

Contents

1 Introduction

The goal of the Yakindu Eclipse Plugin is to support software developers mainly in hardware driven projects to implement, verify, simulate and test state machines.

The creation of a state machine is done by a graphical user interface (GUI) that is shipped with the Yakindu plugin. This interface, the state machine Diagram Editor and it's additional views are perfectly integrated into Eclipse to have a uniform development environment.

The simulation of a state machine is integrated into the Yakindu state machine Diagram Editor and provides visual highlighting of the active state and the actual transition. Additionally, the user can interact with the simulation by sending triggers to or by changing variable values within the simulator to drive the state machine.

1.1 Yakindu and Open Architecture Ware

Yakindu is one branch that enqueues into the model driven architecture development. The basis of most model driven developments in the Eclipse environment is the Open Architecture Ware Platform (OAW, please visit the OAW web page for further information at <http://oaw.itemis.de/>). This plugin offers the ability to create model driven development approaches. One of these approaches is Yakindu, which focuses on embedded system development. E.g. for rapid prototyping or software model re-usage.

Hier könnte noch mehr zu OAW und MDSD hin.

1.2 Yakindu Toolchain

The Yakindu toolchain consists of a ...

2 Installation

The Yakindu Plugin installation follows the usual eclipse installation process.

However, before you start, be sure to have the eclipse environment (ganymede) installed from:

<http://oaw.itemis.com/openarchitectureware/language=en/2837/downloads>

This eclipse distribution contains all necessary elements, especially the Open Architecture Ware (OAW) plugin for model driven development.

To install the Yakindu plugin, open the **Software Updates and Add-Ons** dialog which could be found at Help→ **Software Updates...** Press the **Add Site...** button and enter the update

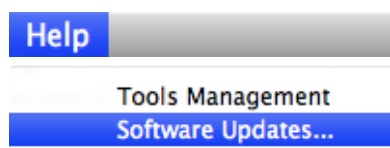


Figure 1: Menu to select software updates

site URL <http://updates.yakindu.com/release> into the *Location* area (see Figure ??). After accepting the new URL the update site will be queried.

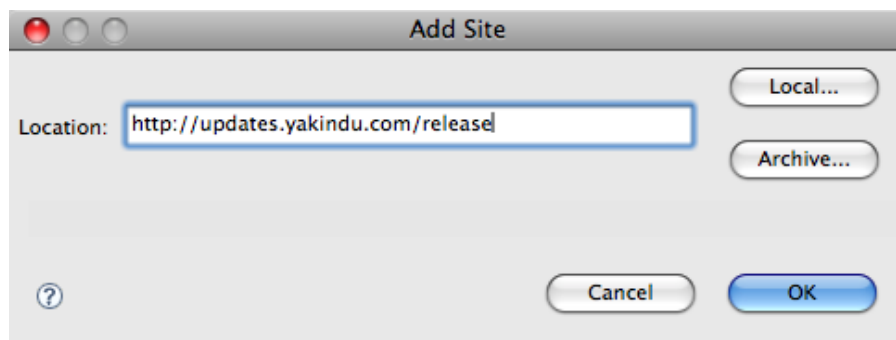


Figure 2: Update site for Yakindu

If everything is correct, you will find a new entry *Yakindu Update Site* in the list in the *Available Software* tab. For quick start check the Feature *Yakindu Feature* below the tree of *Yakindu Update Site* and press the **Install** button.

In the next steps you have to confirm the selected Features (see Figure ??) and accept the License after reading it. The next steps are automatic and are finished by a information Box from eclipse, asking you to restart. Answer with Yes, because Yakindu becomes active after restart.

After a few seconds, your installation is ready to run the quickstart example presented in the next section. Before we continue it is a good idea to change the perspective to Yakindu. With this perspective all main tools from the Yakindu-Toolchain are direct accessible. Although it's also possible to add some of this views to your favorite perspective and edit statecharts in parallel to your all day work.

