jInfer TwoStep simplifier design and implementation

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, scientist willing to implement own inference methods

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

1 Using of factory pattern

TwoStep simplifier is divided into two submodules, which are then divided into submodules and so on. Each submodule is simply class, properly anotated. User selects proper classes - submodules in chain to work on inference. Classes are then, in runtime looked up by using NetBeans lookup mechanism. But NetBeans holds one instance of each class, that may be looked upon. When user runs inference first time, classes are looked up and used. Maybe class members are initialised, instances are in some state after inference completes. Then, user clicks run button again, and same instances of classes are returned by lookups. Ooops, theese are not freshly created instances, using them as they are may cause harm. And problem arises - force each class (submodule) that is being looked up in inference process, 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 theese classes in each inference run and make singleton classes (those which NetBeans return by lookups) their factory classes?

We decided to use second approach. It enables us not only to produce fresh 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:

```
public interface AutomatonSimplifierFactory extends NamedModule, Capabilities, UserModuleDescription {
  <T> AutomatonSimplifier<T> create();
}
```

It extends `NamedModule` with all module methods. It extends `Capabilities`, thus each module have to answer, if it has some capabilities. And, in TwoStep we often extend `UserModuleDescription`, which defines method for obtaining description of the module, comprehensive to user (it is displayed in properties panels). In some circumstances, it is useful to have method `create` generic. The automaton simplifier works with automaton, which itself is generic too. But simplifying does not depend on type of symbol of automaton, so interface `AutomatonSimplifier<T>` is also generic as simplifier can simplify automaton of any symbol java type. Factory interface deals with this by defining create method generic too.

Second interface is the one, which is returned by `create` method in factory. That is the interface, which defines real work cycle with submodule (in example it is `AutomatonSimplifier<T>`). We call this interface the worker interface of submodule. Usage of this factory pattern follows the routine:

```
final Properties p = RunningProject.getActiveProjectProps(getName());

AutomatonSimplifierFactory f = ModuleSelectionHelper.lookupImpl(AutomatonSimplifierFactory.class,
   p.getProperty(PROPERTIES_AUTOMATON_SIMPLIFIER));

AutomatonSimplifier<AbstractStructuralNode> autSmp = f.<AbstractStructuralNode>create();
...
give some work to autSmp
```

If our module has some submodules, we often implement lookups for submodules implementations in our own factory create method. Worker class receives factories of all submodules it needs as a constructor parameters.

Lets look on `AutomatonMergingStateFactory`, that is factory of module, which has AutomatonSimplifier as submodule. Its create method looks like this (shortened):

```
@Override
public ClusterProcessor<AbstractStructuralNode> create() {
  LOG.debug("Creating new ClusterProcessorAutomatonMergingState.");
  return new AutomatonMergingState(getAutomatonSimplifierFactory(), getRegexpAutomatonSimplifierFactory())
}
```

Methods `getAutomatonSimplifierFactory` and `getRegexpAutomatonSimplifierFactory` are analogical, we show you the former one:

```
private AutomatonSimplifierFactory getAutomatonSimplifierFactory() {
  final Properties p = RunningProject.getActiveProjectProps(getName());

  return ModuleSelectionHelper.lookupImpl(AutomatonSimplifierFactory.class,
           p.getProperty(PROPERTIES_AUTOMATON_SIMPLIFIER));
}
```

Cluster processor AutomatonMergingState then receives factories for AutomatonSimplifier and RegexpAutomaton-Sipmlifier submodules in its constructor. Cluster processor then may create as many instances of submodule classes as it needs (maybe simplifying more than one automaton). Thorough this document, we will mention only worker interface when describing submodules, since all factory interfaces are designed same way just described.

