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 Generator*, *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 [**?**], [**?**]. 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<T>` with supplementing classes `RegexpInterval` and `RegexpType`. Each `Regexp<T>` instance has one of the enum `RegexpType` type:

- Lambda (λ) - 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<T>` and can be tested by calling methods `isLambda()`, `isToken()` etc.

Each `Regexp<T>` 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).

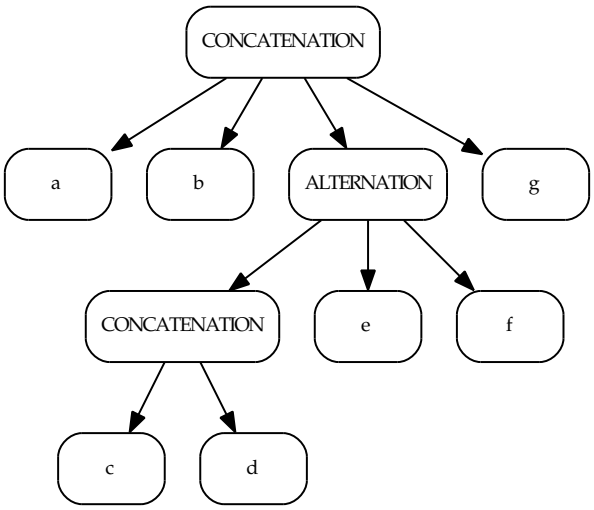


Figure 1: Example tree for regular expression $(a, b, ((c|d), e), f)$

Interval can be either bounded (you have to set both lower and upper bound integers), or unbounded (you have to set only lower bound). Testing interval value 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<T>` can represent regular expression over any alphabet. This is done by using java generics. 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<T>` 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
```

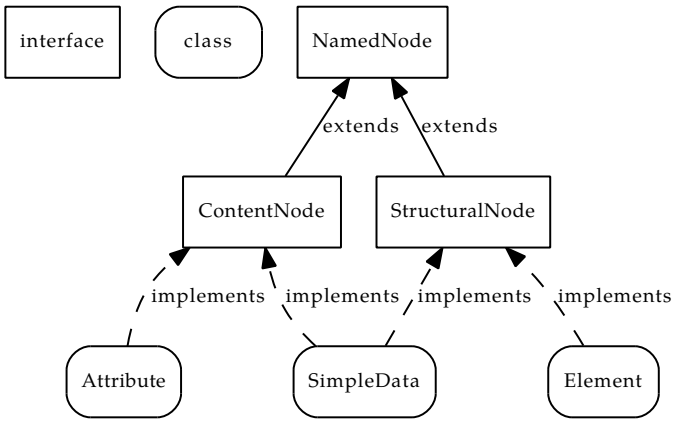


Figure 2: How should interfaces and classes for XML representation look like in theory

```
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 in section 3.1),
- `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.

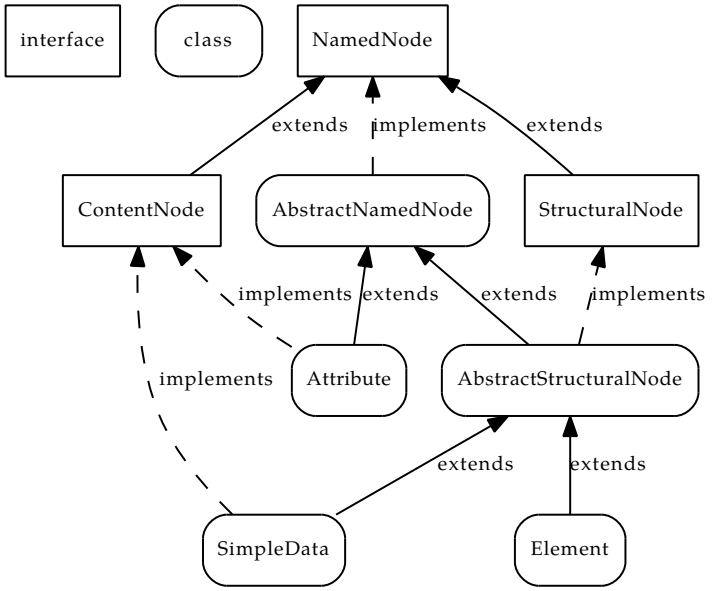


Figure 3: How are interfaces and classes for XML representation arranged in practice

