jInfer TwoStepSimplifier Module Description

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, researchers 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.*

Responsible developer:	Michal Klempa
Required tokens:	cz.cuni.mff.ksi.jinfer.base.interfaces.RuleDisplayer
Provided tokens:	cz.cuni.mff.ksi.jinfer.base.interfaces.inference.Simplifier
Module dependencies:	AutoEditor, Base, JUNG, Lookup
Public packages:	none

1 Introduction

In *TwoStepSimplifier* we provide user a complex solution to grammar simplification problem. This is the only one simplifier shipped with jInfer, but it has many features and exchangeable submodules of which we implement alternatives in basic bundle. We implement is k, h-context method of infering grammar rules (see [**?**]). This process is split into submodules with generic automaton implementation, so it is easy to plug-in any other automaton merging state algorithm. In this document, we will walk over all modules and submodules in simplifier and describe them briefly.

2 Factory pattern usage

Lets stay a while at factory pattern usage. In *TwoStepSimplifier* we employ factory pattern (as described in [**?**, section 4.2]) to divide module into submodules. Since service providing classes are being kept as singletons in the NB platform, we use them as factories: For example:

```
@ServiceProvider(service = AutomatonSimplifierFactory.class)
public interface AutomatonSimplifierFactory extends
                      NamedModule, Capabilities, UserModuleDescription {
  <T> AutomatonSimplifier<T> create();
}
```

The real submodule interface (called familiarly *worker* interface) is the `AutomatonSimplifier<T>` interface (of which instance is returned by factory). For now, it is not important how the worker interface looks like, lets just examine the factory.

In *TwoStepSimplifier* we design service providing factory interfaces so that they extend `NamedModule`, `Capabilities` and `UserModuleDescription`. That means, modules must implement methods:

```
String getName();
String getDisplayName();
String getModuleDescription();
List<String> getCapabilities();
String getUserModuleDescription();
```

Nothing non-standard apart from `getUserModuleDescription`. It returns user-friendly description of the module which is then displayed in properties panels.

Under certain circumstances it is useful to declare method `create()` generic. The *AutomatonSimplifier* works with `Automaton` instances, which are generic too. But automaton simplification does not depend on type of symbol of

automaton transitions, so interface `AutomatonSimplifier<T>` generic too. Factory interface deals with this by defining the `create()` method generic.

Our usage of this factory pattern follows the routine:

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

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

AutomatonSimplifier<AbstractStructuralNode> autSmp = f.<AbstractStructuralNode>create();
...
result= autSmp.simplify(something_to_process);
```

If a module has submodules, we implement lookups for submodule implementations in our own factory `create()` method. Worker class receives factories of all of its submodules as a constructor parameters.

Lets look at `AutomatonMergingStateFactory`. This is factory of module which has *`AutomatonSimplifier`* as a submodule. Its create method (shortened):

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

Methods `getAutomatonSimplifierFactory` and `getRegexpAutomatonSimplifierFactory` are analogical. Here is the former:

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

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

Cluster processor `AutomatonMergingState` then receives factories of *`AutomatonSimplifier`* and *`RegexpAutomaton Simplifier`* 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 submodule, since all factory interfaces are designed the same way as was just described.

3 Structure

`TwoStepSimplifier` is implemented in package `cz.cuni.mff.ksi.jinfer.twostep`. Module is divided into two classes: `TwoStepSimplifier` (main logic) and `TwoStepSimplifierFactory` (lookups, interface to other modules). The latter is registered as service provider (and implements) of `Simplifier` interface (defined in package `cz.cuni.mff.ksi.jinfer.base.interfaces.inference`).

Its main method is `start()` which receives Initial Grammar in form of

```
List<Element> grammar
```

Each grammar rule is represented as class `Element`, where element is left side of rule and its `getSubnodes()` method returns regular expression representing right side of rule (as described in [**?**, 3.1.1], they are all concatenations). Second parameter is

```
SimplifierCallback callback
```

Callback which should be invoked when the simplification is done. On output, simplifier provides Simplified grammar as a parameter of callback function:

```
void finished(List<Element> grammar);
```

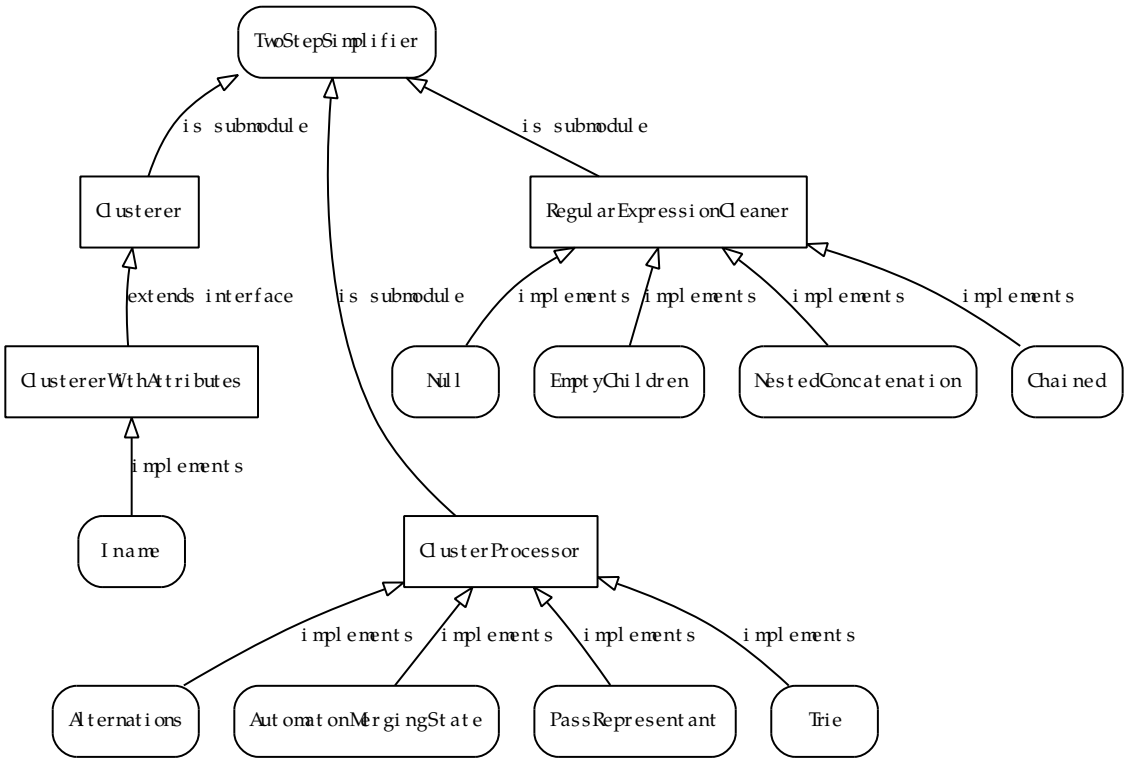


Figure 1: Submodules of TwoStep simplifier with their implementations

Thus simplifier receives Initial Grammar and returns Simplified Grammar, both represented same way - list of elements.

`TwoStepSimplifierFactory.start()` creates new `TwoStepSimplifier` class instance, providing three factories of submodules to its constructor. Then it calls `simplify(initialGrammar)` method of this instance and its result is passed as parameter to callback function. That's all, the magic is in `TwoStepSimplifierFactory` class.

4 Modular design

`TwoStepSimplifier` is inspired by [**?**] 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

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. There is also third submodule called *`RegularExpressionCleaner`* actually, its purpose is just to beautify output regular expressions, no inference logic is implemented there. Modules are drawn on fig. 1. We provide one *`Clusterer`*, 4 *`ClusterProcessor`* and 4 *`RegularExpressionCleaner`* implementations. Each of those will be explained further in this document.

Method `TwoStepSimplifier.simplify()` does basically this:

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

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

```
// 3. process rules
ClusterProcessor<AbstractStructuralNode> processor =
             clusterProcessorFactory.create();

for (Cluster<AbstractStructuralNode> cluster : clusterer.getClusters()) {
  AbstractStructuralNode node =
                           processor.processCluster(clusterer, cluster.getMembers());
  RegularExpressionCleaner<AbstractStructuralNode> cleaner =
                           regularExpressionCleanerFactory.<AbstractStructuralNode>create();
  // 4. add to rules
  finalGrammar.add(
              new Element(node.getContext(),
              node.getName(),
              node.getMetadata(),
              cleaner.cleanRegularExpression(((Element) node).getSubnodes()),
              attList));
}

return finalGrammar;
```

It creates clusterer, gives it all rules, orders it to cluster elements. Then, empty list of elements is created as final (simplified) grammar. For each input rule in `initialGrammar`, submodule *`ClusterProcessor`* is called to do infering of regular expression of that element. Finally, regular expression cleaning is done in submodule and new `Element` instance is created as a copy of processed node, but with infered and cleaned regexp.

Sentinel processing There are some specialities in processing sentinel elements (see [**?**, 3.2]). Sentinels can be only on right side of rules and they have to be send to clusterer, as someone further in the chain can ask for cluster of sentinel element - which appears on right side. But sentinels have no content, they must not be taken into account when infering attribute properties (whether it is required or optional) - since sentinel have no attributes at all.

Now we will examine submodules for clustering, processing and cleaning.

4.1 Clusterer submodule

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 members 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) on output. It has methods `add()` and `addAll()` for adding items for clustering. Centerpart 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 [**?**, p. 12]). After clustering, implementation should hold clusters in member, as it will be later requested 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 (`List<Cluster<T>>`) from clusterer by calling `getClusters()` method. Basic work usage with clusterer is:

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

4.1.1 ClustererWithAttributes extended interface

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)`.

We provide one implementation of this interface described in **??**.

4.2 ClusterProcessor module

ClusterProcessor takes rules of one cluster of element and somehow obtains regular expression for that set of elements. It returns one 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 the 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 the resulting element.

Worker interface itself is defined as follows:

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

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 describe each *ClusterProcessor* implementation we've got in sections **??** through **??**.

4.3 RegularExpressionCleaner module

Last we examine *RegularExpressionCleaner* interface. Purpose of this submodule is only to make output regular expression corrections, to make them nicer. For example it is common that converting automaton to regular expression by state elimination produces nested concatenations such like $(name, (person, id))$. To convert such expression into $(name, person, id)$, one can implement this interface and connect it to work in chain. Interface is definition is straightforward:

```
public interface RegularExpressionCleaner<T> {
  Regexp<T> cleanRegularExpression(Regexp<T> regexp);
}
```

Given regular expression, return regular expression. Converting of regular expression is commonly a recursive task and all our implementations work this way.

5 Preferences

All settings provided by *TwoStepSimplifier* are project-wide, the preferences panel is in `cz.cuni.mff.ksi.jinfer.twostep.properties` package. It is possible to:

- select *Clusterer* submodule implementation from those installed
- select *ClusterProcessor* submodule implementation from those installed
- select *RegularExpressionCleaner* submodule implementation from those installed

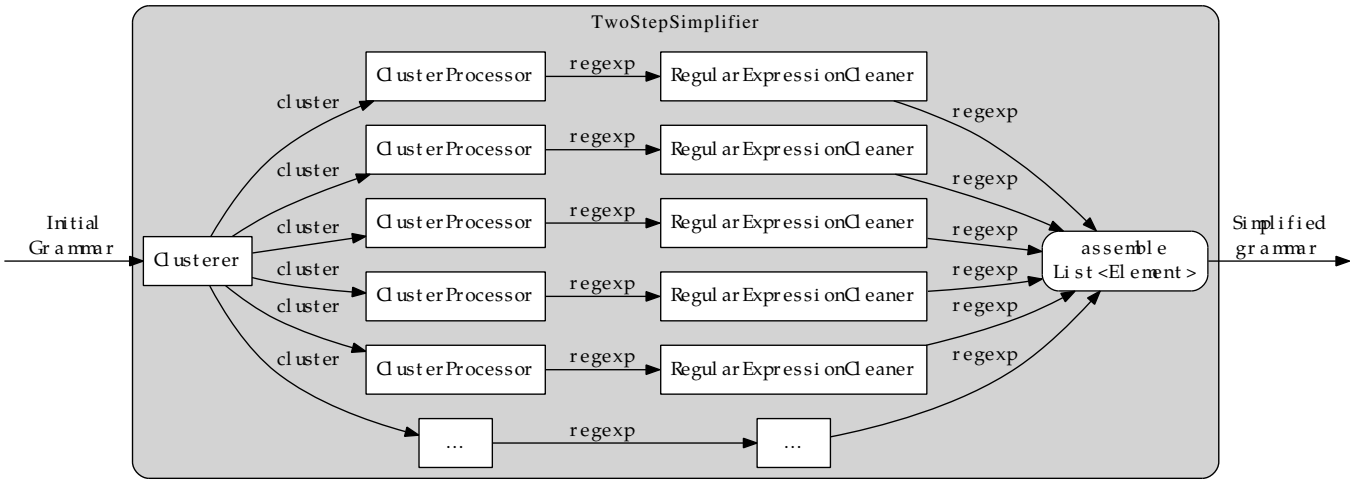


Figure 2: TwoStep data flow

6 Data flow

Process illustrated on fig. **??**:

1. On input there is Initial Grammar, which is ruled into *`Clusterer`*.
2. Then, each cluster is sent to submodule *`ClusterProcessor`*, which returns regular expression for that cluster.
3. Regular expression is sent to *`RegularExpressionCleaner`* submodule for cleaning.
4. From each cluster processing, regular expression is added to the list of simplified grammar rules.
5. List of simplified rules is returned (Simplified grammar).

7 Submodules implementations

7.1 Iname (ClusteterWithAttributes)

We cluster elements in `Iname<AbstractStructuralNode, Attribute>` class, which takes `AbstractStructuralNode` classes to cluster as main, and `Attribute` classes as attributes. Clustering is done based on elements `getName` equality test (ignore case). For each rule (element), it is first clustered by finding cluster where representant has same name - or create new cluster with this element as representant. Then we process right side of this element rule (that are nodes from `getSubnodes`). Since right side of rule is always concatenation, we simply take `.getSubnodes().getTokens()` list and iterate through it. Each node on right side, is examined:

- if it is simple data throw it to `SimpleDataClusterer` class which is associated with main element cluster. It does nothing more, than holds all `SimpleData` instances in one cluster. But in future it may be replaced to cluster simple data somehow, to obtain meaningful content models in schemas.
- if it is element and it is tagged as sentinel by metada - search main clusters to find cluster with representant of same `getName()`, or create new cluster with this sentinel as representant. From IGG, sentinels may be only on right sides of rules and since each element in schemas has to be defined, there must exist another element with same name, which is not sentinel (and maybe it will come to process in future). So there can't be cluster with only one sentinel element in it.
- do nothing otherwise, since it is element, it has to have its subnodes defined, and therefore it is element that is proper grammar rule and therefore it has to be somewhere in Initial Grammr, thus it is already processed or is on schedule.

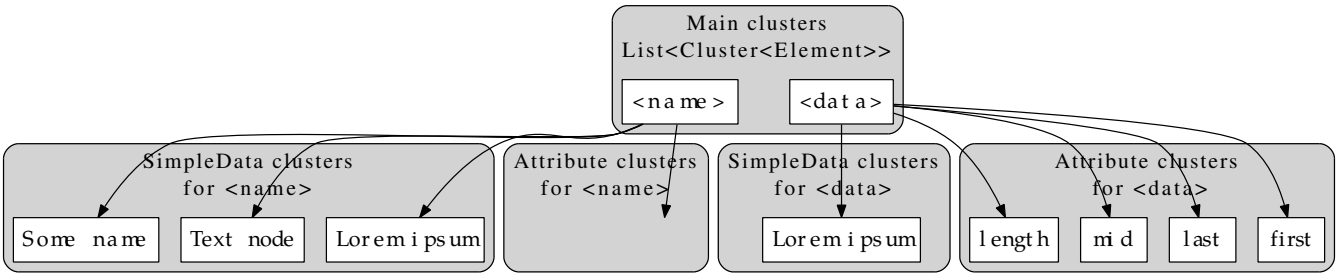


Figure 3: Iname clusterer structure

Attributes of element are processed through helper attribute clusterer, which is created for each element cluster. We have bunch of elements of same name in cluster and one attribute clusterer associated with this bunch. This attribute clusterer is given all attributes instances encountered in all elements that are in bunch. It clusters them by name (case insensitive). Each main cluster also has `SimpleData` clusterer associated with it. Whole scheme is on fig. **??**.

To rehearse, there are main clusters for elements, for each main cluster there are two helper clusterer - one for attribute clusters and one for simple data clusters.

7.2 PassRepresentant (ClusterProcessor)

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.

7.3 Alternations (ClusterProcessor)

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.

7.4 Trie (ClusterProcessor)

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. The tree is the final regular expression.

7.5 AutomatonMergingState (ClusterProcessor)

`AutomatonMergingState` is implementation of merging state algorithm on nondeterministic finite automaton (see package `cz.cuni.mff.ksi.jinfer.twostep.processing.automatonmergingstate`).

7.5.1 Structure

Shortened code of cluster processor operation:

```
// 1. Construct PTA
Automaton<AbstractStructuralNode> automaton = new Automaton<AbstractStructuralNode>(true);

// Take each rule in cluster and pass right side to automaton to create PTA
for (AbstractStructuralNode instance : rules) {
  Element element = (Element) instance;
  Regexp<AbstractStructuralNode> rightSide = element.getSubnodes();
```

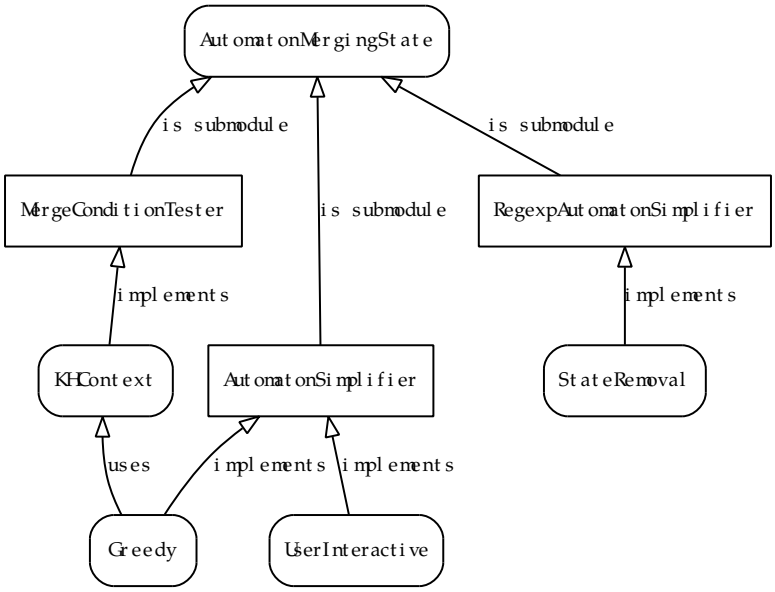


Figure 4: Submodules of AutomatonMergingState cluster processor

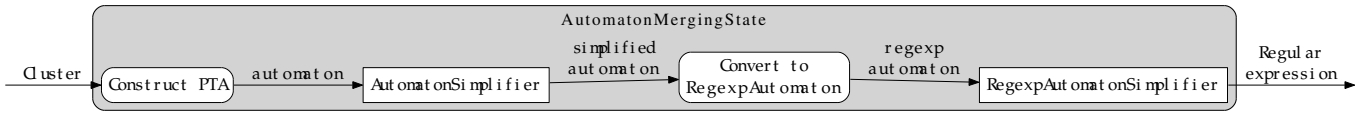


Figure 5: Data flow of AutomatonMergingState cluster processor

```
  List<AbstractStructuralNode> rightSideTokens = rightSide.getTokens();

  List<AbstractStructuralNode> symbolString = new LinkedList<AbstractStructuralNode>();
  for (AbstractStructuralNode token : rightSideTokens) {
    symbolString.add(clusterer.getRepresentantForItem(token));
  }
  automaton.buildPTAOnSymbol(symbolString);
}

// 2. Simplify automaton by merging states using automatonSimplifier
Automaton<AbstractStructuralNode> simplifiedAutomaton =
    automatonSimplifier.simplify(automaton, elementSymbolToString);

// 3. Convert Automaton<AbstractStructuralNode> to RegexpAutomaton<AbstractStructuralNode>
RegexpAutomaton<AbstractStructuralNode> regexpAutomaton =
    new RegexpAutomaton<AbstractStructuralNode>(simplifiedAutomaton);
// 4. Call regexpAutomatonSimplifier to obtain regular expression from regexpAutomaton
Regexp<AbstractStructuralNode> regexp =
    regexpAutomatonSimplifier.simplify(regexpAutomaton, regexpAbstractToString);