2 TwoStep module

TwoStep simplifier is inspired by [VMP08] design. Inference proceeds in two steps:

1. clustering of element instances into clusters of (probably) same elements
2. infering regular expression for each element from examples of element contents taken from all elements in cluster

Main routine of TwoStep simplifer does basically this:

```
    // 1. cluster elements
    final Clusterer<AbstractStructuralNode> clusterer= clustererFactory.create();
    clusterer.addAll(initialGrammar);
    clusterer.cluster();

    // 2. prepare emtpy final grammar
    final List<Element> finalGrammar= new LinkedList<Element>();

    // 3. process rules
    final ClusterProcessor<AbstractStructuralNode> processor= clusterProcessorFactory.create();
    for (Cluster<AbstractStructuralNode> cluster : clusterer.getClusters()) {
      final AbstractStructuralNode node =  processor.processCluster(clusterer, cluster.getMembers() - with
      // 4. add to rules
      finalGrammar.add(
              new Element(node.getContext(),
              node.getName(),
              node.getMetadata(),
              ((Element) node).getSubnodes(),
              attList);
    }
```

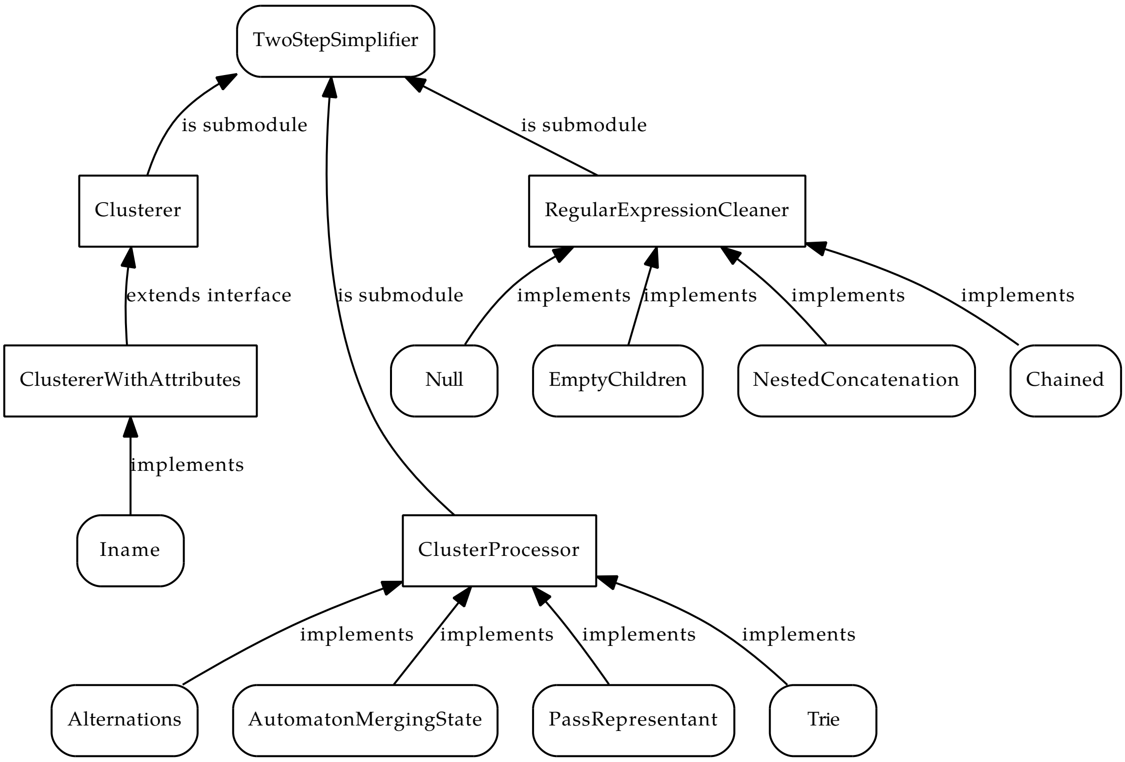

Figure 1: Submodules of TwoStep simplifier

Task of clustering is dedicated to *Clusterer* submodule, and task of infering regular expression for each cluster is dedicated to *ClusterProcessor* submodule. We will examine both of them. Modules currently in jInfer are drawn on fig. 1. We provide one *Clusterer* and four *ClusterProcessor* implementations. Each of those will be explained further in this document.

