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. While this is the only simplifier bundled with jInfer, its strong modularity ensures that it can be extended easily. For most of the points of extension we indeed provide multiple implementations. These vary from trivial proof-of-concept to for example k, h-context method for grammar rules inference (see [Aho96]). Strong emphasis is put on generic automata, therefore it is easy to plug-in any 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 [KMS+a, 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. 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 [KMS+a, 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);
```

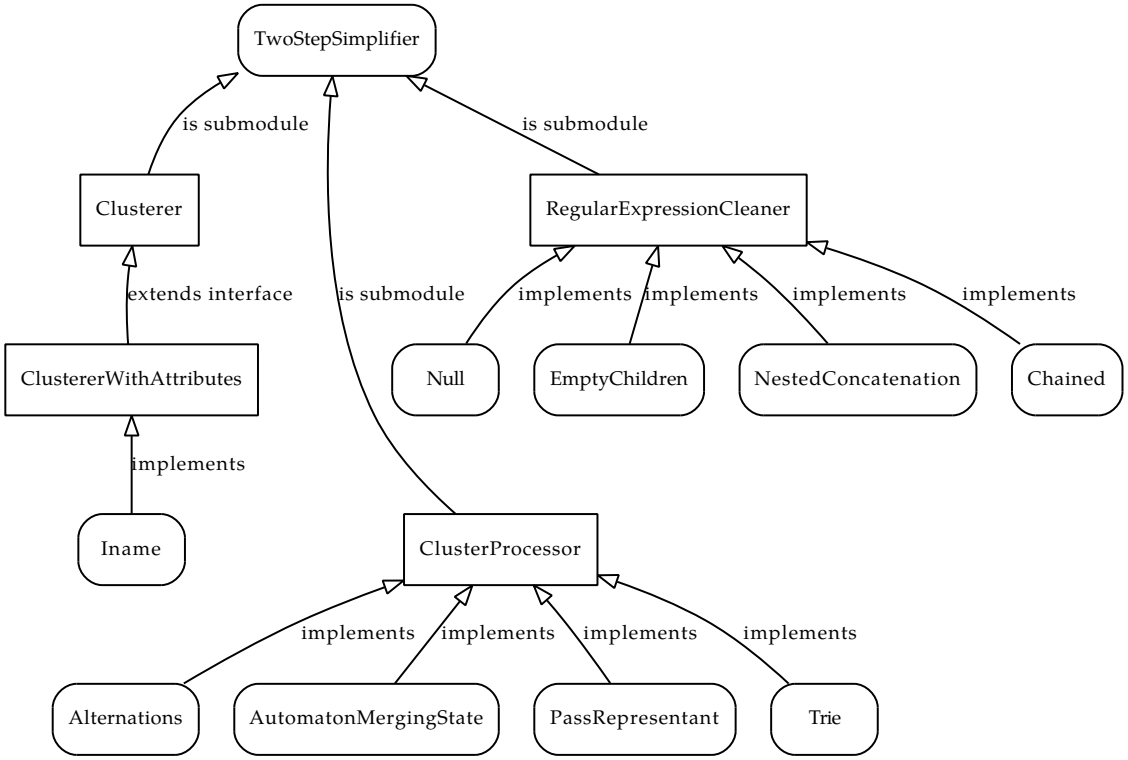


Figure 1: Submodules of TwoStep simplifier with their implementations

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 [VMP08] design. Inference proceeds in two steps:

1. Clustering the 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 a cluster.

Clustering is delegated to *`Clusterer`* submodule, and task of regular expression inference for each cluster is delegated 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()` basically does the following.

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

// 2. prepare emtpy simplified grammar
List<Element> simplifiedGrammar= 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
  simplifiedGrammar.add(
              new Element(node.getContext(),
              node.getName(),
              node.getMetadata(),
              cleaner.cleanRegularExpression(((Element) node).getSubnodes()),
              attList));
}

return simplifiedGrammar;
```

First, it creates a clusterer, gives it all the rules and performs the clustering. Then, empty list of elements is created as the simplified grammar. For each input rule in `initialGrammar`, submodule *`ClusterProcessor`* is invoked to infer the regular expression of that element. Finally, regular expression cleaning is done in a submodule and new `Element` instance is created as a copy of processed node, but with inferred and cleaned regexp.