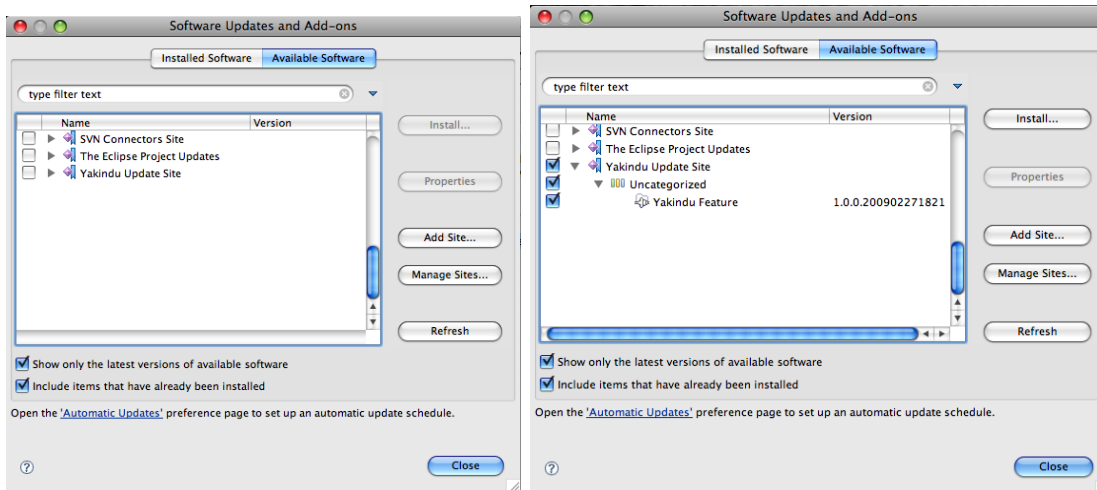


Figure 3: Software Updates and Add-Ons Dialog

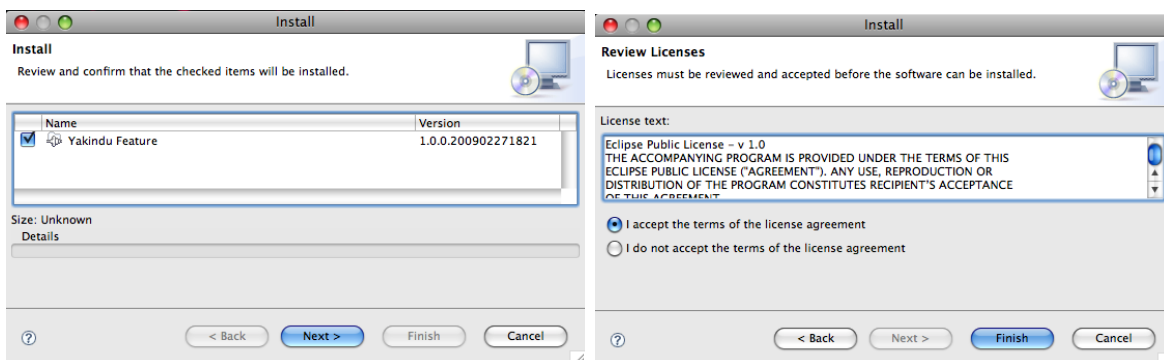


Figure 4: Confirm selected features and the licence

3 Example Quickstart: Staircase Lighting

When Yakindu is properly installed, we can start a small example project. This example project gives an idea of how powerful the Yakindu Toolchain is. It contains a visual editor, a semantic and logic verification check, a simulator unit and a number of code generators.

To present the visual editor, the check mechanism and the simulator of the Yakindu toolchain, an example was chosen, that is simple enough to give an impression of the usage but is not too far-fetched.

3.1 Example State Machine

The idea is to have a state machine, that represents a staircase lighting. This staircase lighting is started with a key-press and stops after 30 seconds.

