jInfer AutoEditor 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, specifically alter displaying of automata .

Responsible developer	Mário Mikula
Required tokens	`org.openide.windows.WindowManager`
Provided tokens	none
Module dependencies	Base JUNG

Public packages	`cz.cuni.mff.ksi.jinfer.autoeditor`
	`cz.cuni.mff.ksi.jinfer.autoeditor.automatonvisualizer`
	`cz.cuni.mff.ksi.jinfer.autoeditor.automatonvisualizer.layouts`
	`cz.cuni.mff.ksi.jinfer.autoeditor.gui.component`

1 Introduction

This is an implementation of a an automaton editor. Using JUNG library, it provides an API to display and modify automata in an interactive mode. For more information on JUNG library, see [jun].

UML diagrams shown in this document are simplified to keep them readable. Simplification involves removing unimportant class members, which are not mentioned in the text. Also, removing members of non-jInfer classes and truncating string of inheritance that are not important for this document. For example, `VisualizationViewer` class from JUNG library has many methods and does not extend `JPanel` class directly. However, in the UML diagram, it does not have any methods and is shown to extend `JPanel` directly, because it is sufficient for understanding of this document.

2 Structure

Structure of *`AutoEditor`* can be divided into following four main parts.

API API for displaying automaton in GUI.

Base classes Classes providing basic functionality that can be extended and combined to achieve desired visualization of an automaton.

Derived classes Classes derived from the base classes that are used in existing modules and simultaneously serve as examples.

Layout creation System for creating `Layouts`.

First, `Layouts` and use of base classes to create a visualization of an automaton will be described.

In case a generic class has a type parameter `T`, this has to be the same `T` that is used to parametrize jInfer automata. JUNG classes usually need a type parameter for a state and for an edge. These should be then `State<T>` (for a state) and `Step<T>` (for edge). Refer to [KMS+a] for more information on these classes and see 2.4.1 and 2.4.2 for source code examples.

2.1 Layout

Layout is a JUNG interface responsible primarily for the representation of an automaton and positions of its states. JUNG library provides several implementation of Layout interface. However, because none of them is convenient for automatic automaton displaying, *AutoEditor* provides two additional implementations. Layout by *Julie Vyhnanovska*, used in her master thesis and Layout which is using external *Graphviz* software.

Class providing creation of Layout instances is named LayoutHelperFactory.

2.1.1 Vyhnanovska Layout

As mentioned above, this Layout was implemented by *Julie Vyhnanovska* as a part of her master thesis (see [Vyh]). It positions automaton states on a square grid. This Layout gives good results for relatively small automata (about 10 states or less) but for larger ones, the results are quite disarranged and confusing.

Source code resides in package cz.cuni.mff.ksi.jinfer.autoeditor.automatonvisualizer.layouts.vyhnanovska.

2.1.2 Graphviz Layout

Graphviz Layout uses *Graphviz*, a third-party graph visualization software (see [gra]), to create positions of automaton states. To use this Layout, *Graphviz* has to be installed and path to *dot* binary has to be set in options. This Layout gives nice results even for large automata.

Automaton is transformed to the *dot* representation and the *dot* binary is invoked with this representation on input. It processes the automaton and writes positions of its states to the standard output. The output is then parsed and an instance of Layout is created using these positions.

Source code resides in package cz.cuni.mff.ksi.jinfer.autoeditor.automatonvisualizer.layouts.graphviz.

2.1.3 LayoutHelperFactory

In project properties, it is possible to select a Layout to be used to display automata. LayoutHelperFactory is a class providing just one static method responsible for creating instances of Layouts according to a selection in project properties.

This method has the following signature.

```
public static <T> Layout<State<T>, Step<T>> createUserLayout(
  Automaton<T> automaton,
  Transformer<Step<T>, String> edgeLabelTransformer)
```

The first argument is an automaton to create the layout from. The second, is a transformer to transform an instance of automaton edge to its string representation, required by the Graphviz Layout.

Source code resides in package cz.cuni.mff.ksi.jinfer.autoeditor.automatonvisualizer.layouts.

