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 be divided into 3 main views: the data view, the process view and the plaform view. The data view will commence by describing the data structures, namely representations of regular expressions and XML elements, attributes and simple data. The notions of a *rule* and a *grammar* will be explained, along with the finite state automata representation we use.

Afterwards the process of inference will be described, both from high-level and programmatic point of view. Finally, we will look at jInfer as at an NetBeans Platform application consisting of a number of modules performing their tasks and communicating together through a set of well-defined interfaces.

1 Package naming conventions

All packages start with `cz.cuni.mff.ksi.jinfer`. Afterwards is the short, normalized name of the NetBeans 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 view

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<T>` with supporting classes `RegexpInterval` and `RegexpType`. Each `Regexp<T>` instance has a type of `RegexpType` enum.

- 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 notation is $(a\&b\&c\&d)$.

Type of regexp is in `type` member in class `Regexp<T>` and can be tested by calling methods `isLambda()`, `isToken()`, `isConcatenation()`, `isAlternation()`, `isPermutation()`.

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.

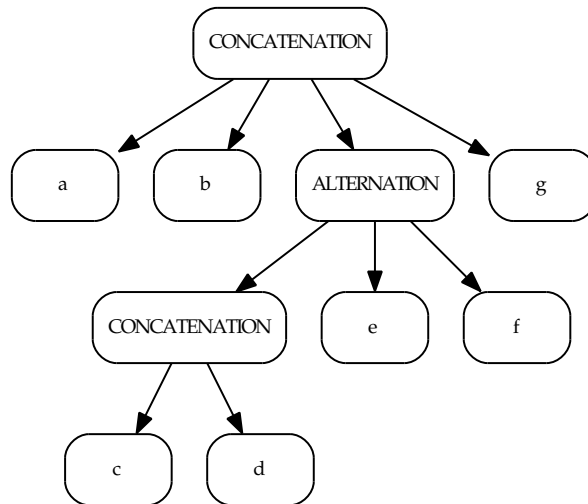- $a\{m, \}$ means at least m-times (unbounded).

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 this 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.

Using Java generics, `Regexp<T>` can represent regular expression over any alphabet. Only token regexps actually 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 in a member of `Regexp<T>` class called `children`, which is of type `List<Regexp<T>>`. List contains children of regexp: a node contains a list of child nodes in the tree view.

Regexp has to obey the following 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 constraints are checked by constructors, therefore 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. In special circumstances, one does not know future children of regexp at the time of creation. This occurs mainly in input modules: while 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 the members set. One has to fill in all properties carefully and call `setImmutable()` afterwards. 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
```
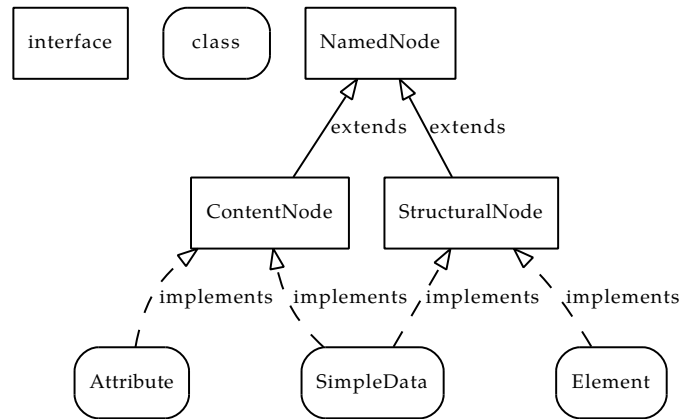
Figure 2: How should interfaces and classes for XML representation look like in theory

```
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 contains a useful method to obtain 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 encompasses elements, text nodes (characters inside elements) and attributes. For maximum generality, we decided to break apart these objects. We define three basic interfaces: `NamedNode`, `StructuralNode` and `ContentNode` (see package `cz.cuni.mff.ksi.jinfer.base.interfaces.nodes`).

The first stands for a bare node in XML document tree, it has its name and context within the tree (path from root). The latter two extend `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 laid out as in 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 3.2),

- `AbstractStructuralNode` which implements only the task of deciding if instance is `Element` or `SimpleData`.

On the one hand, we are interested in the XML structure - for this reason we use `isElement()` and `isSimpleData()` of `AbstractStructuralNode`. On the other hand, we want to access textual content and its type, for this reason we use `getContent()` and `getContentType()` of `ContentNode`.