TODO sentinel processing

2.1 Clusterer submodule

Lets start by example, let input document(s) contain XML:

```
<person name="john" surname="smith">
  <info>
    Some text
    <note/>
  </info>
</person>
<inform>
  Another text
  <note/>
</inform>
<person>
    <information>
      Some text
      <note/>
    </information>
</person>
```

We examine info-like named elements. If we cluster elements sipmly by their name, we get one cluster with info element, another cluster with inform element and another one with information element. As [VMP08] suggests, we should consider element content in clustering process. If we take care of element content, elements info, inform and information would look same (in means of node tree inside them - text node and element note). Elements information and info have even same context (inside person element).

We implement clustering in `cz.cuni.mff.ksi.jinfer.twostep.clustering` package. One cluster is represented by class `Cluster<T>` with `T` as type of clustered items. This class simply holds java set of member of cluster and one of the references is held also in `representant` member.

We provide `Clusterer` interface for classes to implement. Its purpose is to cluster bunch of elements (rules) on input, into bunch of cluster class instances (cluters). It has methods `add()` and `addAll()` for adding items for clustering. Center is method `cluster()`, which does the clustering itself. As it may be time-consuming operation, method throws `InterruptedException`, implementation should take care of checking whether thread is user interrupted (see [KMS+]). After clustering, implementation should hold clusters in member, as it it will be further asked by calling method `T getRepresentantForItem(T item)`. Given item to this method, one can ask for representant of cluster, to which the item belongs. If no such cluster exists (item was not added for clustering before), we recommend you throwing an exception rather than returning `null`. Missing will probably indicate error in algorithm rather than normal workflow. One can pull clusters from clusterer by calling `getClusters()` method.

Basic work usage of clusterer is:

```
Clusterer <T> c = new MyContextClusterer<T>();
c.addAll(initialGrammar);
c.cluster();
...
c.getClusters();
or
c.getRepresentantForItem(x);
```

2.1.1 ClustererWithAttributes extended interface

Maybe you noticed that whole clusterer interface and cluster class are generic. They may be used as design pattern not only for clustering elements in inference process. To address clustering of elements in more detail, we created `ClustererWithAttributes<T, S>` interface, which extends `Clusterer<T>` interface. It add method `List<Cluster<S»` `getAttributeClusters(T representant)`, implying that each representant of type `T` (that is representant of some main cluster) has some "attribute" clusters associated with it. Theese clusters are obtained by calling `getAttributeClusters(x)`. Attribute clusters are of type `S`.

Finally, we use this scheme to implement clusterer `Iname<AbstractStructuralNode, Attribute>` class, which takes `AbstractStructuralNode` classes to cluster as main, and `Attribute` classes as attributes. For each cluster of elements, it creates list of clusters of attributes observed with all elements in main cluster.

All simple data are considered equal and given one dedicated cluster. But in future implementations, extensibility is ensured to cluster simple data to obtain content models of elements.

All sentinel elements are clustered too. TODO sentinels TODO attributes clustering

2.2 ClusterProcessor submodule

Cluster processor takes rules of one cluster of element and somehow obtains regular expression for that set of elements. It returns rule - element with name set to desired name of element in schema (not all elements in cluster have to have same name, if advanced clustering scheme is used, then processor has to choose right name for resulting element) and with subnodes set to regular expression infered. It process attributes of all elements in cluster to obtain meaningful schema attribute specification and theese attributes has to attach to resulting element.

Worker interface itself is defined as follows:

```
public interface ClusterProcessor<T> {
  T processCluster(
          final Clusterer<T> clusterer,
          final List<T> rules
          ) throws InterruptedException;
}
```

Maybe you are asking, why cluster processor is given the clusterer instance. Rules themself contain information about which elements to process, but clusterer has more information about the topic. Clusterer can tell you representant for any element in whole input (not only those elements in rules, but also those that may be on right side of rules), clusterer (if it is with attributes) has information about attributes <of each cluster.

We will now shortly describe each cluster processor implementation we've got.

2.2.1 PassRepresentant

Simple example to read. For each cluster return its representant as a rule to be in schema. This has nothing to do with infering grammar, it is just proof of submodules concept. Input documents are not valid against this odd grammar. Do not use this in practice, just read the code to understand the bare minimum needed to implement submodule.