2.1.4 How to create a new Layout

New Layouts can be implemented using the modular system. For information on system of modules, see [KMS+a]. To create a new implementation of Layout interface, it is necessary to create a new class implementing LayoutFactory interface (package cz.cuni.mff.ksi.jinfer.autoeditor.automatonvisualizer.layouts) and annotate it by the following code.

```
@ServiceProvider(service = LayoutFactory.class)
```

Created implementation will be shown in project properties in the Layout selection.

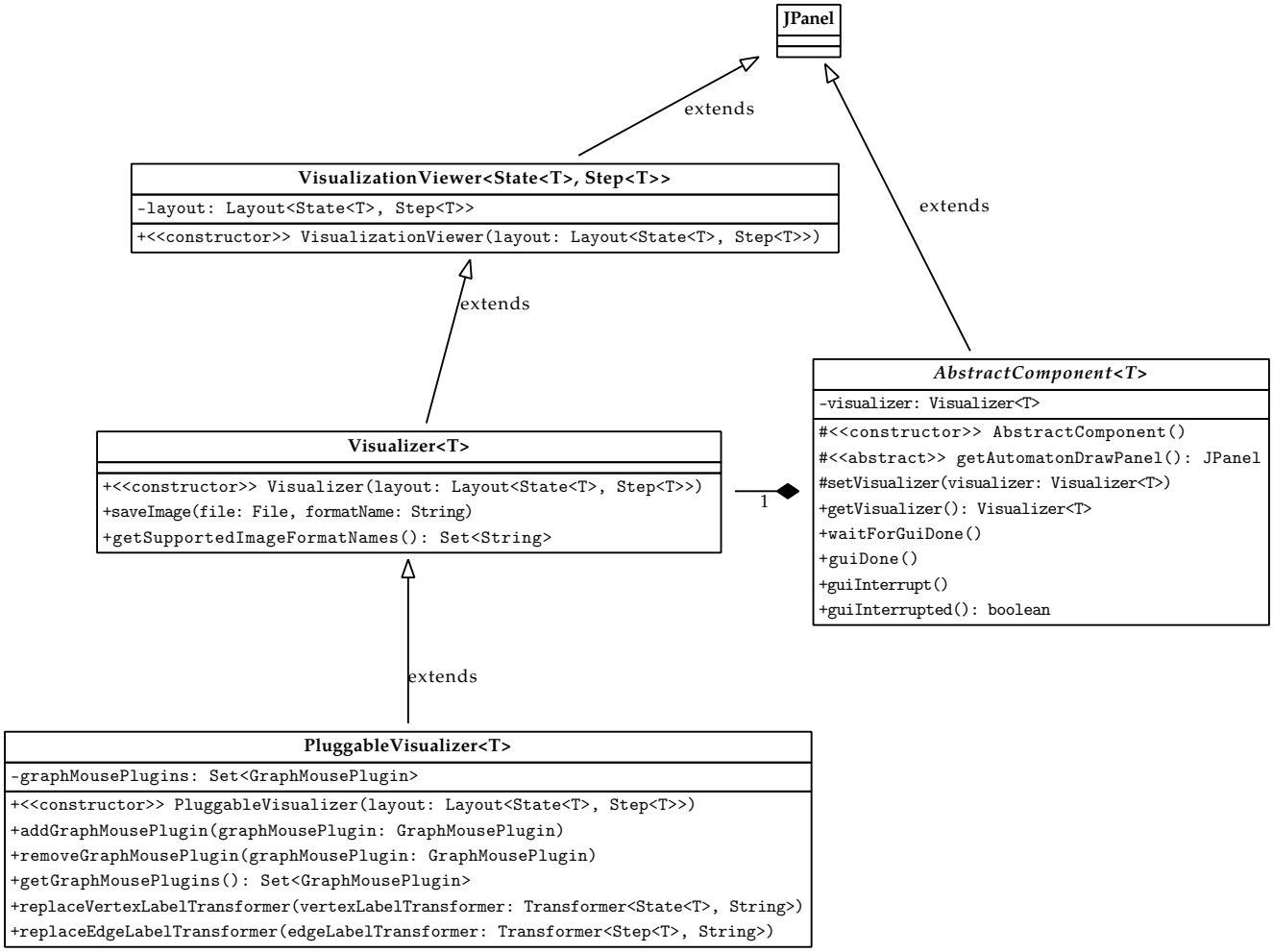

Figure 1: Class diagram for the base classes.

