

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

TwoStepSimplifier is divided into three submodules, which are then divided into submodules and so on. Each submodule is simply class, properly annotated. User selects proper classes - submodules to work in chain on inference. Classes are then, in runtime looked up by using NetBeans lookup mechanism (see [KMS⁺, p. 16]). 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. Oops, these are not freshly created instances, using them as they are may cause harm. The problem is illustrated on fig. 1.

There are basically two solutions:

- 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 these classes in each inference run and make singleton classes (those which NetBeans return by lookups) their factory classes.

We decided for the latter approach (as illustrated on fig. 2). 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:

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

It extends `NamedModule` with its modules describing methods:

```
String getName();
String getDisplayName();
String getModuleDescription();
```

It extends `Capabilities`, thus each module has to answer, if it has some capabilities. And, in *TwoStep* we often extend `UserModuleDescription`, which defines method:

```
String getUserModuleDescription();
```

It returns description of the module, comprehensive to user, which is then displayed in properties panels.

In some circumstances, it is useful to have method `create()` generic. The *AutomatonSimplifier* 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 java type symbol. Factory interface deals with this by defining `create()` method generic too.

Second interface is the one, which is returned by `create()` method. 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:

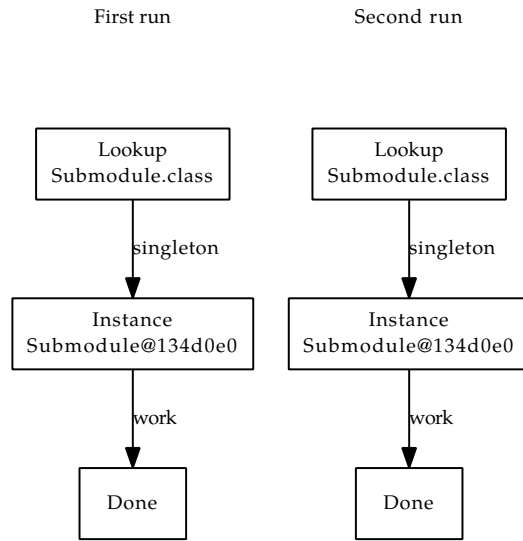


Figure 1: Same instance of class returned by NetBeans in two successive inference runs

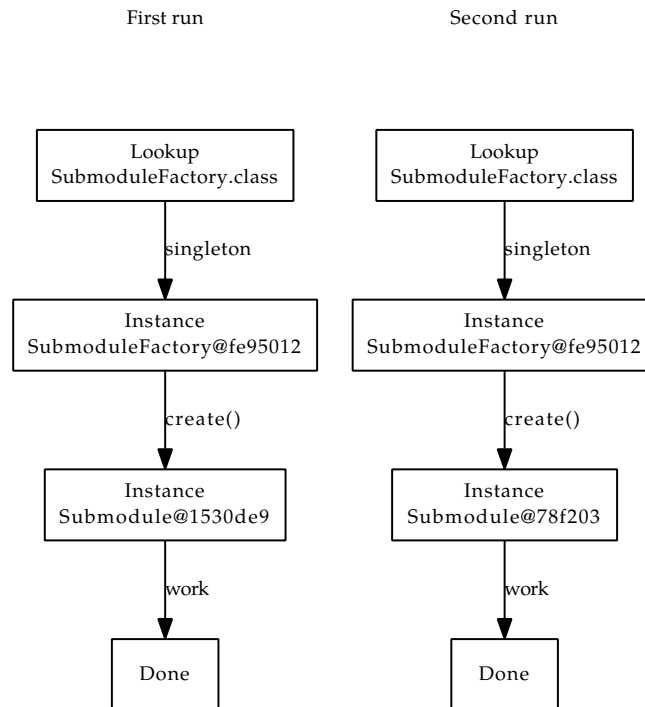


Figure 2: Factory solution of singleton lookup class problem

```

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 submodule 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(), getRegexAutomatonSimplifierFactory())
}

```

Methods getAutomatonSimplifierFactory and getRegexAutomatonSimplifierFactory 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 of AutomatonSimplifier and RegexAutomatonSimplifier 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. inferring regular expression for each element from examples of element contents taken from all elements in cluster