The state machine itself consists of two states: „Lights On” and „Lights Off”. The standard state within the state machine is „Lights Off” and is entered from start-up (the so called initial state). When an occupant enters the staircase and presses the lighting button, a „keypress” event is generated which starts the transition to the „Lights On” state.

On entering the state „Lights On”, the staircase lighting is turned on. When the retention period has expired (after 30 seconds), the „Lights On” state is left with a transition to the state „Lights Off” and the lighting is turned off again.

3.2 Creating a new Project

When everything is set up, your Eclipse editor should look similar to this:

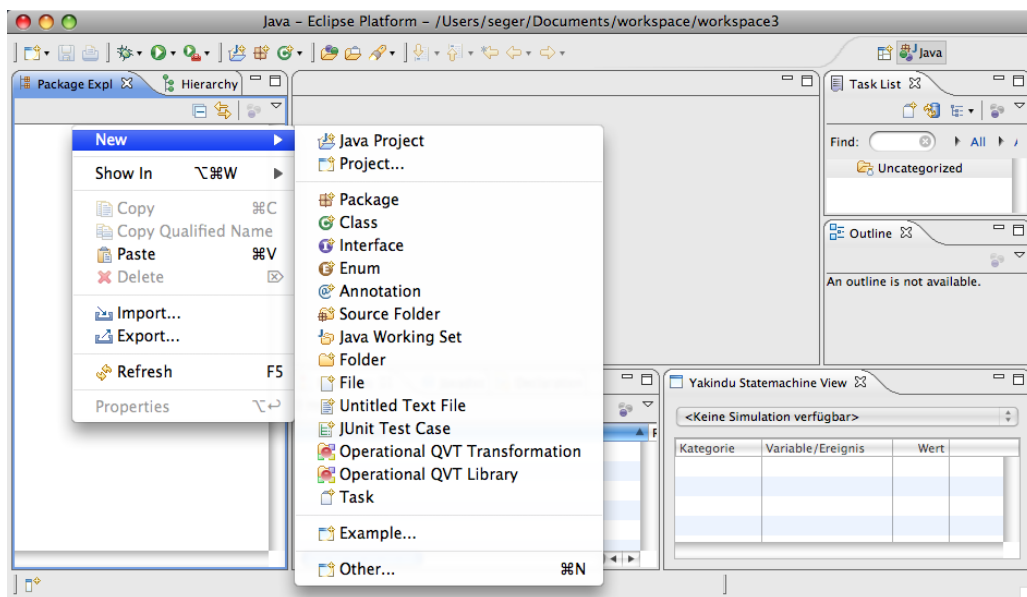


Figure 5: Creation of a new project

To start a new project, open the menu (**File**→**New**→**Java Project**) and create a name for that new project. In our example, the project is called „*StaircaseLighting*“. However, this procedure only creates a default Java project.

To create a **state machine model** to this environment, right-click the **src** directory icon and open the select wizard at **New**→**Other**. Here you have to select **state machine Diagram** from the **mda4e** folder as shown on figure ??.

Figure ?? displays the *New Statemachine Diagram* window, in which the name of the statemachine model is set. In our example, the name is changed from **default.statemachine** to **default.statemachine**.

The wizard has then created two sources: the **staircase.state machine** and the **staircase.state machine_diagram**. The state machine itself is represented as an XML-file in **staircase.state-machine** and the visual representation of the state machine can be found in the **staircase.state-machine_diagram** file.