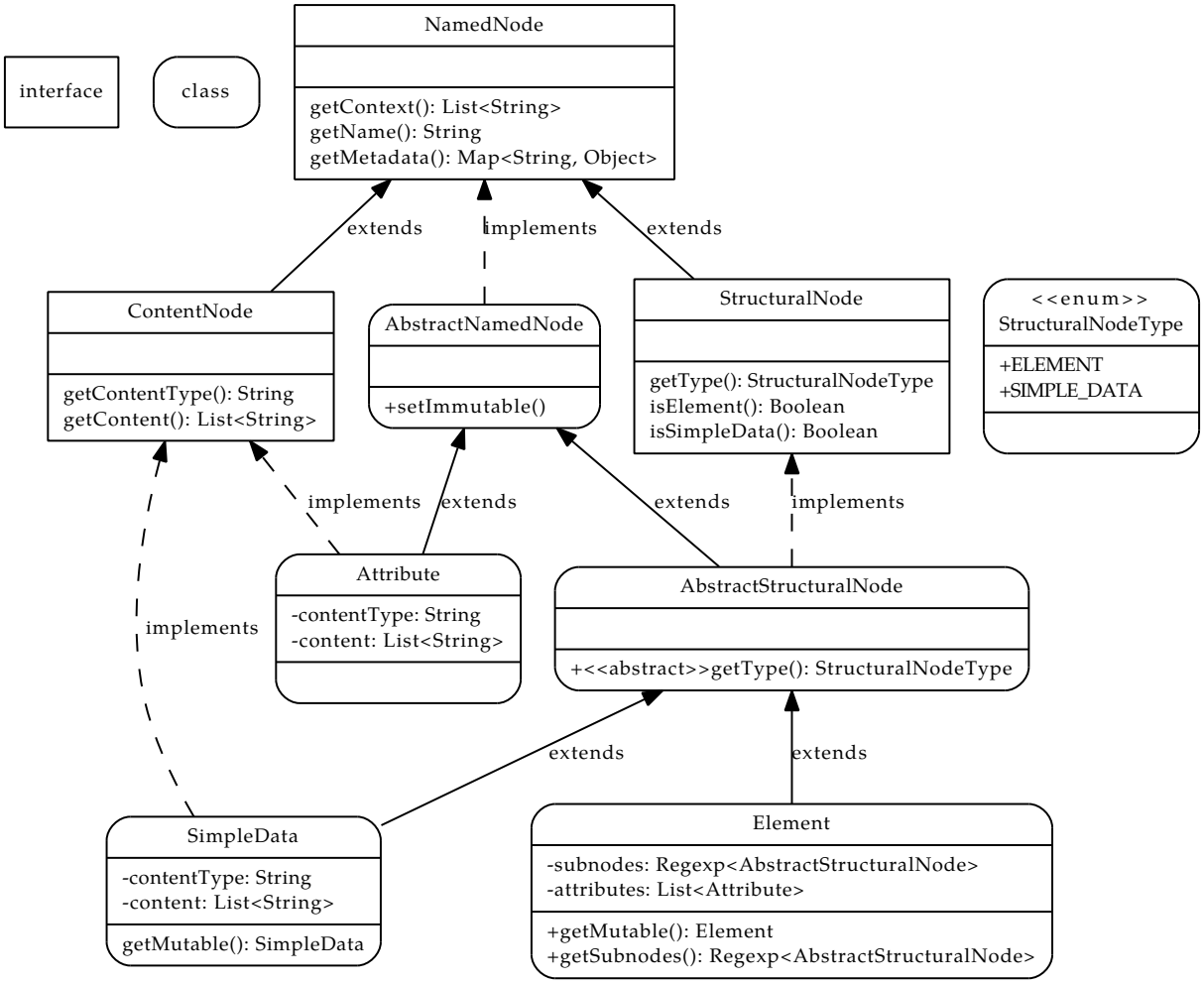


Figure 4: XML representing interfaces and classes in detail

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.