2.2 Base classes

This section describes classes implementing basic common functionality that are supposed to be extended to create a new suitable visualization of automata for a particular method of inference. The new visualization may involve a brand new GUI panel with buttons of various functions. For example, user interaction, like selecting states or edges, and others.

Two main classes representing the visualization of an automaton are `Visualizer` and `AbstractComponent`. `Visualizer` is a graphical representation of automaton and `AbstractComponent` is a panel (extends `JPanel`) containing the `Visualizer` which will be displayed in GUI.

2.2.1 Visualizer

`Visualizer` class in `cz.cuni.mff.ksi.jinfer.autoeditor.automatonvisualizer` package extends JUNG `VisualizationViewer` class thus it inherits all its methods and adds support for saving contained automaton to an image file. Relevant methods are `saveImage()` and `getSupportedImageFormatNames()`. However, to save an image of an automaton, it is not necessary to call this methods directly. *`AutoEditor`* GUI contains button to save an image of displayed automaton. For information on how to to this, see 2.5.

Constructor has one argument, an instance of `Layout` interface created from an automaton, typically by `Layout-HelperFactory` (see 2.1.3).

2.2.2 PluggableVisualizer

`PluggableVisualizer` class in `cz.cuni.mff.ksi.jinfer.autoeditor.automatonvisualizer` package is an extension of `Visualizer` class, which primarily provides an easy way to plug *graph mouse plugins*.

*Graph mouse plugin*s are classes implementing JUNG `GraphMousePlugin` interface and their purpose is to enhance `Visualizer` with mouse support.

By default, instance of `PluggableVisualizer` is constructed with two plugins enabled. They are `ScalingGraphMousePlugin`, providing zooming functionality, and `TranslatingGraphMousePlugin`, providing translation of the displayed automaton in the x and y direction. In most cases these plugins are useful, but otherwise, they can be removed using methods `getGraphMousePlugins()` and `removeGraphMousePlugin()`.

Public (not inherited) methods of `PluggableVisualizer` are the following. Their purpose is clear from their names, for details please see their JavaDoc.

- `addGraphMousePlugin()`
- `removeGraphMousePlugin()`
- `getGraphMousePlugins()`
- `replaceVertexLabelTransformer()`
- `replaceEdgeLabelTransformer()`

2.2.3 AbstractComponent

`AbstractComponent` class in `cz.cuni.mff.ksi.jinfer.autoeditor.gui.component` package is a representation of GUI panel containing an instance of `Visualizer` class for some automaton. It is inherited from `JPanel` class, thus provides JPanel's method and behaviour. In addition, it provides the following methods.

- `setVisualizer()` - Protected setter of `Visualizer`. It is defined protected because extensions of this class may want to declare this setter with argument type that extends `Visualizer` class.
- `getVisualizer()` - Getter of `Visualizer`.
- `waitForGuiDone()` - Suspends its thread until method `guiDone` is called on this instance. Do not call this method directly, it is called by *`AutoEditor`*. For more information, see 2.2.4.
- `guiDone()` - Wakes up this instance from a suspended state. For detailed description, see 2.2.4.
- `guiInterrupt()` - Called when *`AutoEditor`*'s tab is closed to propagate information about terminating of inference to a caller of *`AutoEditor`*. There is no need to call this method directly.
- `guiInterrupted()` - Checks if *`AutoEditor`* GUI was terminated by `guiInterrupt()` method or regularly (`guiDone()` method or GUI was not waiting for user interaction). Again, there is no need to call this method directly, it is called by *`AutoEditor`*. For details, see 2.5.