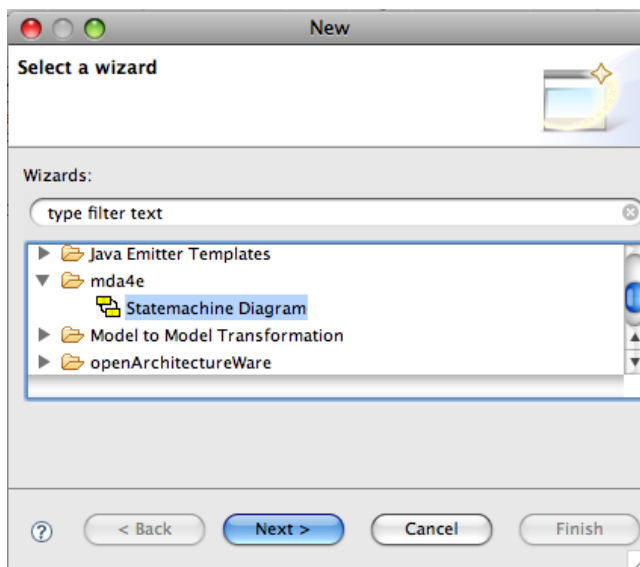


Figure 6: Wizard to create a new State Machine

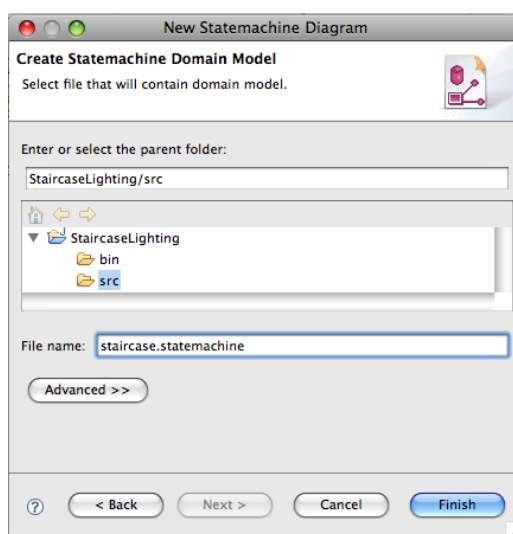






Figure 7: Wizard to create a new state machine domain model

Now you should have a new visual editors view to create a state machine as shown in figure ?? . Here you have all elements to create a state machine from bottom up in the elements menu. The elements that are important for our small project are  **Region** ,  **State** ,  **Transition** and  **Initial State** .