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

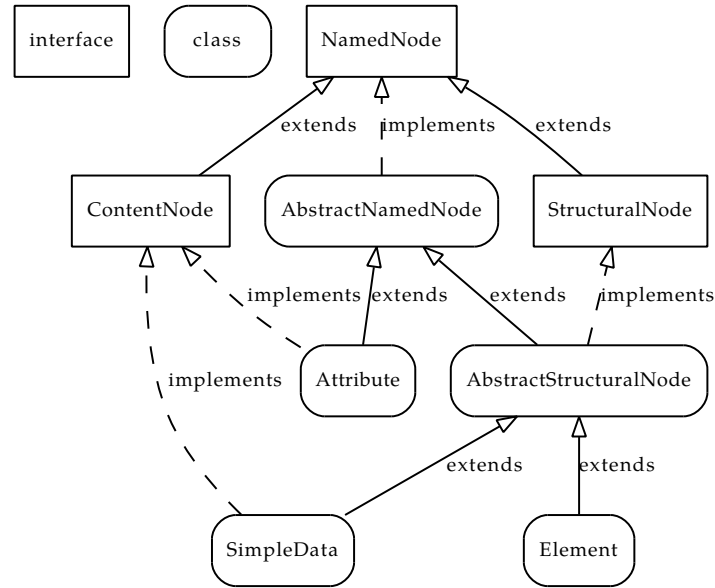Class `Element` has two important members.

3

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

- `Regexp<AbstractStructuralNode> subnodes` - representing the right side of a grammar rule,

- `List<Attribute> attributes` - representing all attributes in this element.

These two are filled in by import modules, processed further by inference (simplification) modules and finally exported by schema generator modules.

As in regular expressions, classes pertaining to 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). The same `getMutable()` principles and good usage practises hold for these classes.

Let's have an example: the following XML document would be represented as tree in fig. 5.