Sentinel processing There are some specialities in processing sentinel elements (see [KMS+a, 3.2]). Sentinels can be only on right side of rules and they have to be sent 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 inferring attribute properties (whether it is required or optional) - since sentinels have no attributes at all.

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

4.1 Clusterer submodule

Clustering is implemented in `cz.cuni.mff.ksi.jinfer.twostep.clustering` package. One cluster is represented by the `Cluster<T>` class with `T` as type parameter of clustered items. This class simply holds a `Set` of members of this cluster and one of the references as `representant` member.

Any clusterer has to comply to `Clusterer` interface. 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. Main part is the method `cluster()`, which does the clustering itself.

As this may be a time-consuming operation, method throws `InterruptedException`. Implementation should take care of checking whether thread is user interrupted (see [KMS+a, p. 12]). To be able to respond to calls to `T getRepresentantForItem(T item)` method, every clusterer implementation has to keep all its clusters even after the clustering is finished. Using this method, given an item the correct cluster representative for this item has to be returned.

If no such cluster exists (item was not added for clustering before), we recommend throwing an exception rather than returning `null`. Missing an item probably indicates error in algorithm rather than normal workflow. One can get all the clusters (`List<Cluster<T>>`) from clusterer by calling `getClusters()` method. Basic usage of the 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 might have noticed that the *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 representative of type `T` (that is representative 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 7.1.

4.2 ClusterProcessor module

ClusterProcessor takes rules of one cluster of element and somehow obtains regular expression representing 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 the right name for the resulting element) and with subnodes set to regular expression inferred. It processes attributes of all elements in cluster to obtain meaningful schema attribute specification and these attributes have to be attached to the resulting element.

Interface of submodule is defined as follows.

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

Why is the cluster processor given a clusterer instance? Rules themselves contain information about which elements to process, but clusterer has more information about the topic. Clusterer can tell you the representative for any element in the whole input (not only those elements in rules, but also those that may be on right side of rules). Also, clusterer (if it is with attributes) has information about attributes in each cluster.

We describe each *ClusterProcessor* implementation shipped in jInfer in sections 7.2 through 7.5.

4.3 RegularExpressionCleaner module

Finally we examine the *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 as $(name, (person, id))$. To convert such expression into $(name, person, id)$, one can implement this interface and connect it to work in chain. Interface definition is straightforward here.

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

Given regular expression, return regular expression. Conversion 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 set the following.

- Select *Clusterer* submodule implementation from those installed.
- Select *ClusterProcessor* submodule implementation from those installed.
- Select *RegularExpressionCleaner* submodule implementation from those installed.