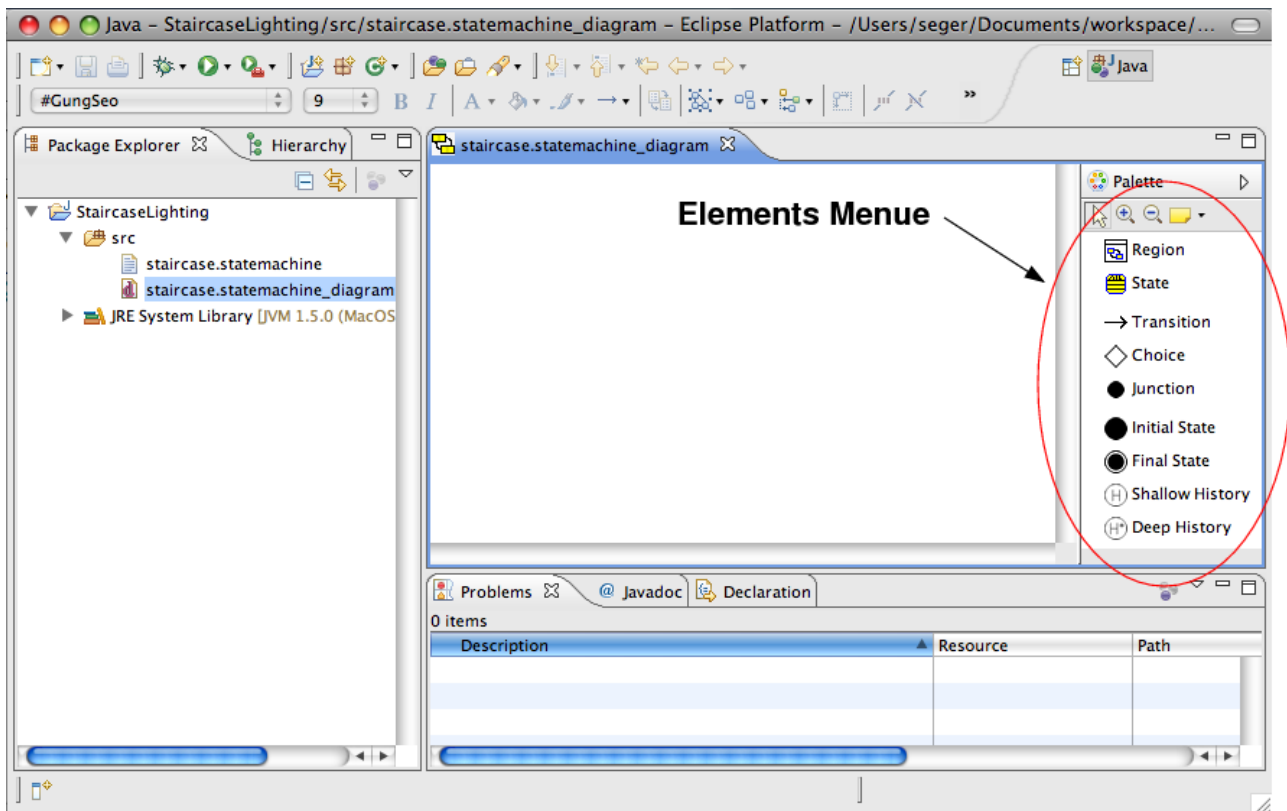





Figure 8: Yakindu State Machine editor

3.3 Defining a State Machine

To start with the visual editor you firstly need a region, in which the states of the state machine reside. Therefore you need to click the  **Region** icon and draw a region on the empty plain. When you have placed the last corner of the region and you release the mouse button, a new region in light green appears. At the properties area, the priority of this region is highlighted and should be set to a value. As we do not have any concurrent regions in this example, the priority is not important and could be set to any valid integer value (10 in our example). If you use more than one region, the priority specifies the processing order in which the states actions and the states transitions are processed (for further information please refer to section ??).

To set the priority of the region afterwards, you can open the properties view. If you cannot find it in your eclipse view, you can open it by clicking **Window** → **Show View** → **Others** and in that menu: **General** → **Properties**.

Now as you have created a region, you need a starting point of your state machine. This starting point is called **initial state** and can also be found as an icon ( **Initial State**) in the elements menu. As explained at the beginning of this chapter, the two states **LightsOn** and **LightsOff** should be installed within the state machine region.

To create a state, you have to select the ( **State**) element and then open the area within the region plain. When the state outline is created, the first line within the properties area is highlighted and needs to be filled by the name of this state (e.g. LightOff). The name of a state has to be unique

within a project.

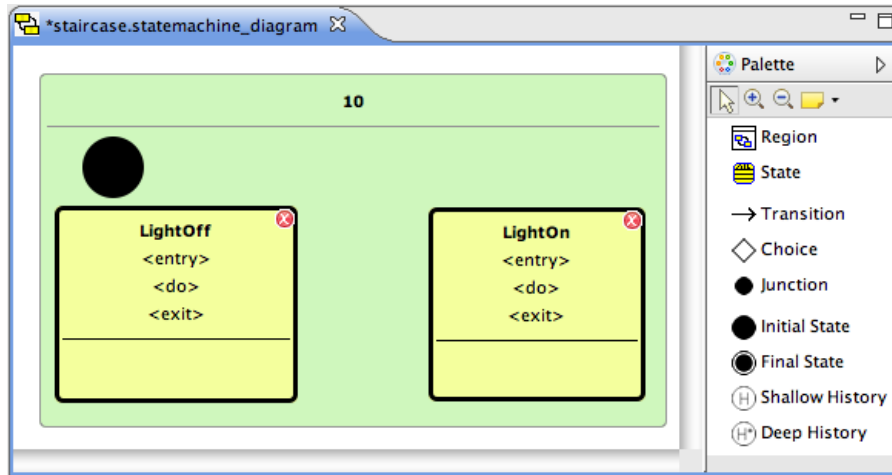
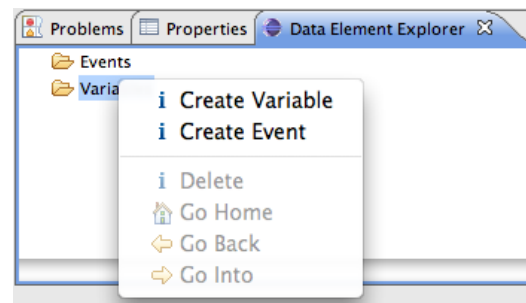


