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 *AutoEditor*. Using JUNG library, it provides an API to display and user interactively modify automata, thus the process of inference can be easily made user interactive.

TODO spomenut vsade packages
TODO UML diagramy su zjednodusene z dovodu prehladnosti

2 Structure

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

- API - API to display 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 of creating `Layouts`.

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

2.1 Layout

`Layout` is a JUNG interface responsible primarily of representation of automaton and positions of its states. JUNG library provides several implementation of `Layout` interface. However, 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. TODO odkazy v predchadzajucej vete.

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. TODO link? It positions automaton states to a square grid. This `Layout` gives good results for relatively small automata (about 10 states of less) but for larger ones, the results are quite disarranged and confused.

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

2.1.2 Graphviz Layout

`Graphviz Layout` uses *Graphviz*, third-party graph visualization software, to create positions of automaton states. TODO link? 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 on large automata.

TODO ako presne ziskava pozicie z graphvizu
Source codes 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 class, providing just one static method, responsible for creating instances of `Layouts` according to a selection in project properties.

This method has following signature.

```
public static <T> Layout<State<T>, Step<T>> createUserLayout(final Automaton<T> automaton, final Transform
```

The first argument is a automaton to create a layout from. The second is transformer to transform an instance of automaton edge to its string representation, required by the `Graphviz Layout`.
Source codes resides in package `cz.cuni.mff.ksi.jinfer.autoeditor.automatonvisualizer.layouts`.

2.1.4 How to create a new Layout

`Layouts` can be implemented using the modular system. To create a new implementation of `Layout` interface, it is needed to create a new class implementing `LayoutFactory` interface (package `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.

System of modules is described in detail in TODO ref.

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 with various functions, user interaction like selecting states or edges and others.

Main two classes representing visualization of 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. TODO obrazok ako AC dedi od JPanelu a obsahuje Visualizer.

2.2.1 Visualizer

`Visualizer` class extends JUNG `VisualizationViewer` class thus it provides all its methods and adds support for saving contained automaton to an image file. Responsible methods are `saveImage()` and `getSupportedImageFormatNames()`. However, to save an image of 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 TODO ref.

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

2.2.2 PluggableVisualizer

`PluggableVisualizer` class is 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, and `TranslatingGraphMousePlugin`, providing translating the displayed automaton in the x and y direction. In the most cases, these plugins are useful but if they are not wanted they can be removed using methods `getGraphMousePlugins()` and `removeGraphMousePlugin()`.

TODO obrazok ako PluggableVisualizer dedi od Visualizeru
Public (not inherited) methods of `PluggableVisualizer` are the following. Their purpose is clear from their names, for details see their JavaDoc.

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

2.2.3 AbstractComponent

`AbstractComponent` class 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()` - Setter of `Visualizer`.
- `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 called this method directly.
- `guiInterrupted()` - Checks if *`AutoEditor`* GUI was terminated by interrupt or regularly. Also, there is no need to call this method directly, it is called by *`AutoEditor`*. For details, see TODO ref GUI.

Besides 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 returns 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 its own. It is just needed to create `JPanel` and define `getAutomatonDrawPanel()` method to return this `JPanel`. *`AutoEditor`*