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 as well). Same `getMutable()` principles and good usage practises as for regexps, hold for theese classes.

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

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

Although in example we present whole document tree, input modules produce slightly different format (consisting of rules).

2.3 Rules and grammars

jInfer and its documentation uses extended context-free grammars[**?**]. *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 (even with no children, see fig. 5 again). Initial Grammar is produced by *Initial Grammar Generator*.

2.4 Nondeterministic Finite Automaton

We recommend skipping this section in first read as it describes advanced features. 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 `T` the java type of symbol. Take care of symbol class and its `equals()` implementation. Automaton uses `equals()` to compare symbols on transitions (when building prefix-tree automaton and when merging states). If you are using strings, you're just fine, but with complicated objects, either take care, that equivalent objects are properly tested in `equals()`. Or (maybe faster) solution is to cluster objects into classes of equivalence before inserting them into automaton. Then give automaton only cluster representant object (which will use java `Object.equals()` with reference comparison) for each object encountered.

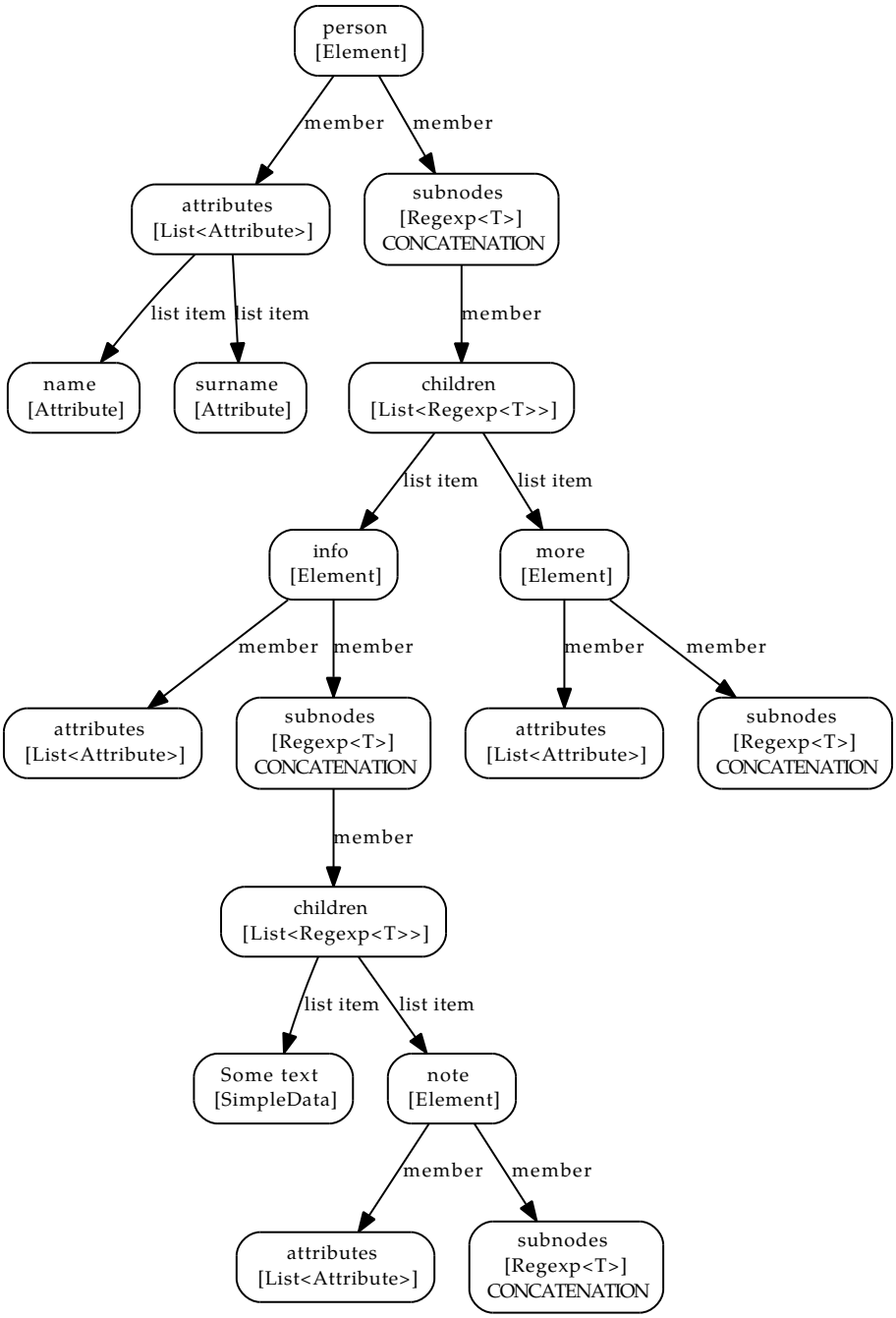
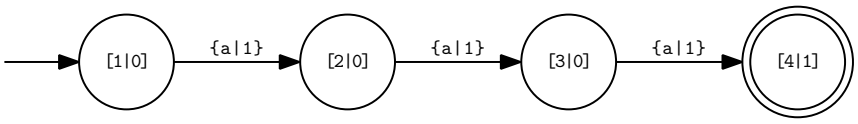
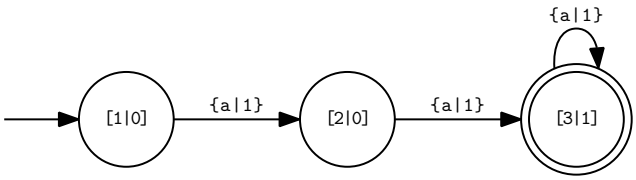