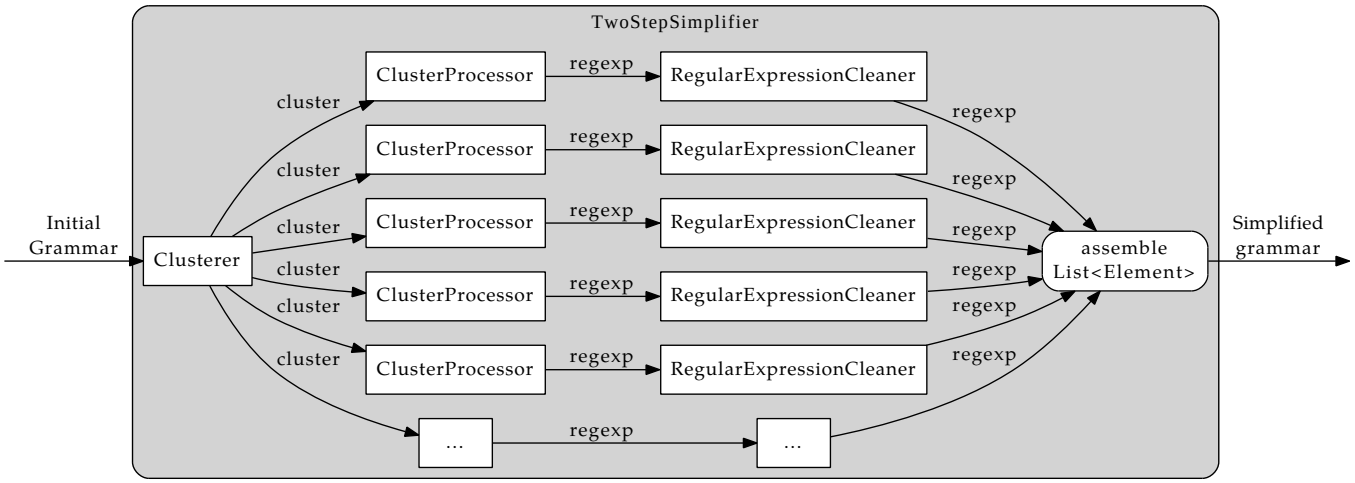


Figure 2: TwoStep data flow

6 Data flow

Process illustrated in fig. 2 is as follows.

1. Initial Grammar on input, which is sent into *`Clusterer`*.
2. 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. Regular expressions representing all the processed clusters are added to the list of simplified grammar rules.
5. List of simplified rules is returned (Simplified grammar).

7 Submodule implementations

7.1 Iname (ClusteterWithAttributes)

We cluster elements in `Iname<AbstractStructuralNode, Attribute>` class (Iname je clusterer!), which takes `AbstractStructu` classes to cluster as main, and `Attribute` classes as attributes. Clustering is done based on element's `getName()` equality (ignoring case). For each rule (element), it is first clustered by finding the cluster which representative has the same name - or create new cluster with this element as representative. Then we process right side of this element rule (that are nodes from `getSubnodes()`). Since right side of rule is always concatenation (*`TwoStep`* accepts only simple grammar representation), we simply take `.getSubnodes().getTokens()` list and iterate through it. Each node on right side, is examined in the following way.

- 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 - search main clusters to find cluster with representative of same `getName()`, or create new cluster with this sentinel as representative. 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 Grammar, thus it is already processed or is on schedule.

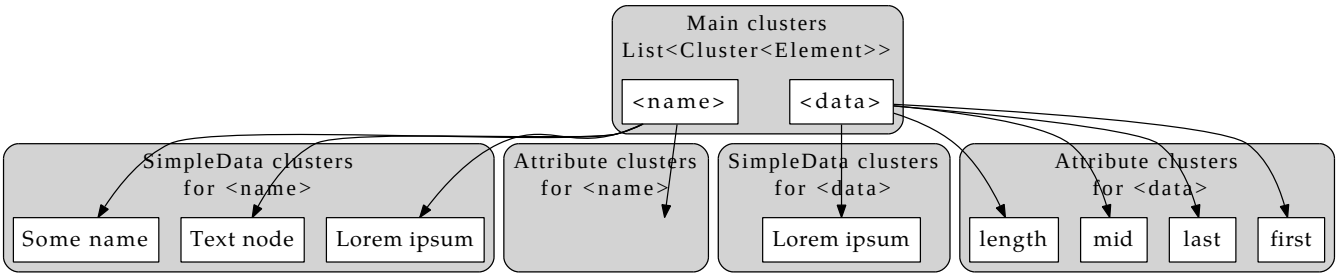


Figure 3: Iname clusterer structure

Attributes of element are processed through a helper attribute clusterer, which is created for each element cluster. We have bunch of elements of same name in a 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. 3.