Figure 9: Creation of the **Initial**, **LightOff** and **LightOn** state

After the creation and naming of the states, the actions (`<entry>`, `<do>`, `<exit>`) should be created. In our case, the only action that should be performed is switching the light on or off. The status of the light is represented by a variable that could hold one of the two values 1 or 0. This variable needs to be created in the **Data Element Explorer**. Within this view, you can right-click on **Variables**, where you get a new window to specify the variable that should be created (refer figure ??). Here you add the variable name (*Light*), the port, the IO-type and the data type. In our example all other information except for the variable name do not have to be changed.

Then you add the action `Light=0` into the `<entry>` line within the states properties area (either within the Diagram or within the **Properties** view). This definition creates a new action, that is performed whenever the state **LightOff** is entered. So when the unconditioned state transition from the **initial state** to the **LightOff** state is performed, the internal variable *Light* is set to zero. The same takes place, when the state *LightOff* is entered through a transition from any other state.



The state **LightOn** is created in the same way, except that the action is set to `Light=1`.

Figure 10: Creating a variable in the *Data Element Explorer*

To create a transition between the **Initial State** and the **LightOff** state, you choose the **Transition** element and connect the **Initial State** and the **LightOff** state. Every transition claims an **expression**, when this transition should be executed.

This expression could consist of one or more *triggers*, *guard operations* and *actions*, which can also be mixed. Additionally a transition must have a priority, to define the order, in which the expressions of different, concurrent transitions are processed. To learn more about transition expressions, please read section ??.

In our example, the transition between the **Initial state** and the **LightOff** state do not need any expression, as this transition has no condition. When the expression area of a transition is left blank,

the expression is represented by an asterisk (*).

The transition between **LightOff** and **LightOn** is performed, if the trigger *keypress* was received. Therefore a new event has to be implemented in the *Data Element Explorer* (refer also to figure ??). Here you add an event that can be used as a trigger within a transition expression. Implementing a trigger with a transition needs only the trigger name as the expression string (*keypress*).

To specify the transition from **LightOn** to **LightOff** after 30 seconds, you use the keyword *after*(*<duration>* *>* *s*) within the transition expression string. The *after*() expression switches to the target state when the specified time has been expired.

So in the end your state machine looks like in figure ??.

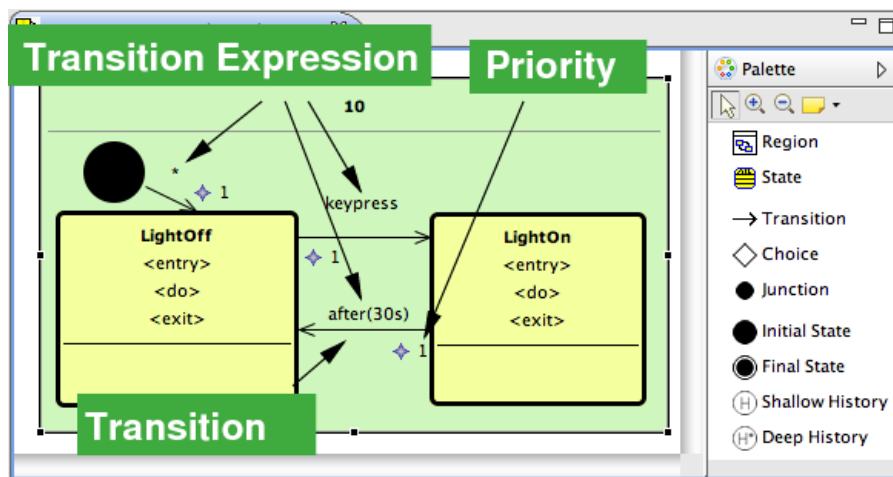


Figure 11: Complete State Machine (State Machine Diagram Editor)