// 5. Return element with regexp
return new Element(
      new ArrayList<String>(),
      rules.get(0).getName(),
      new HashMap<String, Object>(),
      regexp,
      new ArrayList<Attribute>()
```

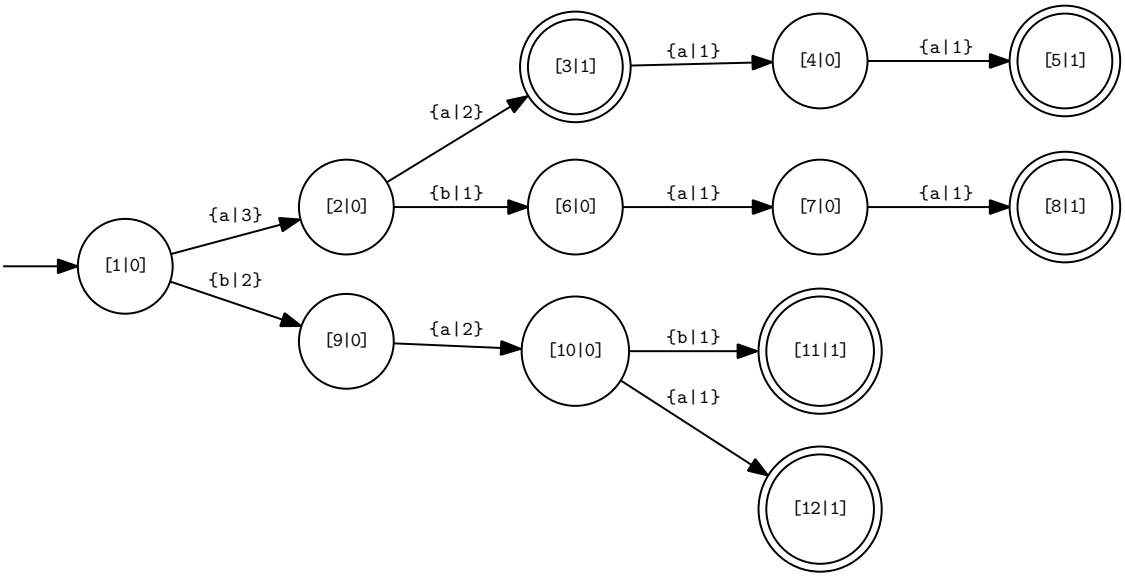


Figure 6: PTA created from input strings: $aa, aaaaa, abaa, bab, baa$

```
);
```

The whole submodule structure is drawn on fig. **??**.

7.5.2 Data flow

It works in steps:

1. 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.
2. Simplified automaton is then converted to an instance of `RegexpAutomaton` (see **??**) by using clone constructor. 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`.
3. `RegexpAutomaton` 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.

The data flow is on fig. **??**.

7.5.3 Data structures - Automaton

Class `Automaton` is shortly described in [**?**]. In this place, it is interestesting to look on method `buildPTAOnSymbol`. Given `List<T> symbolString` it traverses automaton - comparing transition symbols with symbols in string. It follows transitions in automaton until first difference with string symbol is found (like in prefix trees). On different symbol, it creates new branch and all states and transitions on that branch. Last state visited got's incremented its `finalCount` value, as one more input string ended in this state. Lets see code:

```
public void buildPTAOnSymbol(List<T> symbolString) {
  State<T> xState= this.getInitialState();
  for (T symbol : symbolString) {
    Step<T> xStep= this.getOutStepOnSymbol(xState, symbol);
    if (xStep != null) {
      xStep.incUseCount();
      xState= xStep.getDestination();
    } else {
      State<T> newState= this.createNewState();
```

```
      Step<T> newStep= this.createNewStep(symbol, xState, newState);
      assert newStep.getDestination().equals(newState);
      xState= newStep.getDestination();
    }
  }
  xState.incFinalCount();
}
```

Start from initial state traverse automaton by asking for `getOutStepOnSymbol`. If there is such step (transition), follow it to its destination state and move the position in `symbolString`. If there is no such step, create one and create also the destination state, then follow this newly created step to new state.

As you can see, it is important that class `T` has proper implementation of `.equals()` method, as symbols are tested for equality by calling this method on them. By the way, that is why we cluster all simple data and all sentinels on right sides of rules. To automaton, instead of instances on right side of rules, their cluster representants are inserted as one can see in section **??**. This assures, that comparisons in automaton are made by using java `Object.equals()` reference comparison, but for each cluster member, its representant is only present in automaton. That results in cluster-equal behaviour. Example of prefix-tree automaton is on fig. **??**.

7.5.4 Data structures - RegexpAutomaton

Class `RegexpAutomaton` is extending class `Automaton` with generic type `T` set to `Regexp<T>`. That is, you have to instantiate it with same symbol type (e.g. `AbstractStructuralNode`) as automaton. Implementation can be found in package `cz.cuni.mff.ksi.jinfer.twostep.processing.automatonmergingstate.regexping`, that is same, as where *`RegexpAutomatonSimplifier`* interface is defined. Implementation is nothing special, it has only copy constructor which creates regular expression automaton from ordinary automaton by enclosing each symbol on transition of original automaton by `Regexp.TOKEN`.

7.5.5 AutomatonSimplifier module

`AutomatonSimplifier` module has its worker interface defined as:

```
public interface AutomatonSimplifier<T> {
  Automaton<T> simplify(Automaton<T> inputAutomaton,
                        SymbolToString<T> symbolToString) throws InterruptedException;
  Automaton<T> simplify(Automaton<T> inputAutomaton,
                        SymbolToString<T> symbolToString,
                        String elementName) throws InterruptedException;
}
```

Given input automaton it returns automaton, there is nothing magical on it. Interface is defined as generic, implementation don't have to be, but there is no reason to not make them generic. When implementation needs to present automaton to user, however, it needs some string representation of symbols to display on automaton transitions. For this reason, the second parameter, `symbolToString` (implementation of `SymbolToString` interaface) is given. It is responsible for converting symbol to string. To *`AutomatonSimplifier`* it is passed from *`AutomatonMergingState`* cluster processor, where we know that symbols are of type `AbstractStructuralNode`, so we implement `SymbolToString` by using `getName()` method on nodes and pass it down to generic `AutomatonSimplifier` submodule. Overloaded version is to inform user about name of processed element when presenting automaton by *`AutoEditor`*. This version of interface will probably change in future.

By this genericity, simplifier can be used to perfom same algorithms for simplifying not only to infer rules for elements, but maybe also to infer patterns strings for content model of attributes. The actual implementation of this still waits for its developer to come. Implementations of *`AutomatonSimplifier`* are discussed in **??** and **??**.

7.5.6 RegexpAutomatonSimplifier submodule

Purpose of *`RegexpAutomatonSimplifier`* is to obtain regular expression from given automaton. Interface for doing this is thus simple:

```
public interface RegexpAutomatonSimplifier<T> {
  Regexp<T> simplify(RegexpAutomaton<T> inputAutomaton,
```

```
                    SymbolToString<Regexp<T>> symbolToString) throws InterruptedException;
}
```

Once again we are encountering `symbolToString` with same purpose as before, if submodule would need to present automaton to user it has to be able to convert symbols of (at runtime) unknown type to strings. Our implementation of *`RegexpAutomatonSimplifier`* is discussed in **??**

7.5.7 MergeConditionTester submodule

`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.

Module *`AutomatonSimplifier`* implements solution searching logic in simplifying automaton by merging states. One can implement ACO heuristics or MDL principle heuristics as *`AutomatonSimplifier`* submodule, and can use any of *`MergeConditionTesters`* provided.

7.6 NestedConcatention (RegularExpressionCleaner)

This cleaner does exactly what we have just described. It converts nested concatenations into flat ones. Example: $(name, (person, id))$ is converted to $(name, person, id)$.

7.7 Null (RegularExpressionCleaner)

This cleaner just returns regexp it receives and does nothing. Plug this one into chain, if you want to omit this step in inference.

7.8 EmptyChildren (RegularExpressionCleaner)

Wipes out regular expressions of type: concatenation, alternation, permutation, which have empty `children` member. For example some pitty inferring method produces unnecessary empty regexps in regexp tree as:

$$(name, (), (person, id))$$

There is one empty concatenation/alternation/permutation with no sense. This is wiped out by this cleaner to produce

$$(name, (person, id))$$

7.9 Chained (RegularExpressionCleaner)

Enables user to chain more cleaners with output from first one plugged as input into second one and so one. Thus regular expression from inference can pass through *`EmptyChildren`* and them through *`NestedConcatenation`* cleaners and then it is returned.

7.10 Greedy (AutomatonSimplifier)

We implement greedy strategy of automaton simplifying 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:

```
while (there exist pair of states that are equivalent) {
  merge them
}
```

We provide one *`MergeConditionTester`* implementation, see **??**.

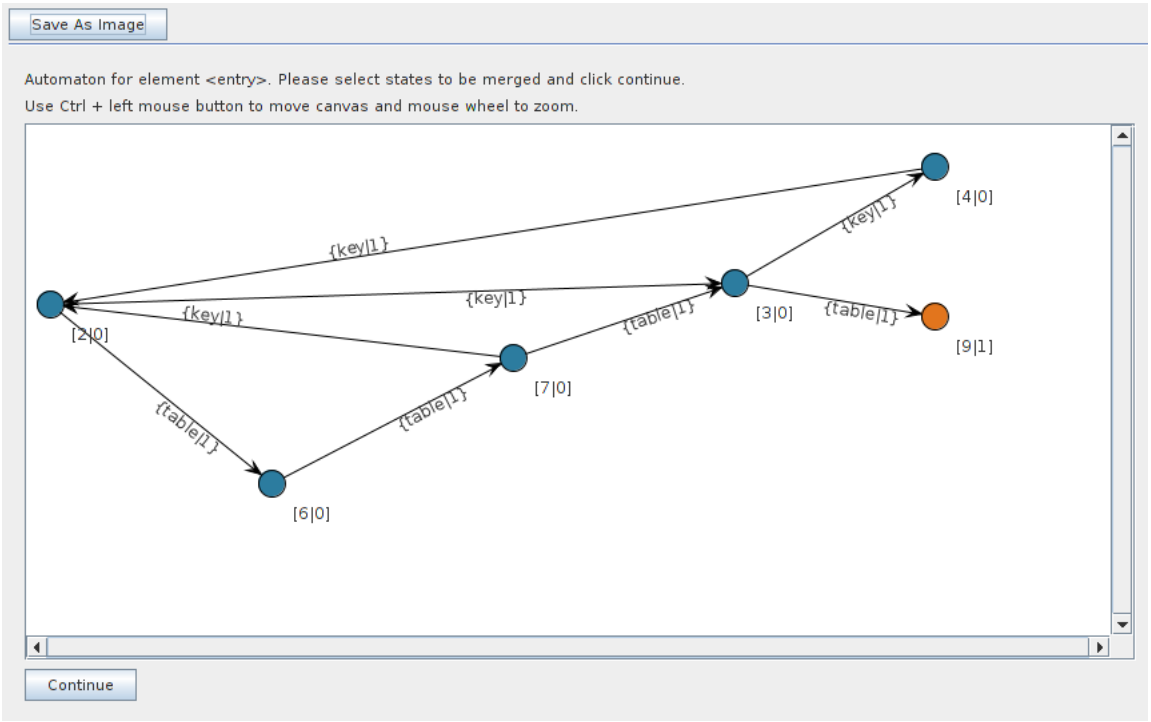


Figure 7: Screenshot of UserInteractive (AutomatonSimplifier).

7.11 UserInteractive (AutomatonSimplifier)

We have created automaton simplifier which displays user input automaton asking him to select some states to merge. Then it merges them and asks user again:

```
repeat {
  draw automaton
  if (user selects more than one state to merge) {
    merge them
  }
} until (user selected at least one state in last trial)
```

We use *`AutoEditor`* module described in [**?**]. To make reader more happy, we include one screenshot of automaton visualization on picture **??**.

7.12 KHContext (MergeConditionTester)

We have implemented *k,h-context* state equivalence (see [**?**]) in class `KHContext` (implements `MergeConditionTester` interface), which is used by `Greedy` to test mergability of states by default configuration. TODO vektor : mention this? (we have stupid algorithm:) We don't use $k - grams$ algorithm from [**?**], but a simple DFS.

7.13 StateRemoval (RegexpAutomatonSimplifier)

We are using state removal method (see [**?**]) to convert regexp automaton into equivalent regular expression. This is implemented in `StateRemoval` class (on fig. **??**) which implements `RegexpAutomatonSimplifier` interface. before algorithm starts, states superInitial and superFinal are added to automaton, former one with λ-transition to initial state. From all final states λ transition to superFinal state is added. State removal works by removing states of automaton (and redirecting transitions properly) until there are last to two states - superInitial and superFinal. After removing all states, there is only one transition from superInitial to superFinal state. That transition has final regular expression on it as symbol, it is read and returned.

We defined one submodule of this class with *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 state returned.

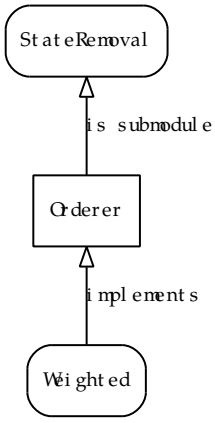


Figure 8: Modules of StateRemoval.

7.13.1 Data Structures - StateRemovalRegexpAutomaton

We extend `RegexpAutomaton` to obtain automaton that fits state removing procedure. This implementation has methods `createSuperInitialState` and `createSuperInitialState`. They does exactly what their names stands for. SuperInitial state has only one λ-out-transition pointing to original initial state. SuperFinal state has as many λ-in-transitions, as there are original final states - from each one, it gets one. Without going further to many technical details, we just mention method `removeState`, which removes given state from automaton. Transitions are handled this way:

1. Collapse all loop transition of removed state to one loop transition with new `Regex.ALTERNATION` of all regexps on original loops (with Kleene interval set).
2. Collapse all in | out-transitions, which same source | destination state to one transition with `Regexp.ALTERNATION` of original regexps.
3. For each in-transition, loop, and out-transition, add transition from in-transition source to out-transition destination with symbol of `Regexp.CONCATENATION` of in-transition regexp, loop regexp and out-transition regexp. Thus by-passing the state completely.
4. Remove the state and transitions associated with it.

Whole process is step-by-step illustrated on fig. **??**.

7.14 Weighted (Orderer)

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.

8 Extensibility

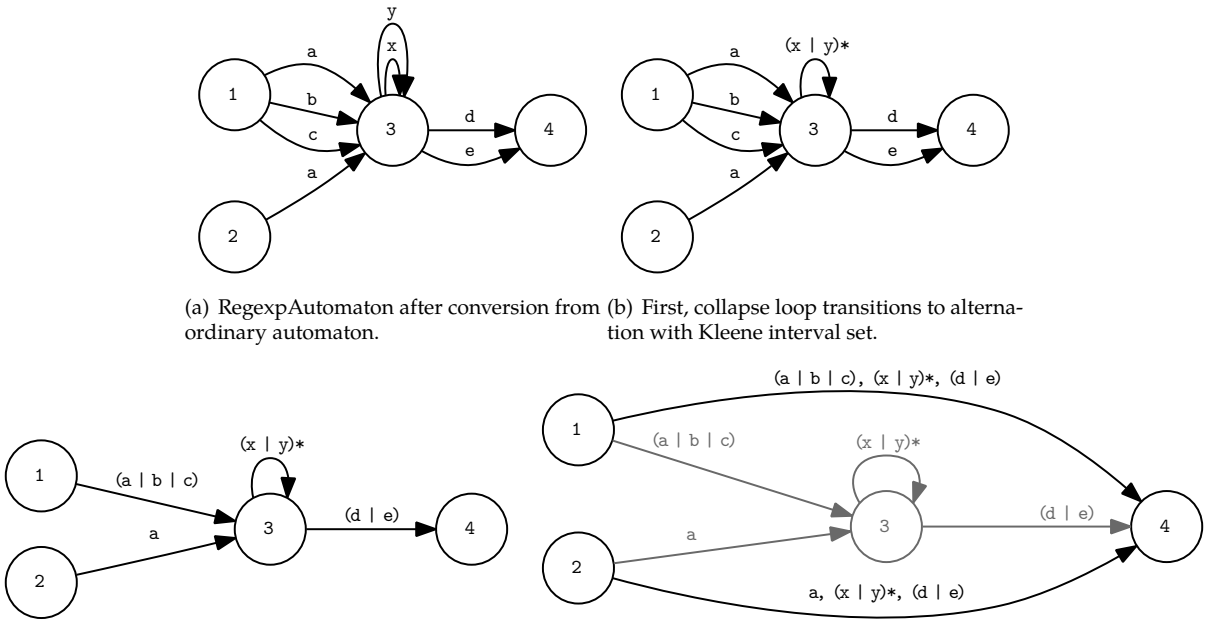
If you are willing to, you can replace each of three submodules of *TwoStepSimplifier*. It may be useful to replace default *ClustererWithAttributes* implementation to a clever one. And regular expression cleaning will be probably target of adding new implementations, too. We recommend replacing other parts at lower levels, however. You will need to replace *ClusterProcessor* only if you want to write infering method that is not based on merging state algorithm (or NFA's at all).

If this is not your case, you can replace module *AutomatonSimplifier* to implement ACO heuristics or MDL principle in searching solutions of merging states.

Of course, merge criterion is propably the point to add implementation, implement *MergeConditionTester* interface to obtain for example s, k-string merge criterion.

Changing ordering of states to remove in *StateRemoval* module is easy by replacing *Orderer* implementation, for example to one of suggested in [**?**] may lead to shorter resulting regular expressions.

Whole TwoStep submodules structure is on fig. 3.



(a) RegexpAutomaton after conversion from ordinary automaton.

(b) First, collapse loop transitions to alternation with Kleene interval set.

(c) Collapse in | out-transitions from | to same states.

(d) Bypass state 3 by adding concatenation transitions from states 1 and 2 to state 4. Remove state 3 and associated transitions.

Figure 9: RegexpAutomaton and removing state 3 from it.

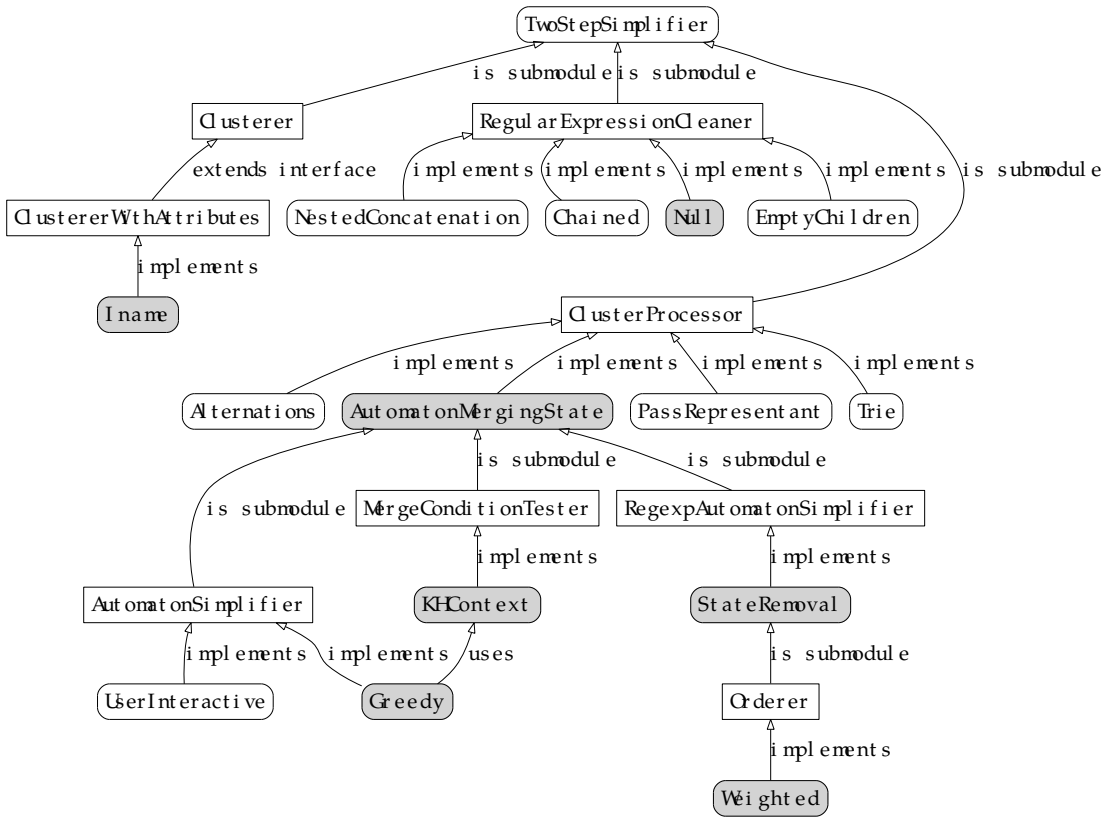


Figure 10: Modules of TwoStep simplifier and their submodules. Filled classes are default selection (best of).