Once again, there are main clusters for elements, for each main cluster there are two helper clusterers - one for attribute clusters and one for simple data clusters.

7.2 PassRepresentant (ClusterProcessor)

Trivial ClusterProcessor implementation. For each cluster return its representative as the rule to be in resulting schema. This has nothing to do with grammar simplification, it is just a proof-of-concept submodule. 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 a 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 an implementation of merging state algorithm on nondeterministic finite automaton (see package `cz.cuni.mff.ksi.jinfer.twostep.processing.automatonmergingstate`).

7.5.1 Structure

Abridged code of cluster processor operation follows.

```
// 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;
```

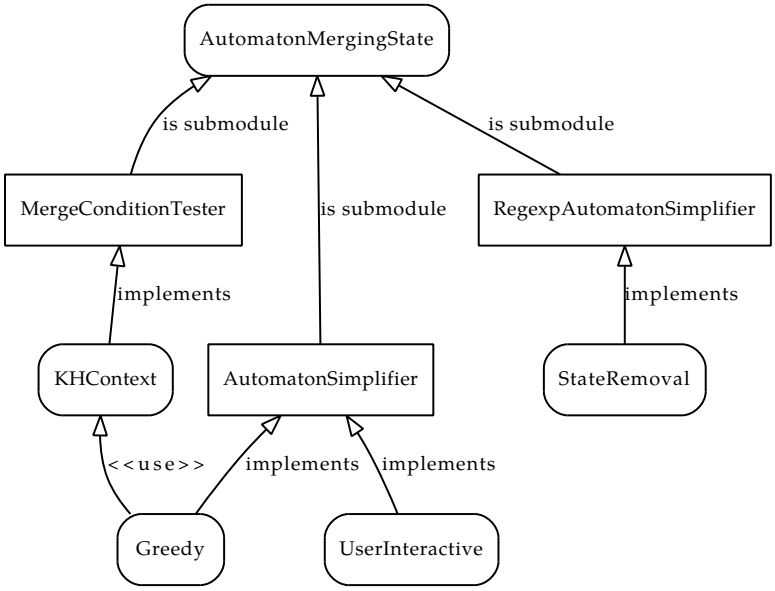


Figure 4: Submodules of AutomatonMergingState cluster processor

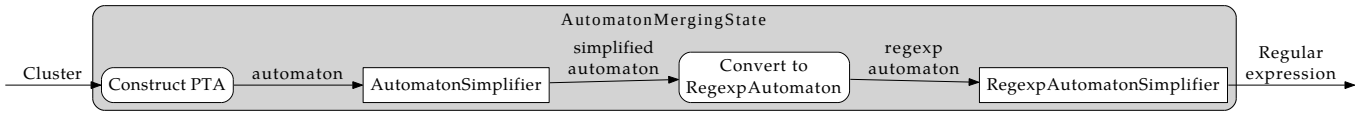


Figure 5: Data flow of AutomatonMergingState cluster processor

```
  Regexp<AbstractStructuralNode> rightSide = element.getSubnodes();

  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(),
```