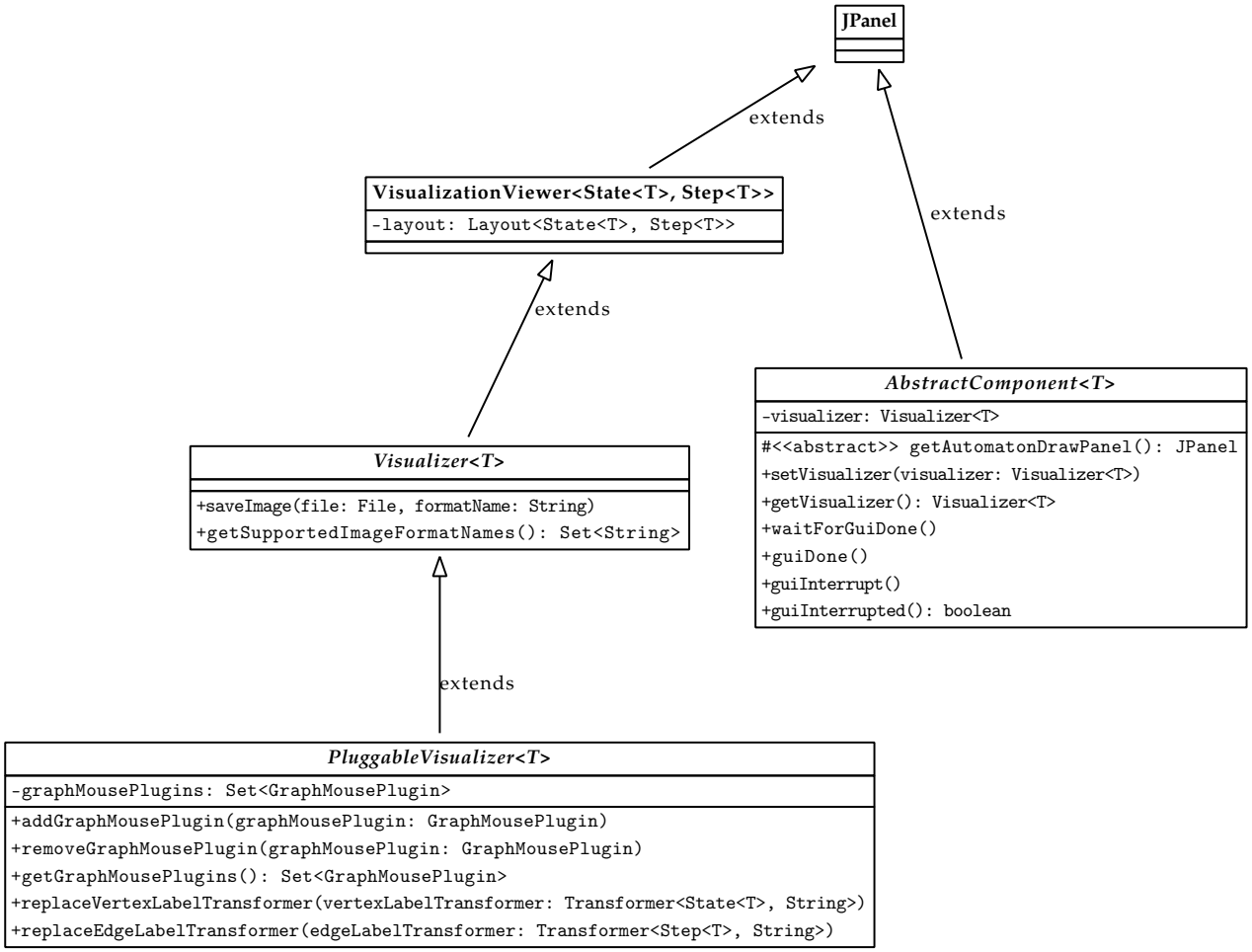

Figure 1: Class diagram for the base classes.

will take care of placing and displaying the `Visualizer` in the `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 new instance of `AbstractComponent` for each `Visualizer`, but subsequently call `setVisualizer()` method using one instance of `AbstractComponent`.

2.2.4 AbstractComponent user interactivity support

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

To display the component in GUI and wait for some user action, method `waitForGuiDone()` is used. After displaying the component, calling of this method suspend running thread thus code execution of a caller module is stopped at the place of this call. However, GUI is ran in another thread, 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 TODO ref.

Method important for a programmer extending `AbstractComponent` is named `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 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 TODO ref.

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 these.
- `drawComponentAndWaitForGUI()` - Displays given `AbstractComponent` in a GUI thread and a caller thread is suspended until `guiDone()` method of `AbstractComponent()` is called. This method can wait for GUI events thus is convenient for user interaction. How to achieve it 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 TODO ref.

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 extending the base classes.

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 trivial extension of `PluggableVisualizer`. It has no additional methods. Its constructor has additional two arguments, superinitial and superfinal states as we want to distinguish these states in displayed automaton. And 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 TODO ref.

`StatePickingComponent` provides the following additional methods.

- `getPickedState()` - After displaying the component using `drawComponentAndWaitForGUI()` *API* method (see TODO ref), 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 `VertexPickingGraphMousePl` in `cz.cuni.mff.ksi.jinfer.autoeditor.automatonvisualizer.graphmouseplugins`.

Example of usage follows.

