate into the following IG rules

$$\begin{array}{rcl} a & \rightarrow & b, d \\ b & \rightarrow & c \\ c & \rightarrow & \lambda \\ d & \rightarrow & \lambda \end{array}$$

2. Simplification: done by the *Simplifier* module, this is the process of simplifying, compressing or somehow compactly describing the IG by a smaller number of (more complex) rules. User interaction might be used in this step to help achieve better simplification. At the end of this step, all rules are sent to the last step.
   For example, rules

$$\begin{array}{rcl} a & \rightarrow & b \\ a & \rightarrow & c, d, d, d \end{array}$$

   could be simplified to a single rule

$$a \rightarrow b|(c, d*)$$

3. Schema export: done by the *Schema Generator (SchemaGen)* module, this is the process of actually creating the resulting schema from the simplified rules. Result of this step is a string representation of the schema, which is sent back to the framework (and later displayed, saved, etc).

Important thing to note here is that all these steps are executed consecutively. That means, *Simplifier* is only started *after* the *IGG* completely finished its work and returned IG to be simplified. Similarly, *SchemaGen* gets all the rules to export at once, in one list.

This modular architecture means that (at least theoretically) it is possible to replace any and all of these modules with *something* else that does similar job. In practice, it might be useful to use the logic already implemented in *BasicIGG* and just extend it to handle new input type.

3.1 Programmatic view

From developer's point of view, inference modules are just properly annotated classes implementing one of the following interfaces (all found under `cz.cuni.mff.ksi.jinfer.base.interfaces.inference`)

- `IGGenerator`
- `Simplifier`
- `SchemaGenerator`

A nice way to name such a class is by adding `-Impl` to the name of implemented interface, for example `SimplifierImpl`.

Annotation required for the framework to recognize such a class as an inference module is the following

```
@ServiceProvider(service = <interface>.class)
```

for example

```
@ServiceProvider(service = Simplifier.class)
```

The most important method in each module is `start`, defined in each of the interfaces. This method is called by the framework when the respective step of inference is being executed. It has always two parameters: the actual input data for the module, and a callback object to report to when this step is finished. We will look at both parameter now in more detail.

3.1.1 Module input

Each inference module takes the actual input data as the first parameter of its `start` method. The type of the argument differs based on the inference module.

Initial Grammar Generator takes an object of type *Input*. This class encapsulates all the input files in 3 collections of `File`: `documents`, `schemas` and `queries`. Enumerating these files provides IGG with access to all data it needs to create Initial Grammar.

Simplifier and *Schema Generator* take grammar, in other words a list of rules as input. In the first case, this grammar is the Initial Grammar, in the second case it is the simplified grammar.

3.1.2 Module output

Second parameter of each inference module's `start` method is a callback object. There are 3 callback interfaces defined in the `cz.cuni.mff.ksi.jinfer.base.interfaces.inference` package

- `IGGeneratorCallback`
- `SimplifierCallback`
- `SchemaGeneratorCallback`

Each callback interface naturally belongs to the similarly named module interface. As their respective interfaces, also callbacks define one crucial method: *finished*. Each inference module is responsible for invoking this method on the callback it got as a parameter after it has finished its work and has results to be passed on. Again, these 3 finished methods have different arguments based on the inference module.

`IGGeneratorCallback.finished()` and `SimplifierCallback.finished()` have a grammar (Initial Grammar in case of IGG) as their only argument.

`SchemaGeneratorCallback.finished` has two `Strings` as arguments: `schema` is the actual string representation of the resulting schema, `extension` is a file extension of the result (such as "dtd" or "xsd") which the framework will use when saving the result in a file.

3.1.3 Error handling

Because the run of each inference module is encapsulated in a `try-catch` block by the framework, it is safe to throw any exception out of the `start` method: it will get logged, presented to the user and inference will stop. However, if the module uses threads that could throw an exception, it is responsible for catching these exceptions and possibly re-throwing them in the thread where `start` runs.

3.1.4 Interruptions

User running the inference might change his mind and try to stop this. For this reason, modules have to check for this case in every time-consuming place such as long loops with the following code

```
for (forever) {
  if (Thread.interrupted()) {
    throw new InterruptedException();
  }
  doStuff();
}
```

3.1.5 Runner

The part of framework responsible for actually gathering user input, running all modules one after another and presenting the results is the `Runner` class in `cz.cuni.mff.ksi.jinfer.runner` package.

A new instance of `Runner` is constructed for each inference run. While being created, `Runner` loads the preferences for current project and looks up user-selected inference modules. Also, callback objects pointing back to methods in `Runner` are created. The inference process itself is then as follows

1. Selected IGG's `start` is encapsulated with error/interruption handling and executed, passing `Input` and first callback as parameters.
2. When IGG finishes, it invokes callback's `finish` method, passing the IG as parameter.
3. This in turn causes `Runner` to encapsulate and execute Simplifier's `start`, passing IG from the first callback and the second callback as parameters.
4. When Simplifier finishes, it invokes callback's `finish` method, passing the simplified grammar as parameter.
5. This again causes `Runner` to encapsulate and execute SchemaGen's `start`, passing the simplified grammar from the second callback and the third callback as parameters.
6. SchemaGen finishes and invokes the last callback's `finish`, passing the resulting schema and its extension as parameters.
7. `Runner` receives the resulting schema and based on preferences, saves it to a file, displays it, etc.

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.

[ext] `http://www.engr.mun.ca/~theo/Courses/fm/pub/context-free.pdf`.

[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.

[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`.