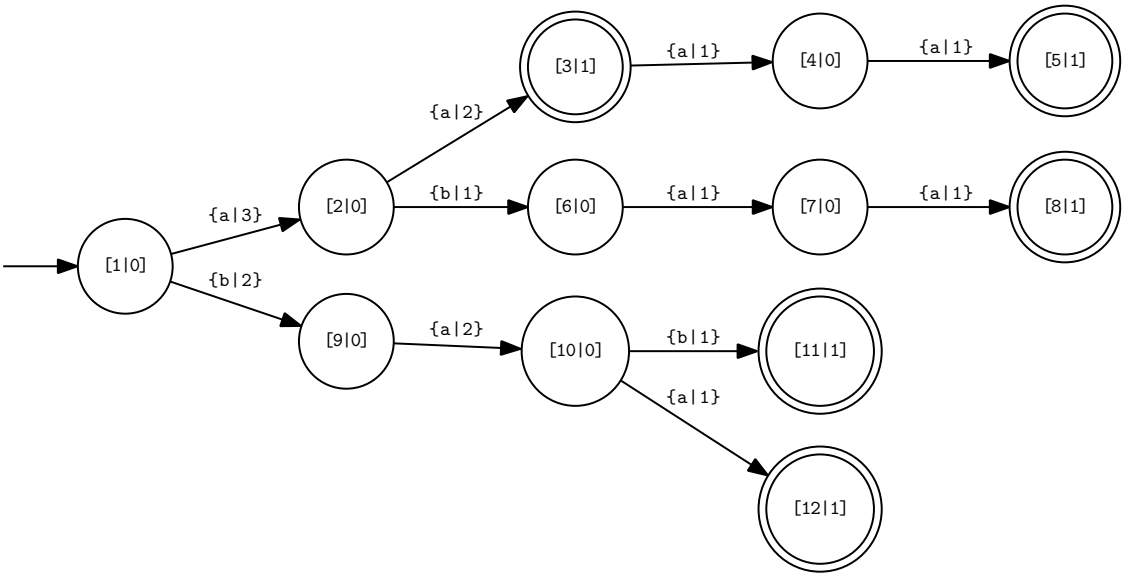


Figure 6: PTA created from input strings: $aa, aaaaa, abaa, bab, baa$

```
new HashMap<String, Object>(),
regexp,
new ArrayList<Attribute>()
);
```

The whole submodule structure is in fig. 4.

7.5.2 Data flow

AutomatonMergingState works in following steps.

1. Create prefix-tree automaton (PTA) from positive examples - right sides of rules. Call its submodule *AutomatonSimplifier* to modify PTA to some generalized automaton by merging states.
2. Simplified automaton is then converted to an instance of `RegexpAutomaton` (see 7.5.4) 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 represent the same language.

The data flow is in fig. 5.

7.5.3 Data structures - Automaton

Class `Automaton` is shortly described in [KMS+a]. Let's take a look at method `buildPTAOnSymbol` here. 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 gets its `finalCount` value incremented, as one more input string ended in this state. The code is as follows.

```
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();
}
```

Starting from initial state, traverse the 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, this is why we cluster all simple data and all sentinels on right sides of rules. While adding a concatenation to the automaton, instead of the instances themselves their *representatives* are used. See 7.5.1 for further reference. This assures that comparisons in automaton are done using Java `Object.equals()` reference comparison, but for each cluster member, its representative is only present in automaton. That results in cluster-equal behaviour. Example of prefix-tree automaton is in fig. 6.

7.5.4 Data structures - RegexpAutomaton

Class `RegexpAutomaton` extends 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 is in package `cz.cuni.mff.ksi.jinfer.twostep.processing.automatonmergingstate.regexping`. This is the same package where *`RegexpAutomatonSimplifier`* interface is defined. Implementation is nothing special - there is a copy constructor which creates regular expression automaton from ordinary automaton by enclosing each symbol on transition of original automaton in a `Regexp.TOKEN`.