Main routine of TwoStep simplifier does basically this:

```

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

// 2. prepare empty 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() - without
// 4. add to rules

```

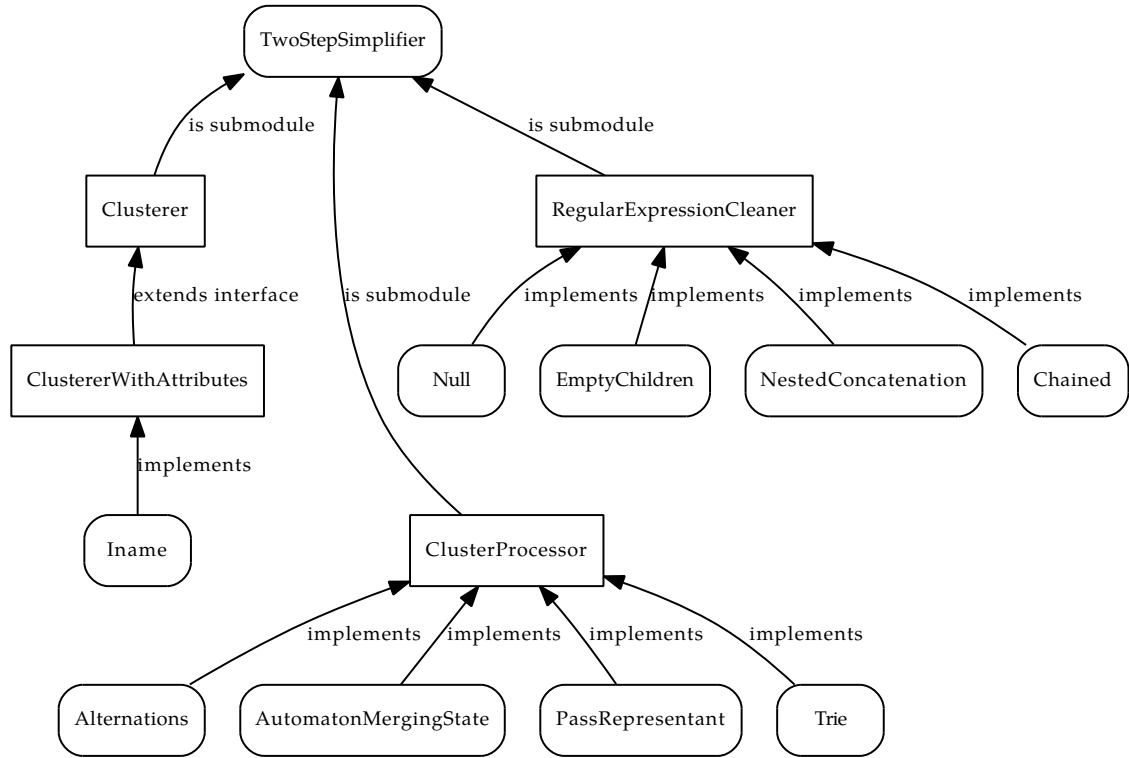


Figure 3: Submodules of TwoStep simplifier

```

finalGrammar.add(new Element(node.getContext(),
    node.getName(),
    node.getMetadata(),
    ((Element) node).getSubnodes(),
    attList);
}

```

Task of clustering is dedicated to *Clusterer* submodule, and task of inferring regular expression for each cluster is dedicated to *ClusterProcessor* submodule. We will examine both of them. There is also third submodule called *RegularExpressionCleaner* actually, its purpose is just to beatify output regular expressions, no inference logic is implemented there. Modules currently in jInfer are drawn on fig. 3. 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
  </info>
</person>
<inform>
  Another text
</inform>
<person>
  <information>

```