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

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

2.3   Rules and grammars

jInfer and its documentation use extended context-free grammars[Nor]. *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*.
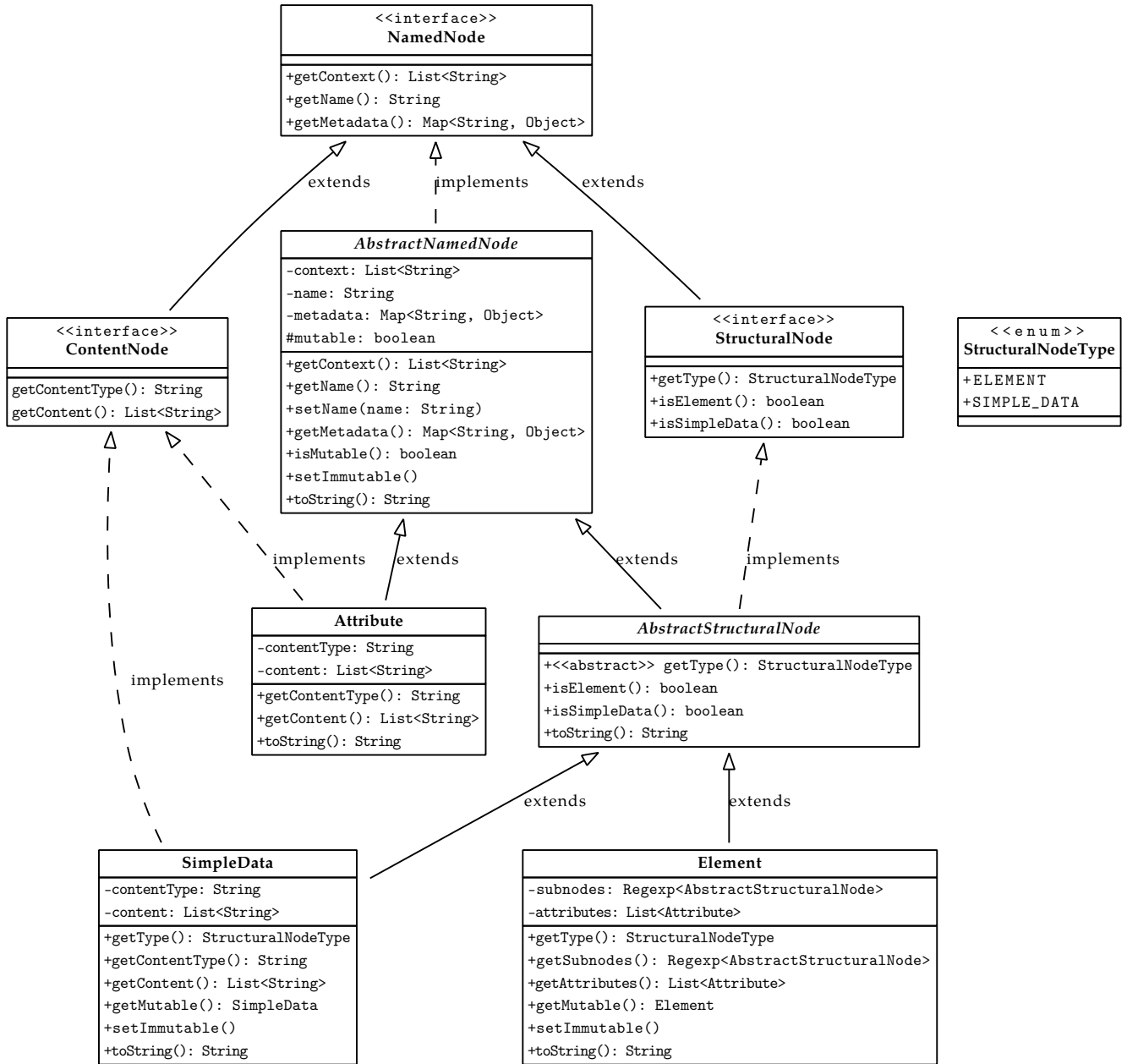