Beside those methods, `AbstractComponent` has one abstract method, named `getAutomatonDrawPanel()`.

Purpose of this class is to be extended to create own GUI panel, which displays some automaton using a supplied instance of `Visualizer`. Method `getAutomatonDrawPanel()` is meant to be overridden to return an instance of `JPanel`, in which the `Visualizer` is to be drawn.

Programmer implementing an extension of `AbstractComponent` is not forced to place the `Visualizer` on his own. It is just needed to create `JPanel` and define `getAutomatonDrawPanel()` method to return this `JPanel`. *`AutoEditor`* will take care of placing and displaying the `Visualizer` in the returned `JPanel`.

`Visualizer` is not set in constructor, because it is often desired to subsequently display several different automata (`Visualizers`) in the same panel. In this case, it is not needed to create a new instance of `AbstractComponent` for each `Visualizer`, but subsequently calling `setVisualizer()` method using one instance of `AbstractComponent`.

2.2.4 AbstractComponent user interactivity support

If some kind of user interactivity is desired, `AbstractComponent` is the right place to implement it.

To display the component in GUI and wait for some user action, we use `waitForGuiDone()` method. After displaying the component, calling this method suspends the running thread, thus code execution of a caller module is stopped at the place of this call. However, since GUI is ran in another thread, the user is able to interact with the panel (component).
Do not call method `waitForGuiDone()` directly! It is called by *`AutoEditor`* when displaying the component by *`AutoEditor`* *API* method named `drawComponentAndWaitForGUI()`. For more information on *`AutoEditor`* *API*, see 2.3.

Method important to the programmer extending `AbstractComponent` is called `guiDone()`. This method wakes up the thread suspended in `waitForGuiDone()` method and the programmer is responsible for calling it. Typically, it is called upon some user action like a button click, vertex pick or other GUI event.
After calling of `guiDone()` method, code execution of the caller module is resumed, and holding instance of the `AbstractComponent` it is able to retrieve results of user interaction, saved in its state.

For examples of user-interactive component, see 2.4.1 and 2.4.2.

2.3 API

`AutoEditor` *API* is pretty simple. Package `cz.cuni.mff.ksi.jinfer.autoeditor` contains class `AutoEditor` with three public static methods.

- `drawComponentAsync()` - Displays given `AbstractComponent` asynchronously in a GUI thread and immediately returns. Use this method to just display automaton, without any user interaction and without waiting for any external event. This method does not support it.
- `drawComponentAndWaitForGUI()` - Displays given `AbstractComponent` in a GUI thread, while the caller thread is suspended until `guiDone()` method of `AbstractComponent()` is called. This method can wait for GUI events and thus is convenient for user interaction. How to achieve this is described in detail in 2.2.4.
- `closeTab()` - Closes *`AutoEditor`*'s GUI tab and interrupts inference, if running.

For examples of API usage, see 2.4.1 and 2.4.2.

2.4 Derived classes

This section describes classes derived from the base classes used in *`Two Step Simplifier`* module to display automata. This classes may also serve as examples of base classes extensions.

2.4.1 StatePickingComponent

`StatePickingComponent` (extensions of `abstractComponent`) alongside with `StatePickingVisualizer` (extension of `PluggableVisualizer`) provides possibility to pick one automaton state in GUI and immediately return to the calling code, which can retrieve the picked state.

`StatePickingVisualizer` is a trivial extension of `PluggableVisualizer`. It has no additional methods. Its constructor has several additional arguments. Their description follows.

```
public StatePickingVisualizer(
  Layout<State<T>, Step<T>> layout,
  Transformer<Step<T>, String> edgeLabelTransformer,
  StatePickingComponent<T> component,
  State<T> superinitialState,
  State<T> superfinalState)
```

- `layout` - Instance of `Layout` which will be provided to the constructor of parent `Visualizer` class.

- `edgeLabelTransformer` - Transformer to convert automaton edges to their string representations.
- `component` - Instance of `StatePickingComponent` to call `guiDone()` method on, when a state is picked.
- `superinitialState` and `superfinalState` - Superinitial and superfinal states as we want to distinguish these states in displayed automaton.