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

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

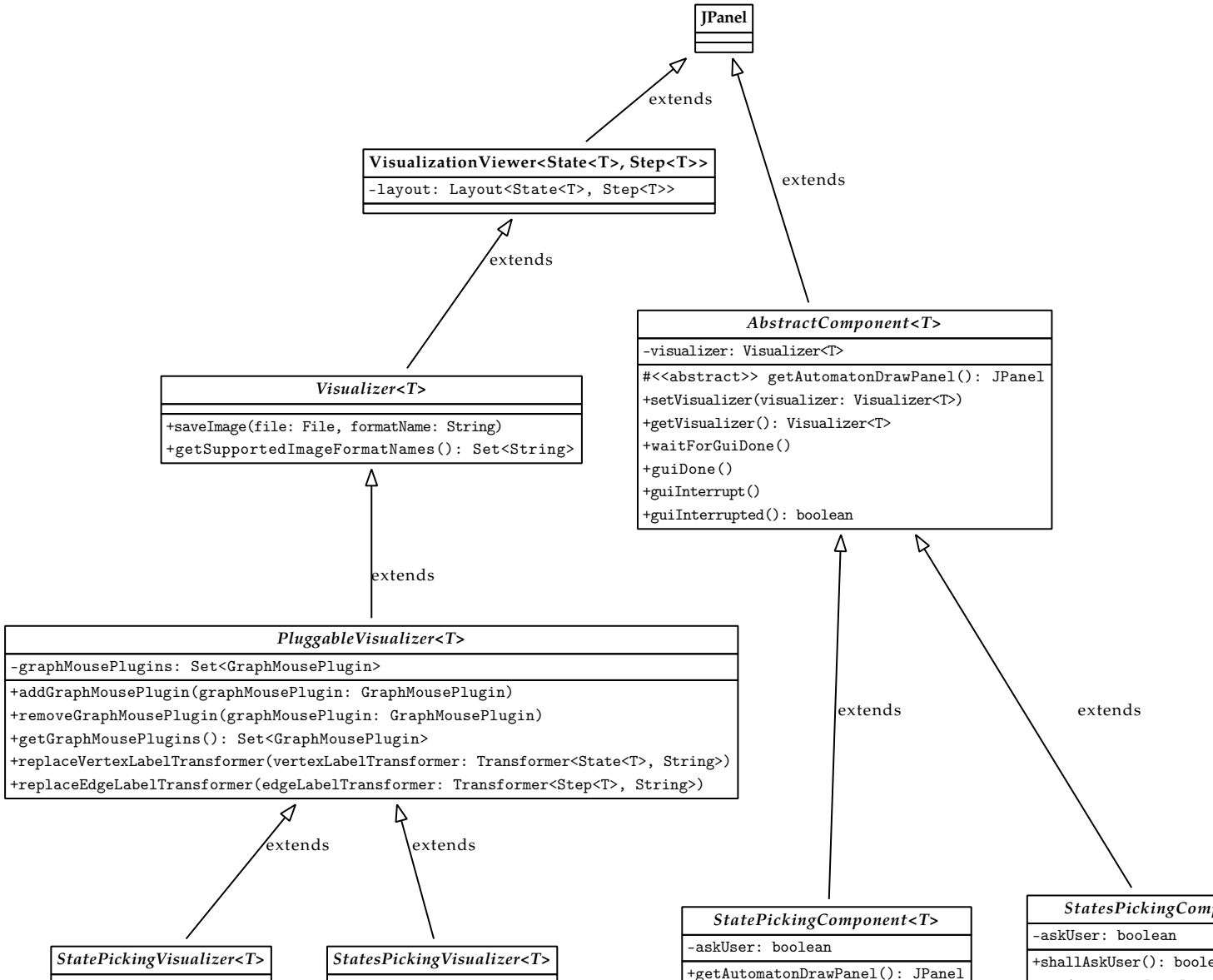

Figure 2: Class diagram for the derived classes.

```
      sb.append("}");
      return sb.toString();
    }
  };

  State<Regexp<T>> removeState;
  final StatePickingComponent<Regexp<T>> component = new StatePickingComponent<Regexp<T>>();
  final StatePickingVisualizer<Regexp<T>> visualizer = new StatePickingVisualizer<Regexp<T>>(LayoutHelpe
  component.setVisualizer(visualizer);

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

    if ((removeState.equals(automaton.getSuperFinalState())) || (removeState.equals(automaton.getSuperIn
      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 extension of `PluggableVisualizer` and upon its construction it adds `VerticesPickingGraphM` 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 `VertexPickingGraphMousel` is `VerticesPickingGraphMousePlugin` does not call component's `guiDone()` method.

`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 merger' checkbox. The button is supposed to be pushed when user picked all desired states and it will cause calling of component's `guiDone()` method (see TODO ref). 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.

- `shallAskUser()` - Retrieves state of the checkbox.
- `getPickedStates()` - After displaying the component using `drawComponentAndWaitForGUI()` *API* method (see TODO ref), this method retrieves the list of user picked automaton states. In case that none of automaton states was picked, it returns an empty list.
- `setLabel()` - Sets text of a component label. The label can be used to communicate some information to user, for example instructions.