2.2.2 Alternations

This processor simply gets all right sides from elements in cluster, puts them in one big list and creates alternation regular expression with this list as children. That is, it creates one big rule with alternation of every positive example observed. No generalization is done at all.

2.2.3 Trie

TODO vektor describe trie cluster processor

2.2.4 AutomatonMergingState

AutomatonMergingState is implementation of merging state algorithm on nondeterministic finite automaton. It creates prefix-tree automaton (PTA) from positive examples - right sides of rules given. Then it calls its submodule called *AutomatonSimplifier* to modify PTA to some generalized automaton by merging states.

Simplified automaton is then converted to an instance of `RegexpAutomaton` by using clone constructor in regexp automaton. Regexp automaton is automaton with regular expression as symbol on transitions. In automata theory, such automaton is called extended NFA. Clone constructing is done by converting each symbol in source automaton to regexp token with that symbol as content.

Regexp automaton is then passed into second submodule called *RegexpAutomatonSimplifier*. Its job is to derive regular expression from automaton, such that automaton and regular expression represents same language.

AutomatonMergingState has one more submodule, the *MergeConditionTester*, which is not called directly by AutomatonMergingState. It is at disposal for implementations of AutomatonSimplifier interface for testing, whether two states in automaton are equivalent and should be merged into one state.

AutomatonSimplifier implements solution searching logic in simplifying automaton by merging states. We implement greedy strategy in class `Greedy`. It simply asks given MergeConditionTester if it can merge any of automaton states and merges states until there are no states to be merged (on every pair of states asked, MergeConditionTester answers, they cannot be merged). One can implement ACO heuristics or MDL principle heuristics as AutomatonSimplifier submodule using MergeConditionTesters provided.

Whole submodule structure of AutomatonMergingState cluster processor is drawn on fig. 2. We have implemented *k,h-context* (see [Aho96]) state equivalence in class `KHContext`. We are using state removal method (see [HW07]) to convert regexp automaton into equivalent regular expression.

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.

[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.

[jun] Java universal network/graph framework. `http://jung.sourceforge.net/`.

[KMS$^+$] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. *jInfer Architecture*.

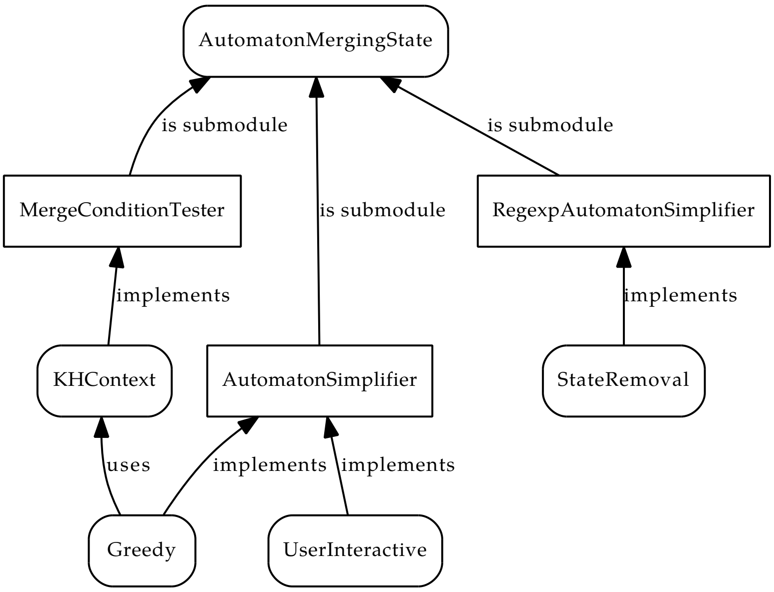

Figure 2: Submodules of AutomatonMergingState cluster processor

[mod] Module system api. `http://bits.netbeans.org/dev/javadoc/org-openide-modules/org/openide/modules/doc-files/api.html`.