Figure 4: Class diagram for XML representing interfaces and classes
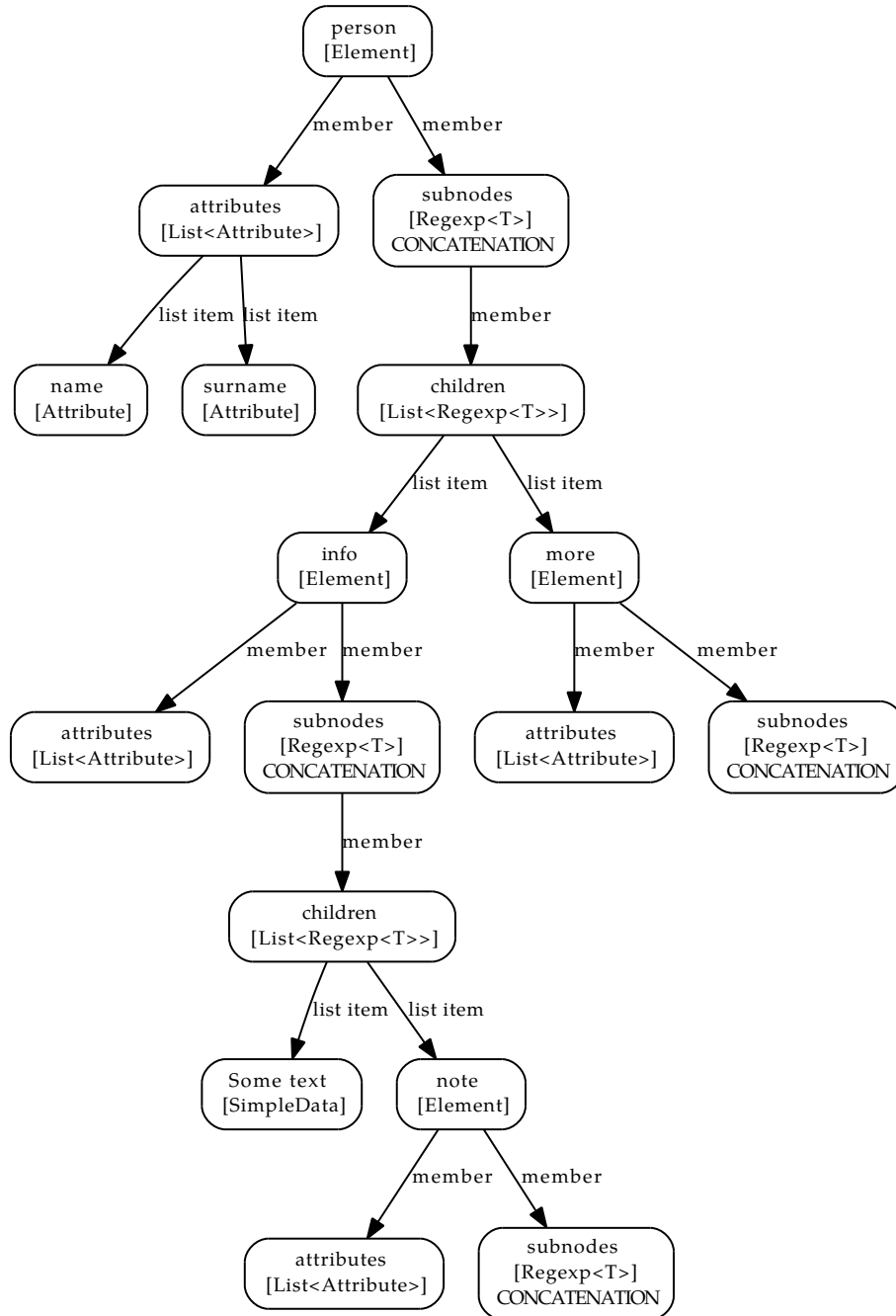
Figure 5: XML document representation

2.4 Nondeterministic Finite Automaton

We recommend skipping this section during the first reading 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 (see package `cz.cuni.mff.ksi.jinfer.base.automaton`).