Upon construction, it just adds `VertexPickingGraphMousePlugin` to the plugins of `PluggableVisualizer`. Purpose of this plugin is to allow user to pick some state of automaton and then call `guiDone()` method on instance of `AbstractComponent`. For description of `guiDone()` method, see 2.2.3.

`StatePickingComponent` provides the following additional methods.

- `setVisualizer()` - Setter for `Visualizer` restricted to accept instances of `StatePickingVisualizer` class to prevent misuse.
- `getPickedState()` - After displaying the component using `drawComponentAndWaitForGUI()` *API* method (see 2.3), this method retrieves the user picked automaton state.
- `setLabel()` - Sets text of a component label. The label can be used to communicate some information to user, for example instructions.

Source codes of `StatePickingComponent` resides in package `cz.cuni.mff.ksi.jinfer.autoeditor.gui.component`, `StatePickingVisualizer` in `cz.cuni.mff.ksi.jinfer.autoeditor.automatonvisualizer` and `VertexPickingGraphMousePlugin` in `cz.cuni.mff.ksi.jinfer.autoeditor.automatonvisualizer.graphmouseplugins`.

Example of usage follows.

```
Transformer<Step<Regexp<T>>, String>
  transformer = new Transformer<Step<Regexp<T>>, String>() {

    @Override
    public String transform(Step<Regexp<T>> step) {
      StringBuilder sb = new StringBuilder();
      sb.append("{");
      sb.append(symbolToString.toString(step.getAcceptSymbol()));
      sb.append("|");
      sb.append(String.valueOf(step.getUseCount()));
      sb.append("}");
      return sb.toString();
    }
  };

State<Regexp<T>> removeState;
StatePickingComponent<Regexp<T>> component = new StatePickingComponent<Regexp<T>>();
Layout<State<T>, Step<T>> layout =
  LayoutHelperFactory.createUserLayout(automaton, transformer);
StatePickingVisualizer<Regexp<T>>
  visualizer = new StatePickingVisualizer<Regexp<T>>(layout,
                                                     transformer,
                                                     component,
                                                     automaton.getSuperInitialState(),
                                                     automaton.getSuperFinalState());
component.setVisualizer(visualizer);

do {
  AutoEditor.drawComponentAndWaitForGUI(component);
  removeState = component.getPickedState();

  if ((removeState.equals(automaton.getSuperFinalState()))
```