[Nor] Theodore Norvell. A short introduction to regular expressions and context free grammars. `http://www.engr.mun.ca/~theo/Courses/fm/pub/context-free.pdf`.

[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`.

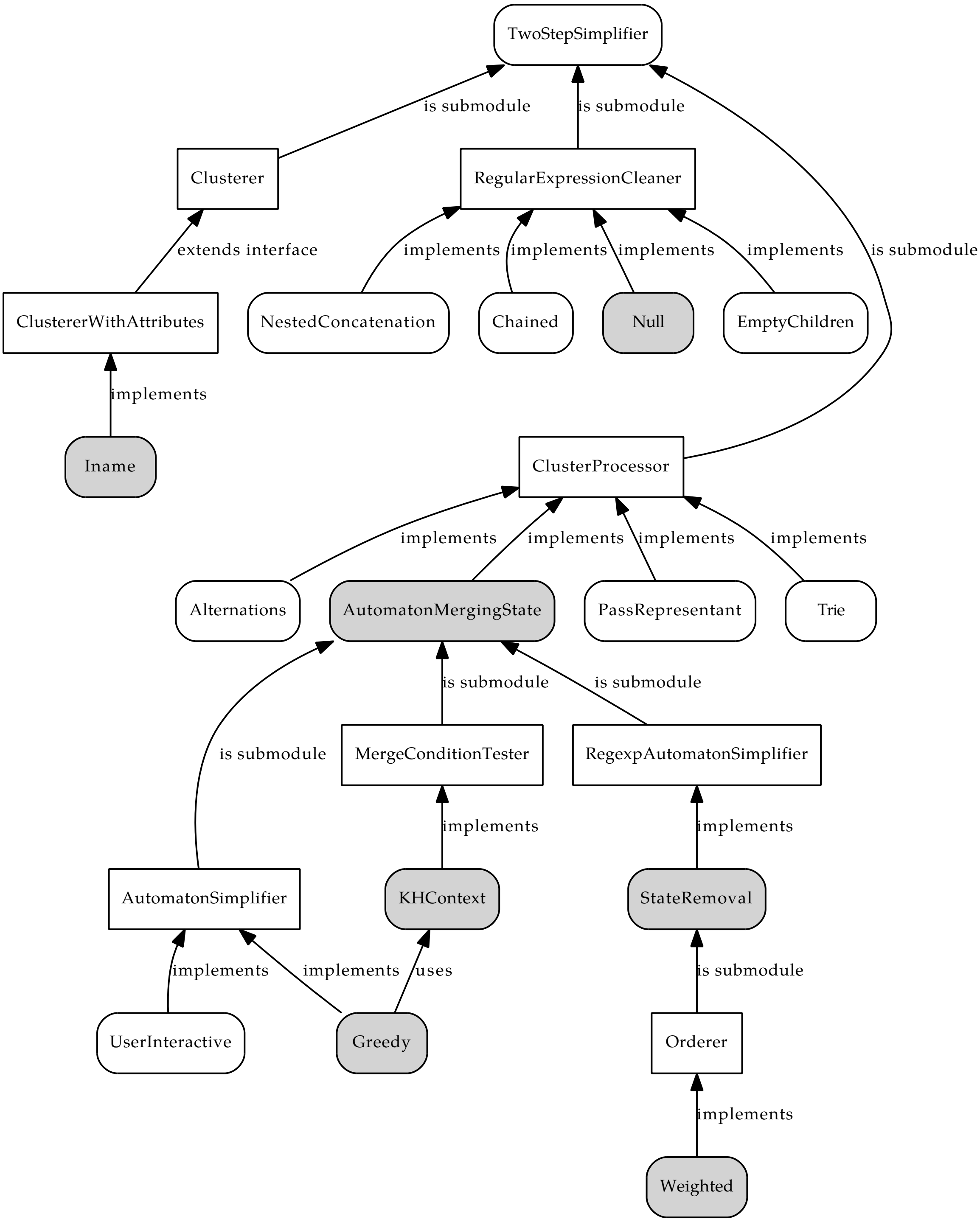

Figure 3: Submodules of TwoStep simplifier