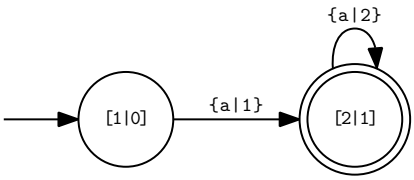
Figure 5: XML document representation



(a) Before merge.



(b) After states 3 and 4 merged.



(c) After states 2 and 4 merged.

Figure 6: Sample automaton.

State Let's begin by smallest of the classes, the `State<T>`. It represents automaton state, it has two integer members: `name` and `finalCount`. Name clearly serves as name of state in visualization and to string conversion. Final count is for representing whether the state is final in automaton. The field is not true/false but integral to help algorithms use stastistics over automatons (how many times in XML input is this state final?).

Step Next we have `Step<T>` which stands for automaton transition. It has its `source` and `destination` states references. Symbol accepted by using this transition is stored in `acceptsSymbol` member of generic type `T`. And finally member `useCount`, which is integer stating how many times the transition was used when constructing prefix tree automaton from input data. Simplifying algorithm can use this number for statistic purposes.

Automaton `Automaton<T>` class puts theese together into nondeterministic finite automaton. It has reference to `initialState`, it has `newStateName` integer value to assure unique state names inside one automaton (incrementing every time new state is created). We use two maps to implement transition function (δ-function). One map of type `Map<State<T>, Set<Step<T>>>` called `delta` represents mapping from state into set of all outgoing transitions from state. Second is just reversed map, called `reverseDelta`, which holds all incoming transitions into state (for better perfomance). There exist only one instance of each step from one state to another. That instance is referenced in delta map (on place of source state), and in reverse delta map (on place of destination state). Loops are no speciality, just source = destination.

Automaton supports creating of automaton as a copy of another one (not reference copy, but deep copy expect of symbols), this can be used when searching solution space to create more versions of automaton to edit. We implemented building of prefix-tree automaton (PTA) in `buildPTAOnSymbol()` method. Create empty automaton and then call this method for every input string of language. You will get PTA with useCounts and finalCounts set properly on steps/states.