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

Example of usage follows.

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

  List<State<T>> mergeLst;
```

```
    final 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 {
      final StatesPickingVisualizer<T> visualizer = new StatesPickingVisualizer<T>(LayoutHelperFactory.cre

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

      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 at figure 3, *AutoEditor*'s panel is placed in a tab in the main NetBeans window. It consists of two buttons, horizontal line below them and below the line, there is panel to place an extension of `AbstractComponent` (see TODO ref).

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 *AutoEditor* tab in NetBeans options. For description of the settings, see **??**.

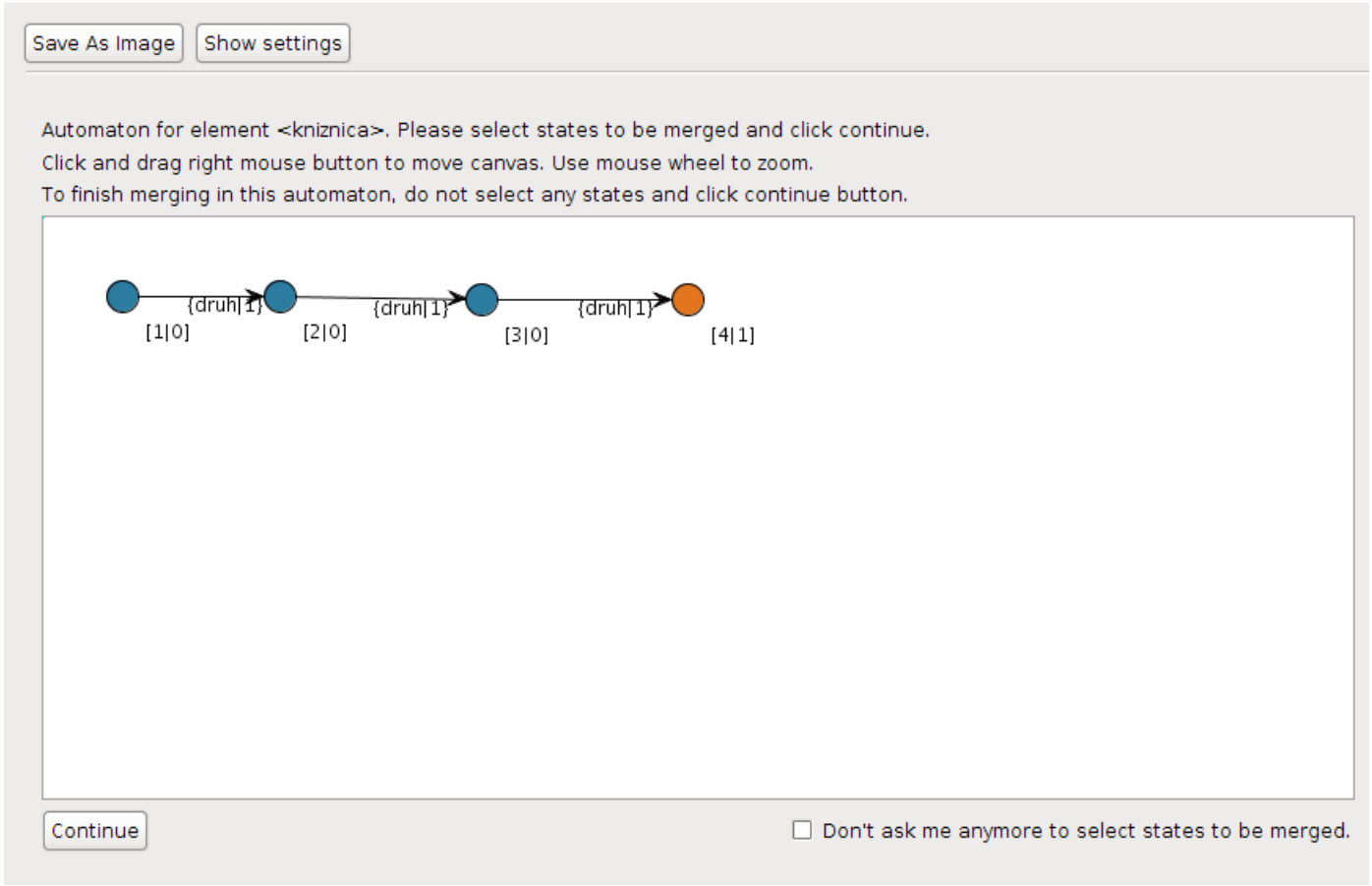

Figure 3: Screenshot of *AutoEditor* GUI

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, see 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`.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison-Wesley, 2001.

[HW07] Yo-Sub Han and Derick Wood. Obtaining shorter regular expressions from finite-state automata. *Theor. Comput. Sci.*, 370(1-3):110–120, 2007.

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

[KMS+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`.

[VMP08] Ondřej Vošta, Irena Mlýnková, and Jaroslav Pokorný. Even an ant can create an xsd. In *DASFAA'08: Proceedings of the 13th international conference on Database systems for advanced applications*, pages 35–50, Berlin, Heidelberg, 2008. Springer-Verlag.

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