```

    Some text
    <note/>
  </information>
</person>

```

We examine info-like named elements. If we cluster elements simply 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 (clusters) on output. 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⁺, p. 12]). After clustering, implementation should hold clusters in member, as 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 item 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 adds 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. Attribute clusters are of type `S` and can be retrieved by calling `getAttributeClusters(x)`.

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 inferred. It process attributes of all elements in cluster to obtain meaningful schema attribute specification and these 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 themselves 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 inferring 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

This processor takes all rules in a cluster, treats them like strings and builds a prefix tree (a "trie") of them. More precisely, it takes the first rule and declares it to be a long branch (concatenation of tokens) in a newly created tree. After that, it adds the remaining rules one by one as branches like this: as long as it can follow an existing branch, it follows it. As soon as the newly added branch starts to differ, it "branches off" (creates an alternation at that point) the existing tree and hangs the rest of the newly added rule there. Repeating this process creates a prefix tree describing all the rules in the cluster.

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 *RegexAutomaton* by using clone constructor in *regex* automaton. *Regex* 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 *regex* token with that symbol as content.

Regex automaton is then passed into second submodule called *RegexAutomatonSimplifier*. 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. 4. We have implemented *k,h-context* (see [Aho96]) state equivalence in class *KHContext*, which is used by *Greedy* to test mergability of states by default configuration.

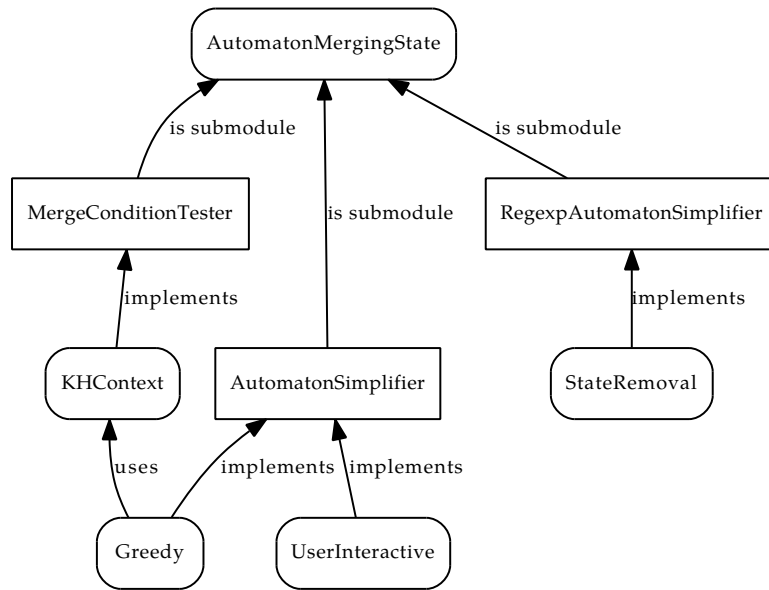


Figure 4: Submodules of AutomatonMergingState cluster processor

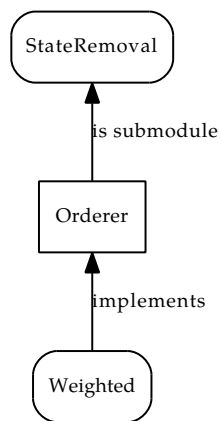


Figure 5: Modules of StateRemoval.

2.2.5 StateRemoval

We are using state removal method (see [HW07]) to convert regexp automaton into equivalent regular expression. This is implemented in `StateRemoval` class (on fig. 5). We defined one submodule of this class with interface called `Orderer`. It has only one method to implement: `getStateToRemove`. Given automaton it has to return reference to one state which should be removed from automaton at first. State removal calls this submodule and removes states given until there are only two states in automaton - `superInitial` and `superFinal` states, with exactly one transition. That transition has final regular expression on it as symbol, it is read and returned to `AutomatonMergingState`. We implement one orderer, called `Weighted`. It is simple heuristic - weights all states (weight = sum of in | out | loop-transition regular expression lengths) and returns state with lowest weight.

2.3 RegularExpressionCleaner module

Last we examine `RegularExpressionCleaner` interface. TODO anti cleaners

Whole `TwoStep` submodules structure is on fig. 6.

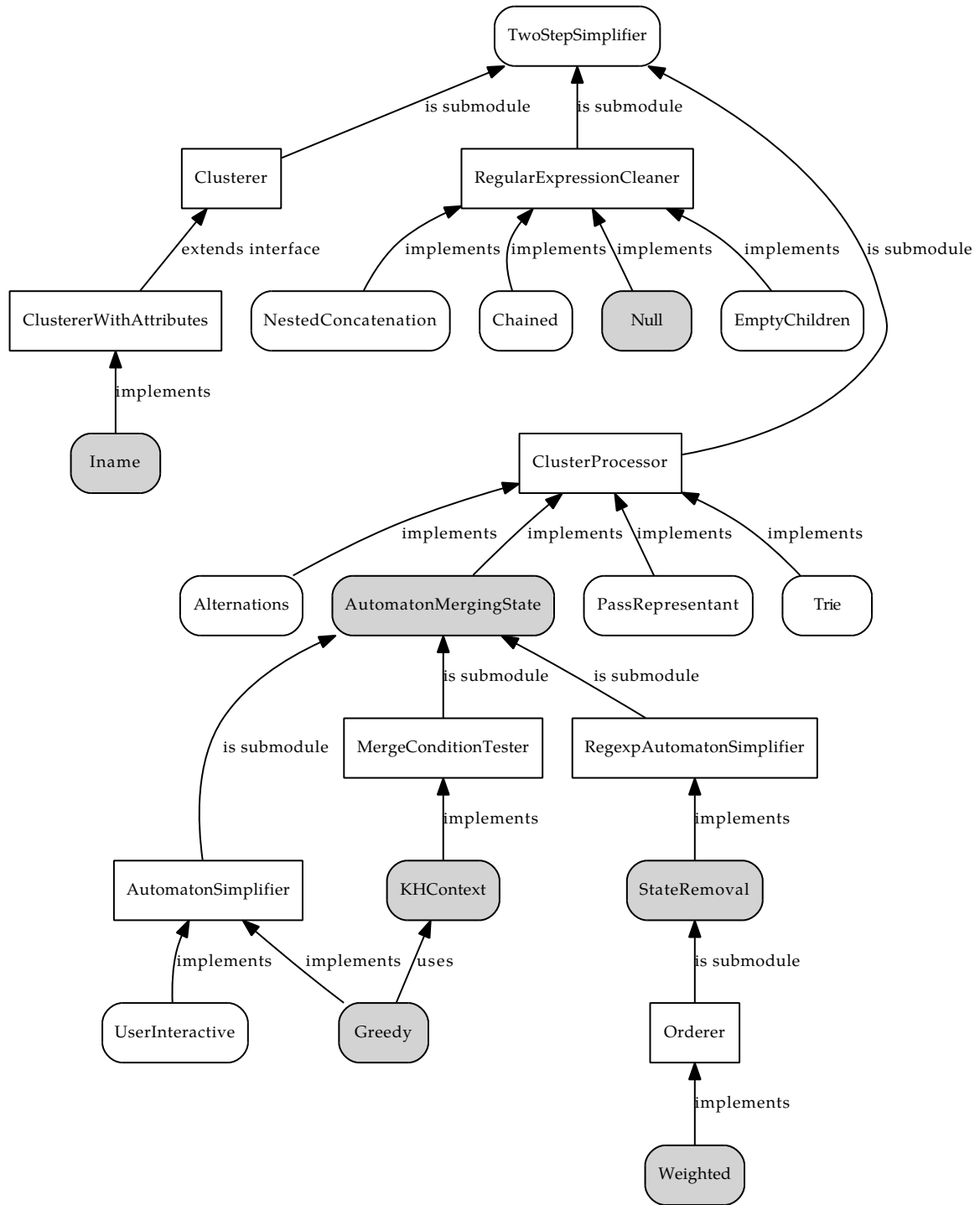


Figure 6: Modules of TwoStep simplifier and their submodules. Filled classes are default selection (best of).

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.
- [HW07] Yo-Sub Han and Derick Wood. Obtaining shorter regular expressions from finite-state automata. *Theor. Comput. Sci.*, 370(1-3):110–120, 2007.
- [KMS⁺] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. *jInfer Architecture*.
- [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.