The biggest thing we offer to scientists is state merging by simply calling `mergeStates(state1, state2)` method. Method merges second state given into first one (or an overloaded version - all states in list into first one in list). All {in | out}-transitions are redirected properly (or discarted as needed). Variables useCount and finalCount are updated to sums of values from merged transitions/states. One can ask for merging states, that are merged out from automaton already. Lets take example automaton of fig. 6(a) (states are labeled [name | finalCount], steps by {symbol | useCount}). One asks to merge states 3 and 4. State 3 then becomes final state with loop and state 4 disappears (fig. 6(b)). If then one ask to merge states 2 and 4, automaton properly handles situation by knowing, that old state 4, was merged into state 3, and merges states 2 and 3 (see fig. 6(c)). This can be useful in $k, h-context$ and $s, k-strings$ implementations (find all context/strings, then supply list of states to merge and don't bother with state names updates).

AutomatonCloner Class `AutomatonCloner<A, B>` has one overloaded method `convertAutomaton()`. First version accepts automaton and class implementing `AutomatonClonerSymbolConverter<A, B>` interface, and returns new automaton with same structure, but with symbols on transitions from alphabet of java type `B`. Second version takes two automatons, second one has to be empty (without initial state created), and symbol converter. It fills in second automaton to have same structure as first one, but with symbols of type `B`.

To fulfill symbol conversion, one have to provide imlementation of `AutomatonClonerSymbolConverter<A, B>` interface. Interface has one method:

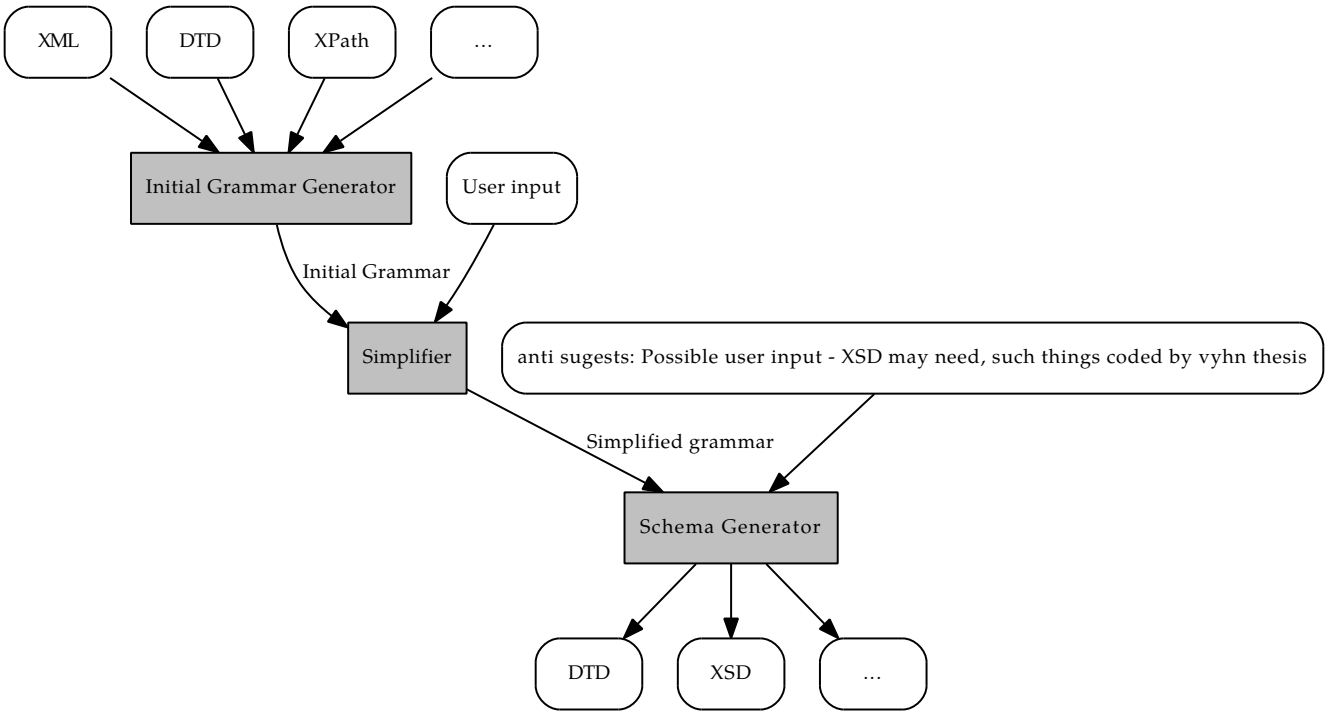


Figure 7: High-level view of the inference process

```
B convertSymbol(final A symbol);
```