- `Automaton<T>`

- `Step<T>`

- `State<T>`

- `AutomatonCloner<A, B>`

The implementation uses Java generics to represent symbol of alphabet. We denote `T` the Java type of that symbol.

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. With complicated objects, the first option is to make sure that equivalent objects are properly tested in `equals()`. Second, 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.

State   Let's begin with the smallest of 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 statistics 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 these together to form a 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 performance). There is 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, only source = destination.

Automaton supports creation 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 researchers 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 discarded as needed). Variables `useCount` and `finalCount` are updated to sums of values from merged transitions/states.

To be able to refer to states that have already been merged (and thus have a different name), we use a map of renamings. 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 asks 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).

(a) Before merge.

(b) After states 3 and 4 merged.
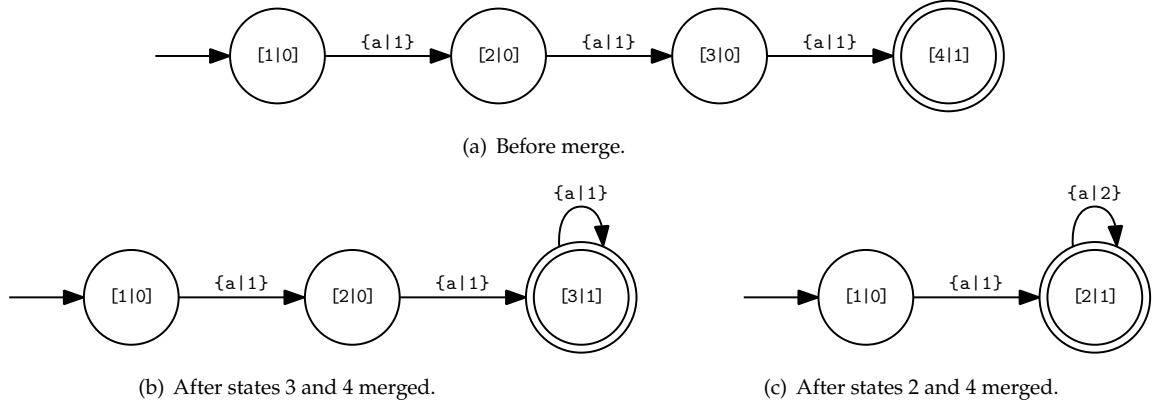
(c) After states 2 and 4 merged.

Figure 6: Sample automaton.

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 fulfil symbol conversion, one have to provide implementation of `AutomatonClonerSymbolConverter<A, B>` interface. Interface has one method.

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

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

3 Process view

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>
</person>
<more/>
<person>
  <more/>
  <more/>
  <more/>
</person>
```
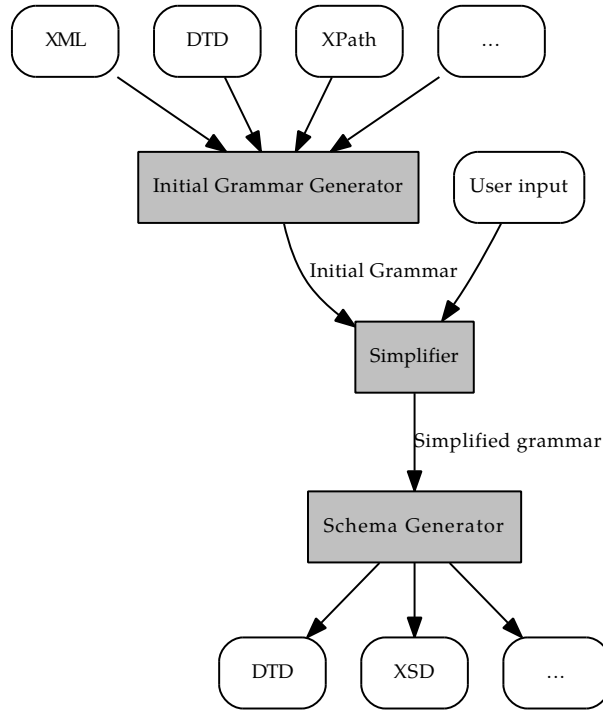
   will translate into the following IG rules

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

8

Figure 7: High-level view of the inference process

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

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".texmaker 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.

3.1 Programmatic view

From developer's point of view, inference modules are just properly annotated classes (see 4.1) 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 `Simplifier Impl`. 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.

3.1.1 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.

Simplifier and *Schema Generator* take grammar (see 2.3), 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 the default format. All regular expressions coming from IGG have to be concatenations, even if they are empty. When importing XML documents this is just fine, because a document is a positive example of language we are trying to identify. But while importing schemas, queries or other stuff that can contain more information, it might be necessary 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 has to generate some positive examples of language generated by regular expression (`info?, more, more?, more?`).

  This is basically done by calling `cz.cuni.mff.ksi.jinfer.base.interfaces.Expander` interface method `expand()`. One can implement own intelligent expander, or use our provided `cz.cuni.mff.ksi.jinfer.basic igg.expansion.ExpanderImpl`.

- **Complex regular expressions.** Simplifier module running in current inference can indicate support for this format by declaring capability `Capabilities.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. It is simplifier's responsibility to divide rules by import file formats (by using metadata described in 3.2) 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.

3.1.2 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.

12