3.4 Checking the State Machine

3.5 Simulating a State Machine

When the state machine diagram is completed, you can start a simulation session, to check, whether your new state machine is working correctly.

To start a simulation you have to create a new *run* configuration, as shown in figure ?? . Choose the entry **Run Configuration ...** to open the configuration dialog.

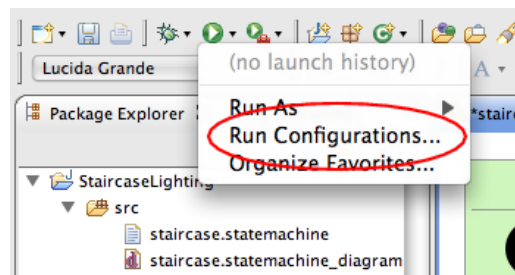


Figure 12: Starting the Simulation creation dialog

In this dialog you add a new **Yakindu Simulation** called *StaircaseLighting* by clicking the 'New' button in the upper left corner. Then the configuration dialog presented in figure ?? is shown. Choose your *StaircaseLighting* project and the staircase state machine as your model file. As your simulation engine, please choose *Yakindu Statechart Simulator*.

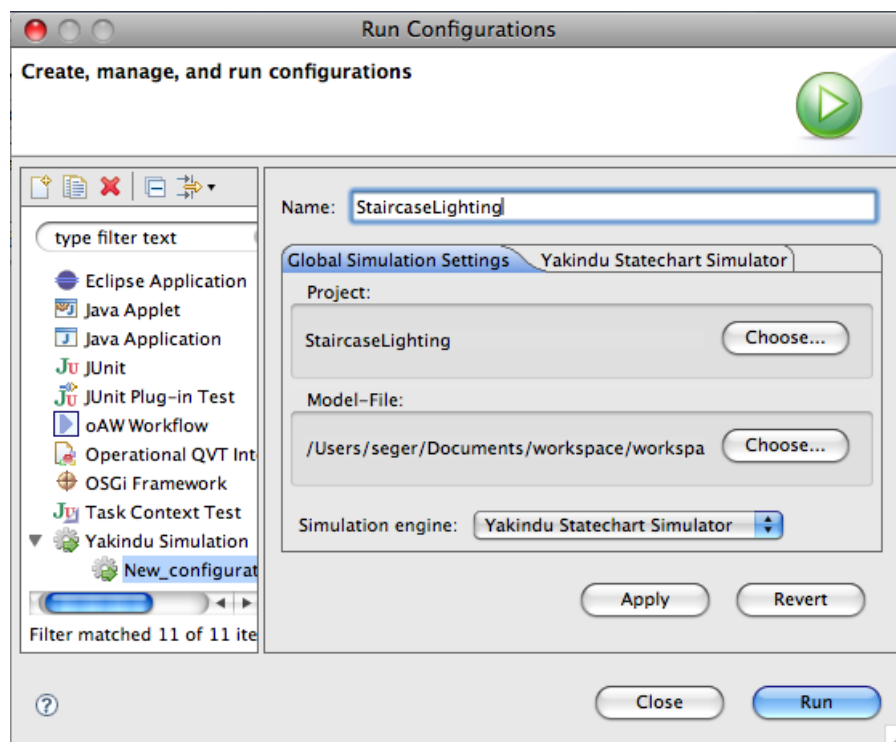


Figure 13: Dialog to create a new simulation environment

When you click the **Run** button in the lower right corner, you start the simulation. To rerun the simulation later, you find an entry *StaircaseLighting* in your **Run** configuration.