Its purpose is to give mapping from symbol over one domain to new symbols. Implementation have to be `equals` consistent, if in first automaton are two transitions with symbols `a`, `b` such that `a.equals(b) == true`, convertor has to produce new symbols that are equal.

3 Inference process

The process by which jInfer infers the resulting schema from various inputs (inference process) is summarized by fig. 7. 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

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

   will translate into the following IG rules

   $$person \quad \rightarrow \quad info, more, more, more, more$$

$$
\begin{aligned}
info &\rightarrow simple\_data, note \\
note &\rightarrow empty\_concatenation \\
more &\rightarrow empty\_concatenation \\
person &\rightarrow more, more, more \\
more &\rightarrow empty\_concatenation \\
more &\rightarrow empty\_concatenation \\
more &\rightarrow empty\_concatenation
\end{aligned}
$$

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 (exactly one rule for each element). 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 export step.
   For example, previous rules for element person could be simplified to a single rule

   $$person \rightarrow info?, more\{1, 3\}$$

   Rules for elements info, more and note after simplification will be:

   $$
   \begin{aligned}
   info &\rightarrow simple\_data, note \\
   note &\rightarrow \lambda \\
   more &\rightarrow \lambda
   \end{aligned}
   $$

   Note the lambda regular expressions for note and more. In Initial Grammar, all regexps are concatenations (even empty), but in simplified grammar, if element have to be empty in schema, it has to have lambda regular expresion as subnodes.

3. Schema export: done by the *Schema Generator (SchemaGen)* module, this is the process of actually creating the resulting schema file 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).
   For previous simplifier rules, the resulting DTD would be:

   ```
   <!ELEMENT person (info?, more, more?, more?)>
   <!ELEMENT info (#PCDATA | note)*>
   <!ELEMENT note EMPTY>
   <!ELEMENT more EMPTY>
   + attributes
   ```

   For element person, even when simplified grammar specifies its occurence to at least once, at most 3 times, as DTD has no such construct, export module have to do some magic. Situation is even worse for elements that contain simple data inside simplified rule. Only way to express mixed content in DTD is to use `(#PCDATA | note)*` construct. Even if regular expression is complicated, export module has to do this "flattening". Lets look on XSD output of same rules:

   ```
   <xs:element name="person">
     <xs:complexType>
       <xs:sequence>
         <xs:element name="info" minOccurs="0" maxOccurs="1">
           <xs:complexType mixed="true">
             <xs:sequence>
               <xs:element name="note" minOccurs="1" maxOccurs="1">
                 <xs:complexType>
                 </xs:complexType>
               </xs:element>
             </xs:sequence>
           </xs:complexType>
   ```

```
      </xs:element>
      <xs:element name="more" minOccurs="1" maxOccurs="3"/>
        <xs:complexType>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
+attributes
```

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 it is possible to replace any and/or all of these modules with *something* else that does similar job. In practice, scientists will probably implement new *Simplifier* modules, while using our implementation of BasicIGG and export modules. This modular scheme is used inside our modules to divide them into submodules. It is possible to add new file format processing to import/export. One can extend automaton merging state algorithm of our implementation of [**?**] in TwoStepSimplifier module by replacing submodules.

3.1 Node metadata

To allow simple extensibility of rules, class `AbstractNamedNode` contains a string-addressed map called `metadata` that can contain arbitrary object values.