7.5.5 AutomatonSimplifier module

`AutomatonSimplifier` module has its worker interface defined as following.

```
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 to it. Interface is defined as generic, implementations don't have to be. However, there is no reason not to make them generic. When implementation needs to present an automaton to user, it needs 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. Overloaded version is to inform the user about name of processed element when presenting automaton in *`AutoEditor`*. This version of interface will probably change in future.

This generality means that simplifier can not only be used to infer rules for elements, but possibly also to infer patterns strings for content model of attributes using the same algorithms. The actual implementation of this still waits for its developer to come. Implementations of *`AutomatonSimplifier`* are discussed in 7.10 and 7.11.

7.5.6 RegexpAutomatonSimplifier submodule

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

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

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

Once again we encounter `symbolToString` with the same purpose as before, if submodule would need to present the 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 7.13

7.5.7 MergeConditionTester submodule

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

One can implement ACO heuristics or MDL principle heuristics as *`AutomatonSimplifier`* submodule, and can use any of *`MergeConditionTesters`* provided.

7.6 NestedConcatention (RegularExpressionCleaner)

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. If for example an inference method produces unnecessary empty regexps in regexp tree such 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)

Allows the 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 then 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 7.12.

7.11 UserInteractive (AutomatonSimplifier)

We have created automaton simplifier which shows the user the input automaton, asking him to select states to merge. Then it merges them and asks user again.

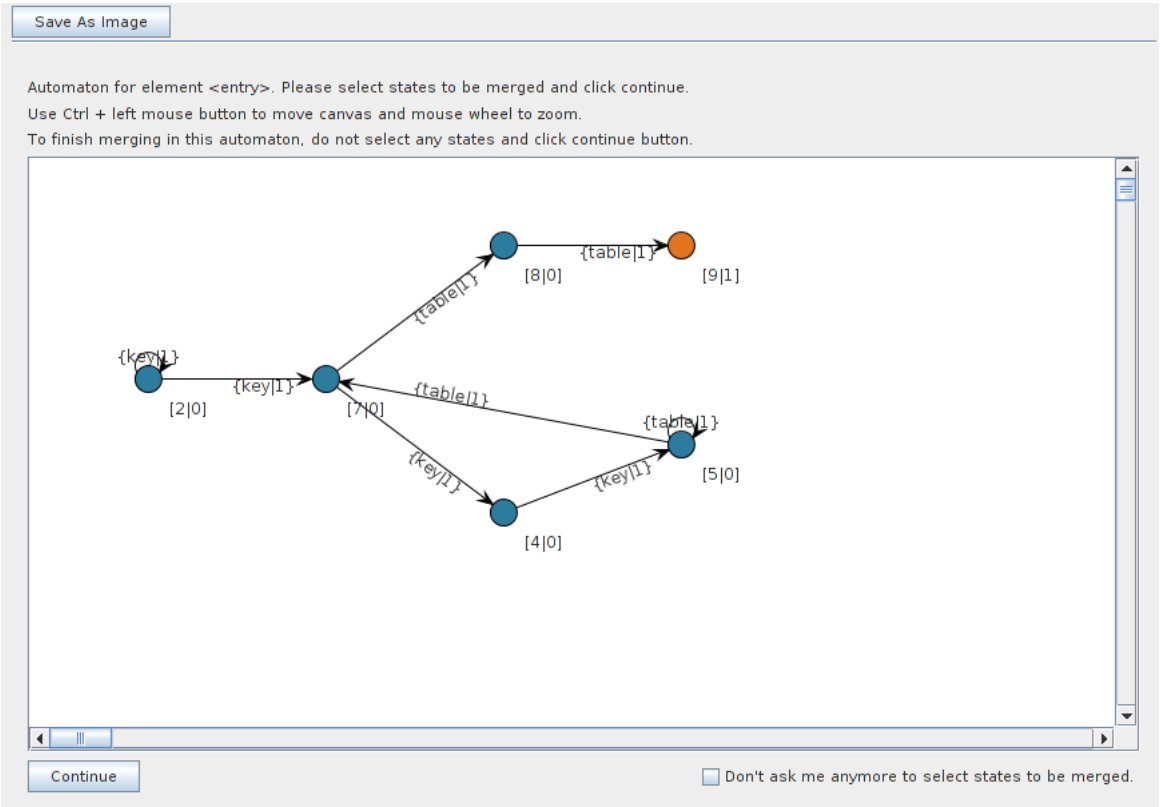


Figure 7: Screenshot of UserInteractive (AutomatonSimplifier).

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

We use *AutoEditor* module described in [KMS+b]. A screenshot of automaton visualization is in picture 7.

7.12 KHContext (MergeConditionTester)

We have implemented *k,h-context* state equivalence (see [Aho96]) in class `KHContext` (implements `MergeConditionTester` interface), which is used by `Greedy` to test mergeability of states. We don't use $k-grams$ algorithm from [Aho96], just a simple DFS.

7.13 StateRemoval (RegexpAutomatonSimplifier)

We are using state removal method (see [HW07]) to convert regexp automaton into equivalent regular expression. This is implemented in `StateRemoval` class (on fig. 8) which implements `RegexpAutomatonSimplifier` interface. Before the algorithm starts, states *superInitial* and *superFinal* are added to automaton, former with λ-transition to initial state. From all final states a 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 the final regular expression as symbol on it. This one is 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.

7.13.1 Data Structures - StateRemovalRegexpAutomaton