```
Runner          Initial Grammar Generator          Simplifier          Schema Generator
```

```
run() {
  igg.run(input,
    iggCallback);
}
```

```
start(input, callback) {
  ...
  callback.finish(IG);
}
```

```
iggCallback(IG) {
  simplifier.run(IG,
    simplifierCallback);
}
```

```
start(IG, callback) {
  ...
  callback.finish(SG);
}
```

```
simplifierCallback(SG) {
  simplifier.run(SG,
    schemaGenCallback);
}
```

```
start(SG, callback) {
  ...
  callback.finish(schema,
    extension);
}
```

```
schemaGenCallback(schema,
extension) {
  display(schema);
  ...
}
```
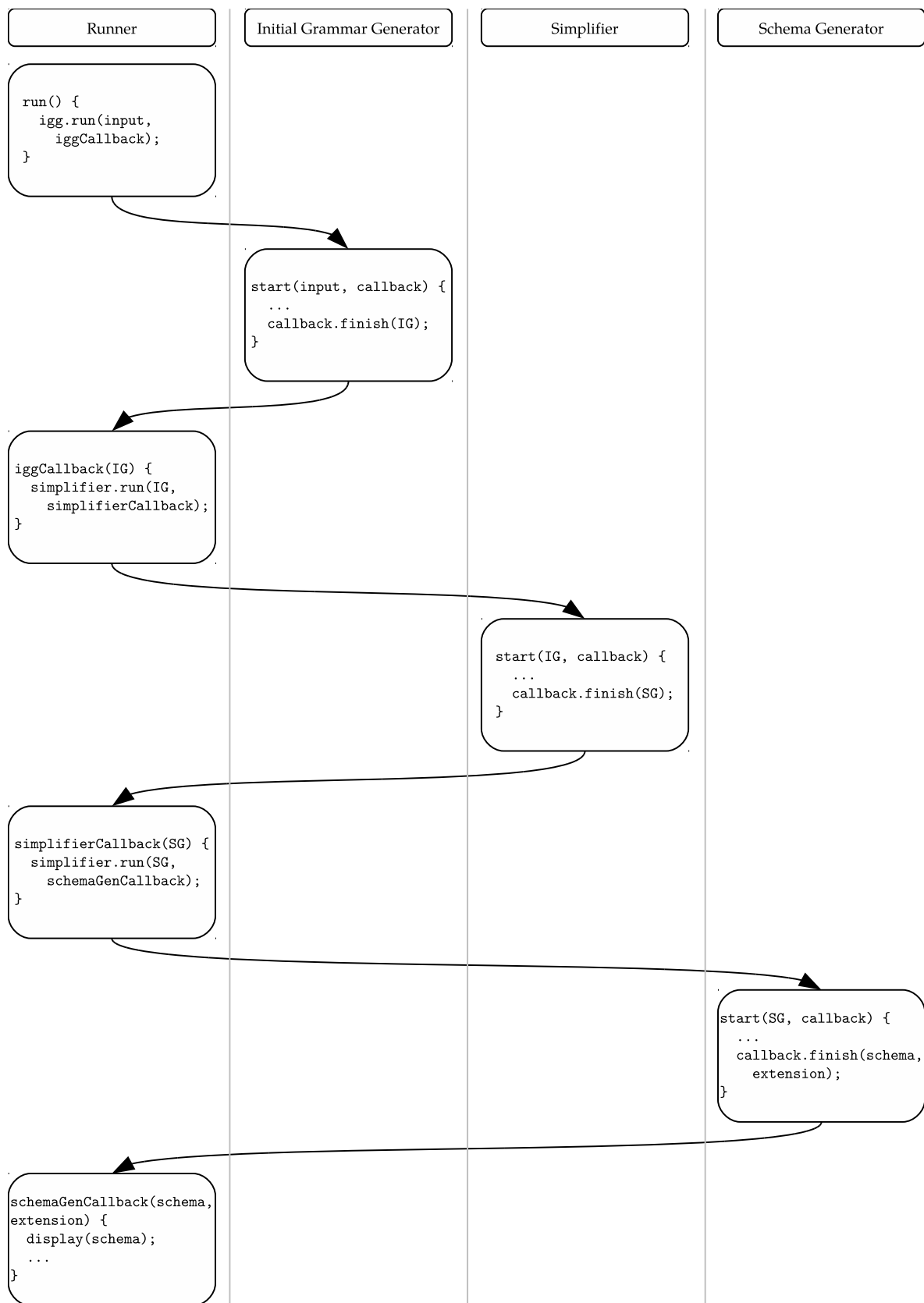
Figure 8: Runner

3.2 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 *XSDImporter* modules: means that this rule was created (originates from) from a schema.

- `from.query`, filled in by *BasicIGG* and *XSDImporter* modules: means that this rule was created from a query.

- `is.sentinel`, filled in by *BasicIGG* and *XSDImporter* modules: 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.

- `required`, filled in by *BasicIGG* and *XSDImporter* modules: indicates that the attribute is required.

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

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.getNextModuleCaps()
```