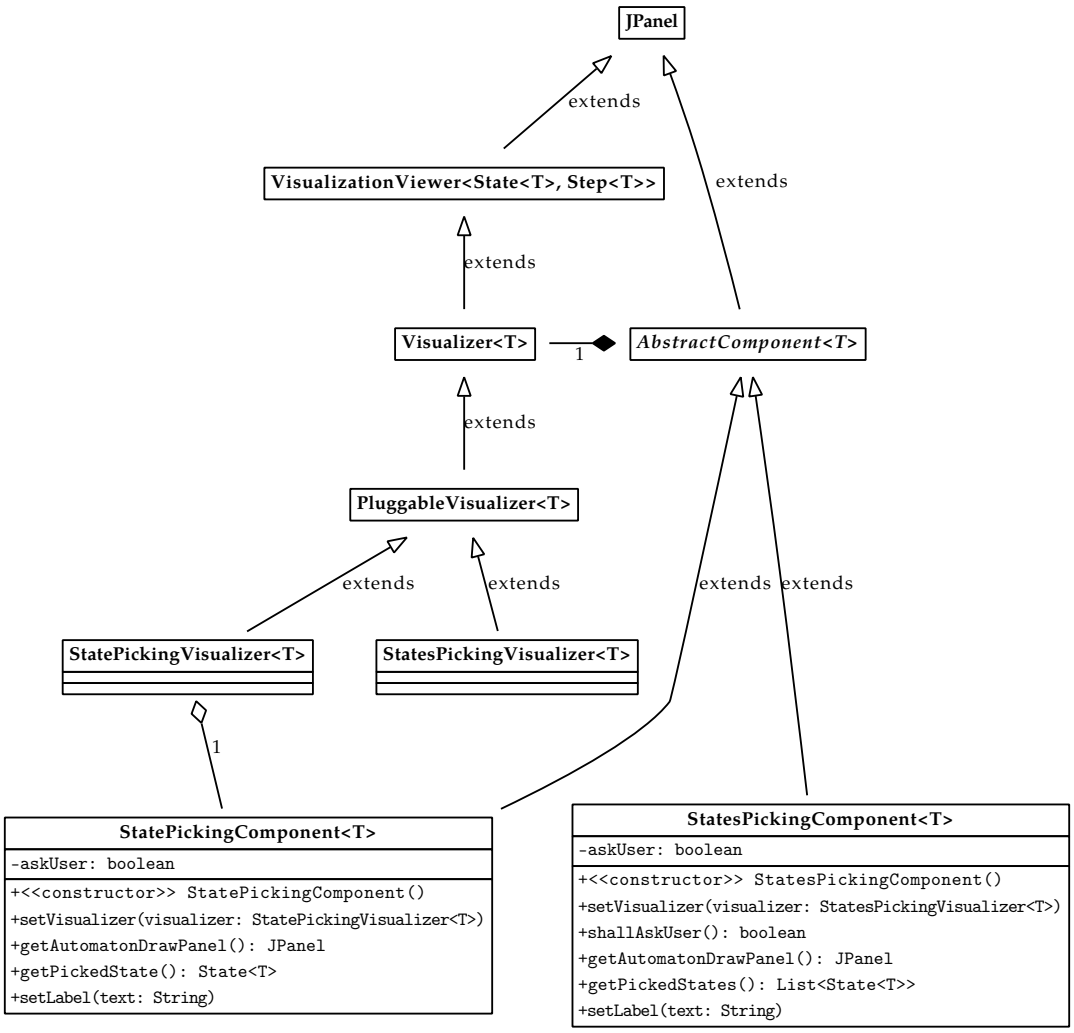

Figure 2: Class diagram for the derived classes, not showing members of classes from figure 1 to keep this diagram simple. Also not showing signatures of constructors of StatePickingVisualizer and StatePickingVisualizer for their excessive length.

```
        || (removeState.equals(automaton.getSuperInitialState()))) {
      component.setLabel("Do not select superInitial and superFinal states.");
      continue;
    }
    return removeState;
  } while (true);
```

2.4.2 StatesPickingComponent

Classes `StatePickingComponent` and `StatesPickingVisualizer` are similar to the classes described in the previous section. Purpose of these is to provide picking of multiple automaton states.

`StatesPickingVisualizer` is an extension of `PluggableVisualizer` and upon its construction it adds `VerticesPickingGraphMousePlugin` to the graph mouse plugins. With this plugin used, user can pick and unpick automaton states separately, or pick several states at once by dragging rectangular selection box over states. Main difference compared to the `VertexPickingGraphMousePlugin` is that `VerticesPickingGraphMousePlugin` does not call component's `guiDone()` method.