At the present, jInfer modules use the following metadata, all defined in `IGGUtils` class:

- `from.xml`, filled in by BasicIGG module: means that this rule was created (originates from) from a XML document.
- `from.schema`, filled in by BasicIGG and XSDImport modules: means that this rule was created (originates from) from a schema.
- `from.query`, filled in by BasicIGG and XSDImport modules: means that this rule was created from a query.
- `is.sentinel`, filled in by BasicIGG and XSDImport module: when importing XML document, one can build whole tree in memory (as in fig. 5) and then construct list of rules from it (thus saving memory). With schema import however, one not only doesn't know right side of rule in advance (solved by stack in XML import), but right side can be defined anywhere in source file. To save complicated loading of whole schema, searching and pairing elements thorough rules, nodes on right side of rule are created empty - holding only name of node. Fact, that this node has no more information than it's name and position on right side of rule is denominated by labeling the node as sentinel.
- `schema.data`, filled in by XSDImport module: TODO reseto Explain

All of these metadata are of set/not set character (using `Boolean.TRUE` as value, but their presence effectively mean true).

3.2 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 to be 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 parameters now in more detail.

3.2.1 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.

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

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.

Initial Grammar has two formats:

- Only concatenations. This is default format. All regular expressions coming from IGG have to be concatenations, even empty. When importing XML documents, this is just fine, as document is positive example from language we are trying to identify. But even when importing schemas, queries or other stuff that can contain more information, one has to convert it somehow to concatenations. For example, when importing DTD:

  ```
  <!ELEMENT person (info?, more, more?, more?)>
  <!ELEMENT info (#PCDATA | note)*>
  <!ELEMENT note EMPTY>
  <!ELEMENT more EMPTY>
  ```

  Elements note and more are empty, but we cannot use lambda regexp. Import will use empty concatenation instead. For element person, import module have to generate some positive examples of language generated by regular expression `(info?, more, more?, more?)`. This generating is basicly done by calling `cz.cuni.mff.ksi.jinfer.l` interface method `expand()`. One can implement own intelligent expander, or use our provided `cz.cuni.mff.ksi.jinfer.`

- Complex regular expressions. Simplifier module running in current inference can indicate support for this format by having capability called `can.handle.complex.regexps`. Even then, import module is not obliged to use it of course. But it is reasonable to do so, if imported file type gives more information than simple concatenations (when importing schemas). Initial Grammar can contain all regular expressions. If schema defines empty element, make its subnodes lambda regexp. If schema defines element content as `(a, b){2,7}(c | d)`, just construct that regexp. Empty concatenations are also allowed - as they are positive examples in XML document input. Its simplifier business to divide rules by import file formats (by using metadata described in **??**) and to handle empty concatenations from XML file import with complex regular expressions from other formats.

If you are writing import module, check if you can provide more information to simplifier from input file type by using complex format. This interface division allows us to plug in old methods (accepting only XML input) to the chain of inference, even with schema input files. It may be useful for method benchmarking too.

Simplified grammar has only one format and that is strict regular expression format. Empty concatenations, alternations or permutations are not allowed. If element should be exported as empty, use lambda regexp as its subnodes.

3.2.2 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.

3.2.3 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.