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
  Expander expander = Lookup.getDefault().lookup(Expander.class);
  // return expanded
  return expander.expand(parser.getRules());
}
// return not expanded rules
return parser.getRules();
```

4 Platform view

jInfer, being a NetBeans Platform application, is divided into multiple *NetBeans Modules*. It is very important to understand that this division is in principle orthogonal to that naturally sketched in the previous chapters - theoretically, the whole jInfer could be contained in a single NB module, because the framework uses Lookups to locate logical
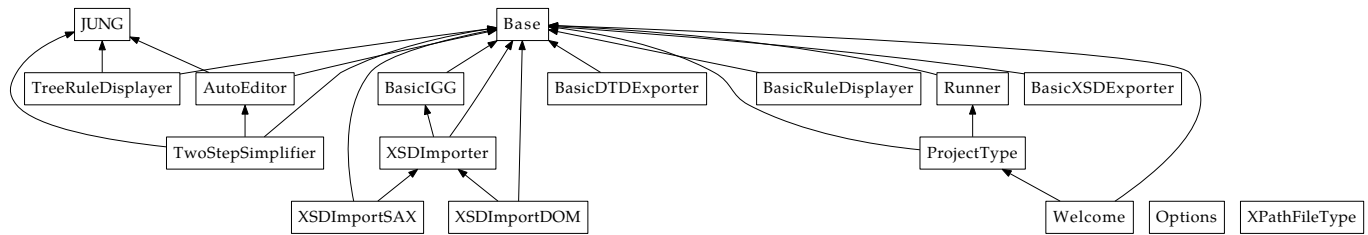
Figure 9: Module dependencies

modules. However, for the sake of organization, jInfer is split into several modules and most of the time there is a strong correlation between logical units of jInfer and its NB modules.

A short overview of these modules and their functions follows. To get a deeper understanding of any of them, please refer to its documentation. To get deeper understanding of how NBP modules work, their versions, dependencies etc, refer to [mod].

Base modules:

- *jInfer*: not a NBP module in fact, but rather a module suite - the "root" NB project for the whole framework.

- *Base*: contains common data structures, interfaces and utility logic shared across all other modules.

Inference modules:

- *Runner*: as described in the section 3.1.5, this module encapsulates the inference process (including the fact that inference consists of 3 phases as described before). Most important class: `Runner`.

- *BasicIGG*: this module contains an extensible implementation of *IGGenerator*. Basic version handles all XML-typed input documents, DTD schemas and XPath queries. To find out more about the extensibility to handle other outputs, refer to *BasicIGG*'s documentation.

- *XSDImport*: demonstrating *BasicIGG*'s extensibility, this is a base module for XSD input.

- *XSDImportSAX*: XSD input handling using SAX parser.

- *XSDImportDOM*: XSD input handling using DOM parser.

- *TwoStepSimplifier*: a *Simplifier* implementation. Implements [Aho96] merging state algorithm with basic handling of attributes. Read the separated documentation of module.

- *BasicDTDExporter*: this is a *SchemaGenerator* implementation with output to DTD.

- *BasicXSDExporter*: this is a *SchemaGenerator* implementation with output to XSD.

Utility modules:

- *BasicRuleDisplayer*: module capable of displaying a grammar in a graphical way.

- *TreeRuleDisplayer*: more advanced version of the former using *JUNG*'s graphing capabilities.

- *AutoEditor*: encapsulation of an interactive finite state automaton editor.

- *JUNG*: wrapper module for the JUNG (see [jun]).

- *Options*: NBP specific: this module adds a jInfer category in NetBeans Options window.

- *ProjectType*: NBP specific: this module defines a jInfer project type, along with its input and output files, settings etc.

- *XPathFileType*: NBP specific: this module teaches NetBeans to recognize a `.xpath` file extension.

The dependencies between all the modules are summarized in figure 9. Figure 10 shows modules implementing the 3 main inference interfaces described in 3.
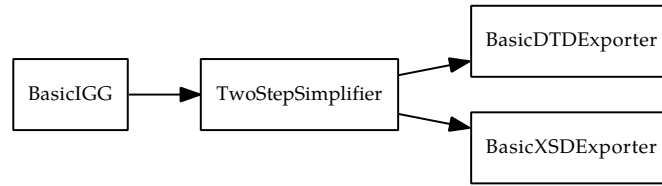
15

Figure 10: Inference modules in default jInfer

4.1 Lookups

Lookups are a part of NBP API, a way to achieve late-binding between code based on this platform. An important notion is the one of a *service provider*. This is a class implementing an interface, annotated in the following way (shown on an example from *BasicDTDExporter*).