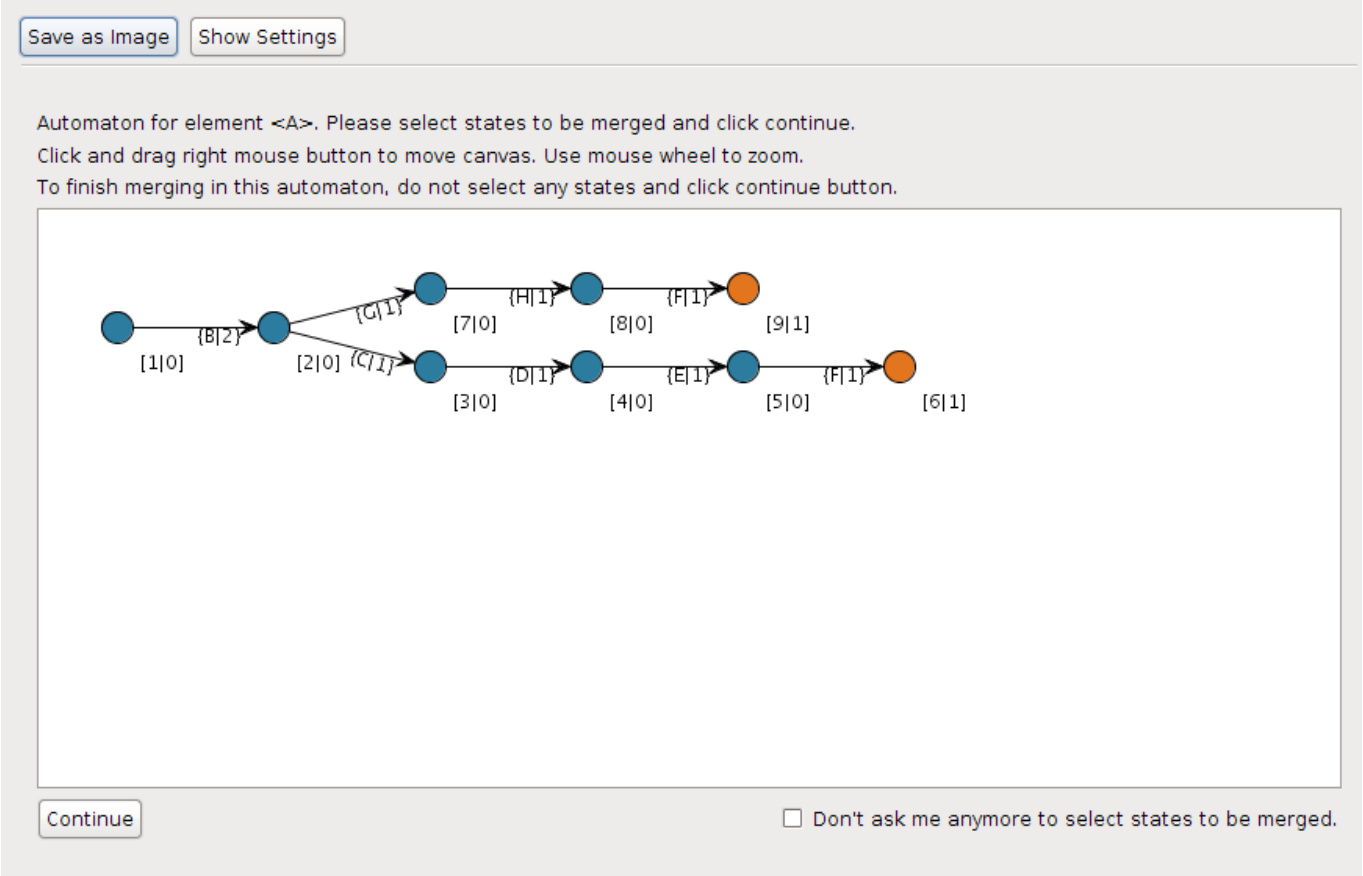

Figure 3: Screenshot of *AutoEditor* GUI

Signature of constructor and description of arguments follow.

```
public StatesPickingVisualizer(
  Layout<State<T>, Step<T>> layout,
  Transformer<Step<T>, String> edgeLabelTransformer)
```

- `layout` - Instance of `Layout` which will be provided to the constructor of parent `Visualizer` class.
- `edgeLabelTransformer` - Transformer to convert automaton edges to their string representations.

`StatesPickingComponent` is almost the same as `StatePickingComponent` but it contains two extra GUI controls. "Continue" button and "Don't ask me anymore to select states to be merged" checkbox. The button is supposed to be clicked when user picked all desired states and it will cause calling of component's `guiDone()` method (see 2.2.3). State of the checkbox can be retrieved by the calling code and the caller is supposed to stop showing this component in the current inference process.

Methods of `StatesPickingComponent` are the following.