We extend `RegexpAutomaton` to obtain automaton that fits the state removal 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 into many technical details, we just mention method `removeState`, which removes given state from automaton. Transitions are handled in the following way.

1. Collapse all loop transitions 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.

The whole process is step-by-step illustrated in fig. 9.

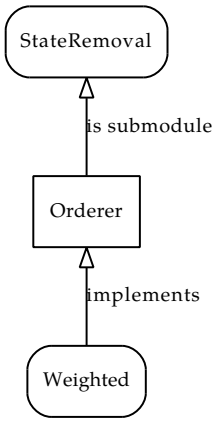


Figure 8: Modules of StateRemoval.

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

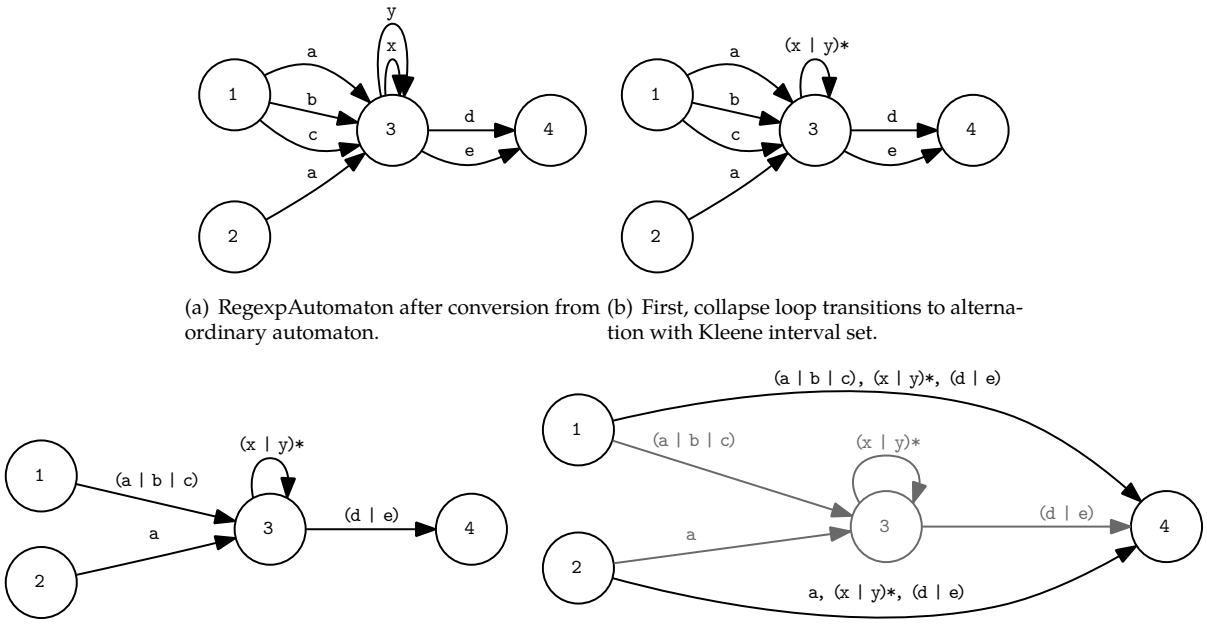
If you are willing to, you can replace each of three submodules of *`TwoStepSimplifier`*. It may be useful to replace the default *`ClustererWithAttributes`* implementation with a clever one. And regular expression cleaning will be probably the aim of new implementations, too. We recommend replacing other parts at lower levels, however. You will need to replace *`ClusterProcessor`* only if you want to write inference 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 probably 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 suggested in [HW07] may lead to shorter resulting regular expressions.

Whole TwoStep submodules structure is in fig. 10.



(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.

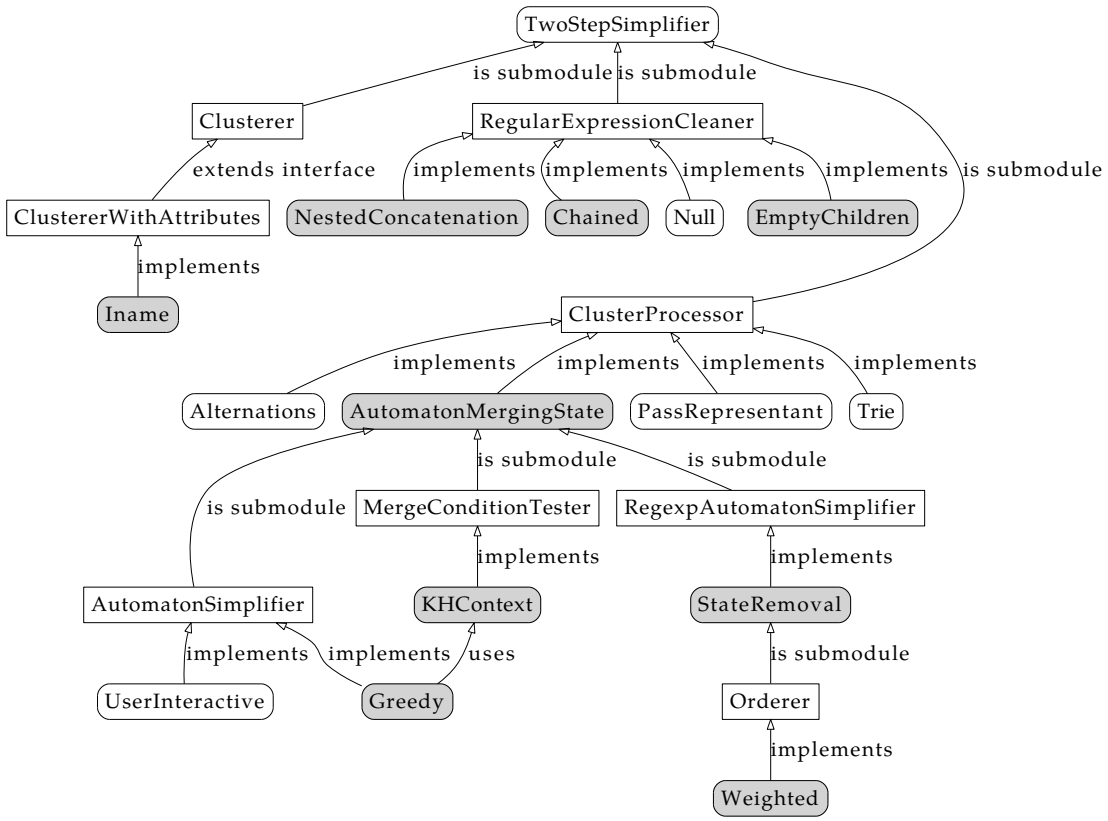


Figure 10: Modules of TwoStep simplifier and their submodules. Filled classes are default selection (best of).

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.

[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`.

[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 Vyhňanovská. Automatic construction of an xml schema for a given set of xml documents.

[wik] Regular expression. `http://en.wikipedia.org/wiki/Regular_expression`.