```
@ServiceProvider(service = SchemaGenerator.class)
public class SchemaGeneratorImpl implements SchemaGenerator {
  ...
```

The `@ServiceProvider` annotation tells NBP that `SchemaGeneratorImpl` is to be registered among all providers of the `SchemaGenerator` service (or interface).

To look up a service provider, or all of them, one can use the *Lookups* API in the following way.

```
Collection<SchemaGenerator> generators = Lookup.getDefault().lookupAll(SchemaGenerator.class);

for (SchemaGenerator g : generators) {
  System.out.println(g.getName());
}
```

Call to `Lookup.getDefault().lookupAll()` will return a collection of all service providers registered to be providing the `SchemaGenerator` interface. This collection can be walked in some way, and on one or more of service providers methods can be called. We didn't have to actually bind providers to logic that uses their services, NB Platform took care of this for us.

See [loo] as the official documentation to lookup API.

4.2 Factory pattern

At this point it is important to understand that when using lookups this way, all service providers are kept as singletons. We ilustrate this behaviour on 11(a), user runs inference first time, classes are looked up and used. When user clicks run button again, and same instances of classes are returned by lookups.

In case a provider needs to change its state during providing the service (and needs to be "fresh" each time it is retrieved), there are two solutions:

- force each service providing class (submodule), to implement some sort of cleanup method, that would restart it into fresh state and make it ready to use by another inference run,

- or create new instances of stateful classes in each inference run and make service providing classes (those which NetBeans return by lookups) factories.

We decided for the latter approach (as ilustrated on fig. 11(b) It enables us not only to produce fresh class instance in each inference run, but also factory classes implement obligatory module methods such as `getName`, `getDescription` and so on, which are really same in each inference run. Each submodule is then defined by (at least) two interfaces. One is the factory interface, like this one (example from *TwoStepSimplifier*:
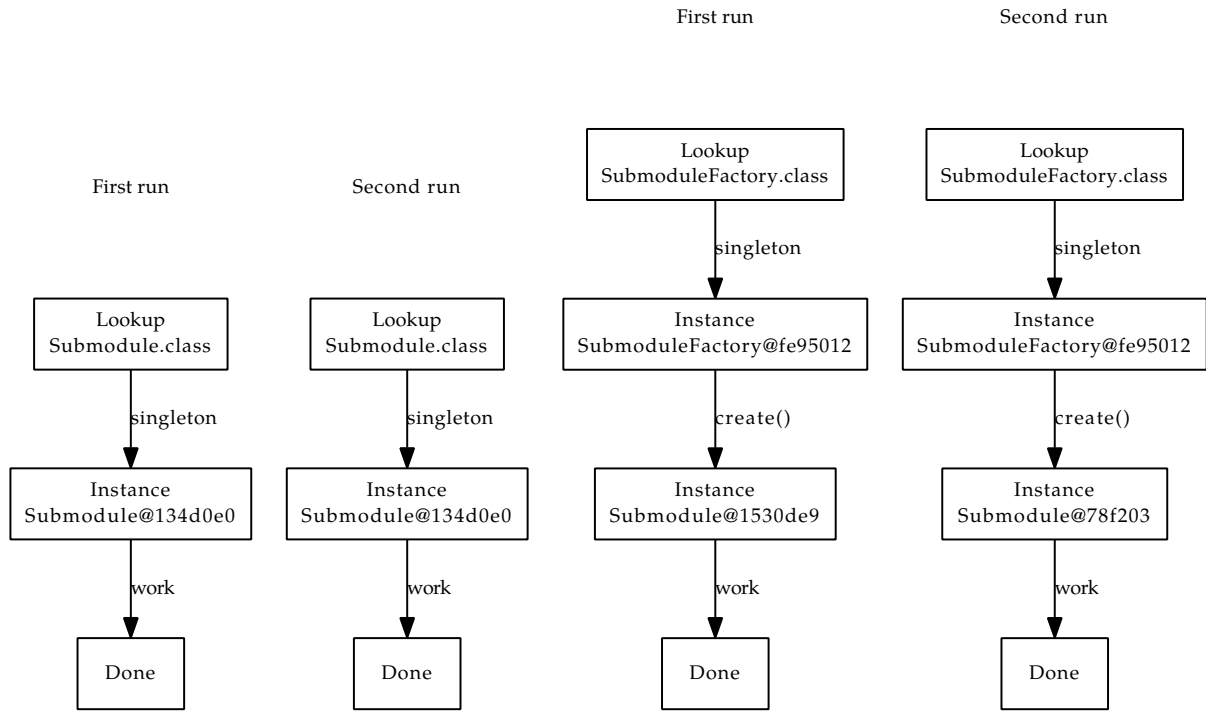
```
public interface AutomatonSimplifierFactory extends
                  NamedModule, Capabilities, UserModuleDescription {
  <T> AutomatonSimplifier<T> create();
}
```

Second is worker interface, in the example it is `AutomatonSimplifier<T>` interface.

This pattern is used thorough jInfer modules to divide module into submodules.

16

First run	Second run

Lookup Submodule.class	Lookup Submodule.class

singleton

Instance Submodule@134d0e0	Instance Submodule@134d0e0

work

Done	Done

(a) Same instance of class returned by NetBeans in two successive inference runs

First run	Second run

Lookup SubmoduleFactory.class	Lookup SubmoduleFactory.class

singleton

Instance SubmoduleFactory@fe95012	Instance SubmoduleFactory@fe95012

create()

Instance Submodule@1530de9	Instance Submodule@78f203

work

Done	Done

(b) Factory solution of singleton lookup class problem

4.3 Tokens

An important topic related to NB modules and their dependencies is a notion of *tokens*. Imagine a situation where module *M* needs *someone* else to provide a certain service *S*, but doesn't care who will that be. It cannot declare a direct dependency on a module containing this service provider, because it might not know such a module. Instead, *M* declares a *required token*. Any module providing *S* will declare a *provided token*, which is a simple string (usually fully qualified Java name of the interface representing *S*). Later, NBP will make sure that *M* can be installed in the platform only if there is at least one module providing that specific token.

jInfer uses tokens in the following way: *Runner*, as the module encapsulating the inference process, and *ProjectType* as an encapsulation of jInfer projects both require the following 3 tokens.

- `cz.cuni.mff.ksi.jinfer.base.interfaces.inference.IGGenerator`

- `cz.cuni.mff.ksi.jinfer.base.interfaces.inference.SchemaGenerator`

- `cz.cuni.mff.ksi.jinfer.base.interfaces.inference.Simplifier`

All inference modules declare their respective token as provided. This way it is ensured that jInfer will run only if there is at least one implementation of each of the inference modules, regardless of the implementation itself.

Another use of tokens is `cz.cuni.mff.ksi.jinfer.base.interfaces.RuleDisplayer` representing a module providing a rule displayer. This token is provided by *BasicRuleDisplayer* and *TreeRuleDisplayer*, requested by *BasicIGG* and *TwoStepSimplifier*.

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.

[BNST06] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Karl Tuyls. Inference of concise dtds from xml data. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 115–126. VLDB Endowment, 2006.

[BNV07]   Geert Jan Bex, Frank Neven, and Stijn Vansummeren. Inferring xml schema definitions from xml data. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 998–1009. VLDB Endowment, 2007.

[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`.

[net]   Todo. `http://netbeans.org`.

[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`.