- `setVisualizer()` - Setter for `Visualizer` restricted to accept instances of `StatesPickingVisualizer` class to prevent misuse.
- `shallAskUser()` - Retrieves state of the checkbox.
- `getPickedStates()` - After displaying the component using `drawComponentAndWaitForGUI()` *API* method (see 2.3), this method retrieves the list of user picked automaton states. In the case that none of automaton states was picked, it returns an empty list.
- `setLabel()` - Sets text of the component label. The label can be used to communicate some information to the user, for example instructions.

Source codes of `StatesPickingComponent` reside in package `cz.cuni.mff.ksi.jinfer.autoeditor.gui.component`, `StatesPickingVisualizer` in `cz.cuni.mff.ksi.jinfer.autoeditor.automatonvisualizer` and `VerticesPickingGraphMousePlugin` in `cz.cuni.mff.ksi.jinfer.autoeditor.automatonvisualizer.graphmouseplugins`.

Example of usage follows.

```
Automaton<T> simplify(Automaton<T> inputAutomaton,
                      SymbolToString<T> symbolToString,
                      String elementName) {
  if (!askUser) {
    return inputAutomaton;
  }

  List<State<T>> mergeLst;
  Transformer<Step<T>, String> transformer = new Transformer<Step<T>, String>() {

    @Override
    public String transform(Step<T> step) {
      StringBuilder sb = new StringBuilder();
      sb.append("{");
      sb.append(symbolToString.toString(step.getAcceptSymbol()));
      sb.append("|");
      sb.append(String.valueOf(step.getUseCount()));
      sb.append("}");
      return sb.toString();
    }
  };

  Boolean selectTwo = false;
  do {
    Layout<State<T>, Step<T>>
      layout = LayoutHelperFactory.createUserLayout(inputAutomaton, transformer);
    StatesPickingVisualizer<T> visualizer =
        new StatesPickingVisualizer<T>(layout, transformer);

    StatesPickingComponent<T> panel = new StatesPickingComponent<T>();
    panel.setVisualizer(visualizer);
    if (selectTwo) {
      panel.setLabel("Automaton for element <" + elementName
        + ">. Please select states to be merged and click continue."
        + " Select at least 2 states.");
    } else {
      panel.setLabel("Automaton for element <" + elementName
        + ">. Please select states to be merged and click continue.");
    }

    AutoEditor.drawComponentAndWaitForGUI(panel);
    mergeLst = panel.getPickedStates();

    if ((!BaseUtils.isEmpty(mergeLst)) && (mergeLst.size() >= 2)) {
      inputAutomaton.mergeStates(mergeLst);
      selectTwo = false;
    } else if (mergeLst.size() < 2) {
      selectTwo = true;
    }

    if (!panel.shallAskUser()) {
```

```
        askUser = false;
        break;
      }
    } while (!BaseUtils.isEmpty(mergeLst));
    return inputAutomaton;
  }
```

2.5 GUI

As shown in figure 3, *AutoEditor*'s panel is placed in a tab of the editor window. It consists of two buttons, horizontal line below them and a panel to place an extension of `AbstractComponent` (see 2.2.3). Class representing this panel is called `AutoEditorTopComponent` and resides in `cz.cuni.mff.ksi.jinfer.autoeditor.gui.topcomponent` package.

Buttons have labels "Save as Image" and "Show Settings". Pushing the first one will raise a dialog box to save currently displayed automaton to an image file. Set of supported image formats depends on installed JRE. The second one will open an *AutoEditor* tab in NetBeans options. For description of the settings, see section 2.6.

2.6 Settings

All settings provided by *AutoEditor* are NetBeans-wide. The options panel along with all the logic is in the `cz.cuni.mff.ksi.jinfer.autoeditor.options` package. Available options include setting the color of the background, colors and shapes of some special types of automaton states.

To have some effect, these settings need to be implemented by extensions of `Visualizer` class. For examples, please see the source codes of `StatePickingVisualizer` and `StatesPickingVisualizer` classes.

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

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

[xml] Xml validation api. `http://download.oracle.com/docs/cd/E17802_01/webservices/webservices/docs/1.6/api/javax/xml/validation/package-summary.html`.