3.2.4 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();
}
```

3.2.5 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.

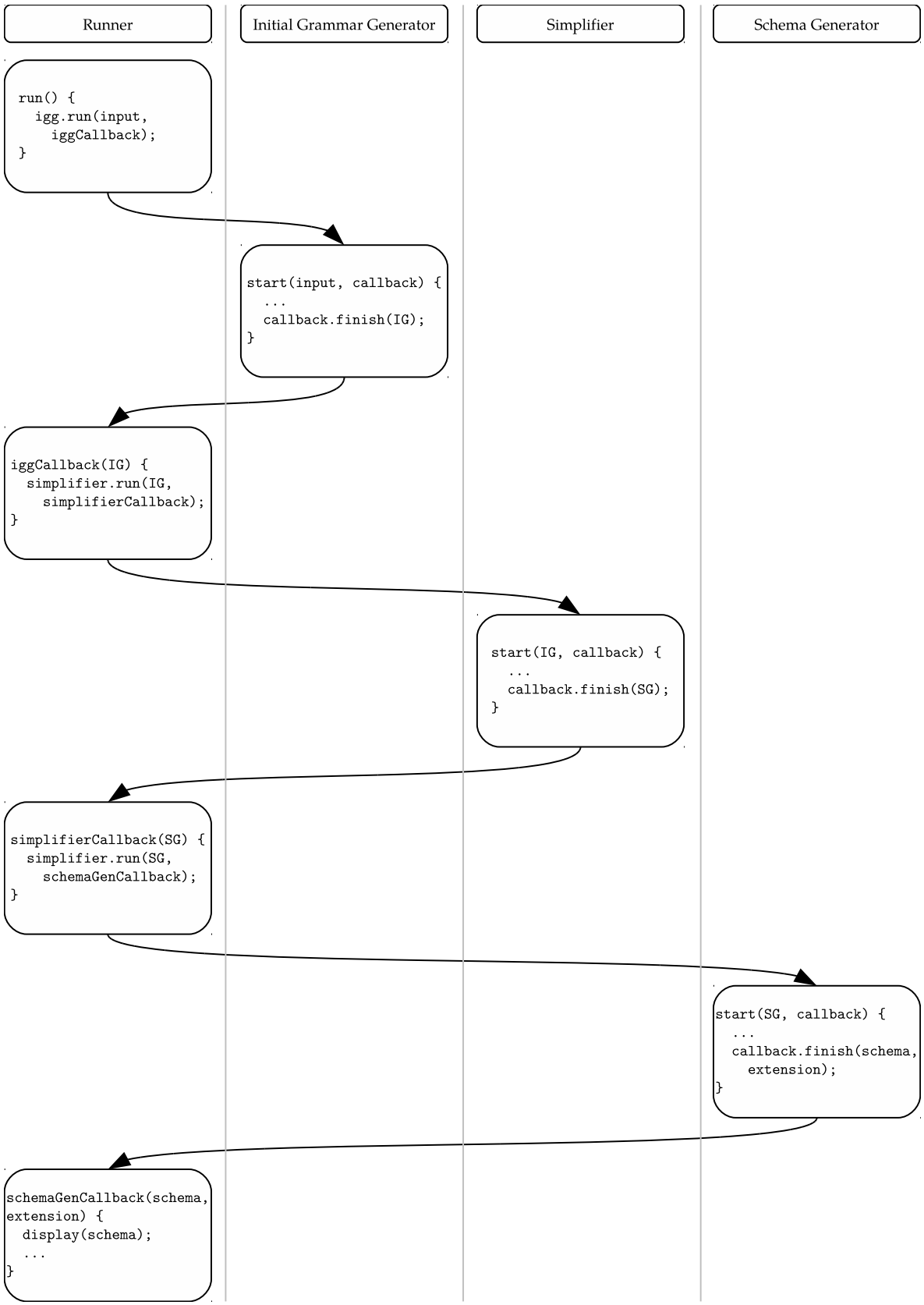


Figure 8: Runner

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.

3.3 Capabilities

From time to time, modules need a flow of certain information in completely opposite way to the usual inference: they need to know what the following module can or cannot do.

For this reason, inference modules extend the `Capabilities` interface defining a single method: `getCapabilities()`. This method returns a list of strings representing the abilities, or "capabilities" of this module.

An inference module can query the capabilities of the *next* module in the inference chain by invoking the `RunningProject.ge` method to obtain the next module encapsulated in the `Capabilities` interface.

At the moment, only one capability is defined to be communicated across modules, more specifically between *IGGenerator* and *Simplifier*. This is `Capabilities.CAN_HANDLE_COMPLEX_REGEXPS` meaning whether the `Simplifier` accepts more complicated regular expressions than simple concatenations of tokens on its input.

Example of Capabilities usage taken from *XSDImporter* module:

```
// if the next module cannot handle complex regexps, help it by expanding our result
if (!RunningProject.getNextModuleCaps().getCapabilities()
             .contains(Capabilities.CAN_HANDLE_COMPLEX_REGEXPS)) {
  // lookup expander
  final Expander expander = Lookup.getDefault().lookup(Expander.class);
  // return expanded
  return expander.expand(parser.getRules());
}
// return not expanded rules
return parser.getRules();
```

4 NetBeans Modules