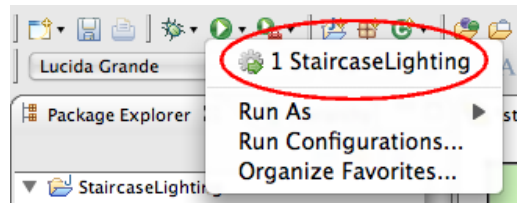


Figure 14: Restart a simulation in the menu

On Simulation startup the Simulator Window (refer figure ??) appears. In this dialog, the simulation process can be started, stopped and paused. Additionally the simulation can be used in single step modus. To activate the single step modus, the simulation must be in the pause position. When a single step should be performed, the single step button can be pressed. The advantage in this modus is, that you can set the input parameter according to your simulation scenario and perform the next step, when you are done with that.

To be able to interact with the simulation by the input values, you open the **Yakindu Statemachine View**. This view can be found in the main menu under **Window** → **Show View** → **Other...**. In the appearing menu choose **Yakindu Statemachine**.

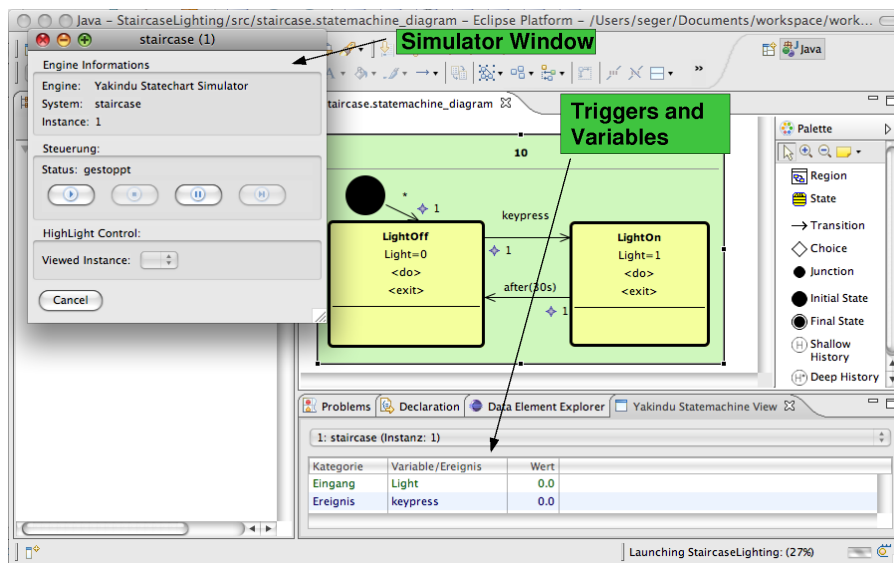
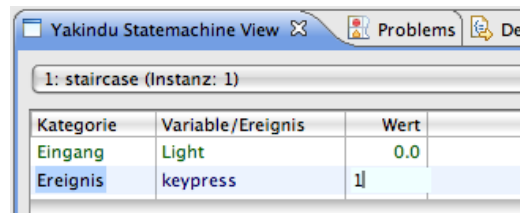


Figure 15: Running a simulation

In our example project, the state machine starts with a transition from the initial state to the *LightOff* state. This state transition is visualized by a red arrow. During a simulation an active state is highlighted in red.

After the active state has changed into *LightOff*, this state can only be left by a transition to *LightOn*. The condition expression is set to the trigger *keypress*. This trigger can be created in the Yakindu Statemachine View during simulation time. To simulate a keypress, just add a number greater than 0 as the value of the *keypress* event and hit return. In this case, the value changes and the transition is followed. After the transition has been performed, the trigger for the transition is reseted.



Kategorie	Variable/Ereignis	Wert
Eingang	Light	0.0
Ereignis	keypress	1

Figure 16: Change trigger value during simulation

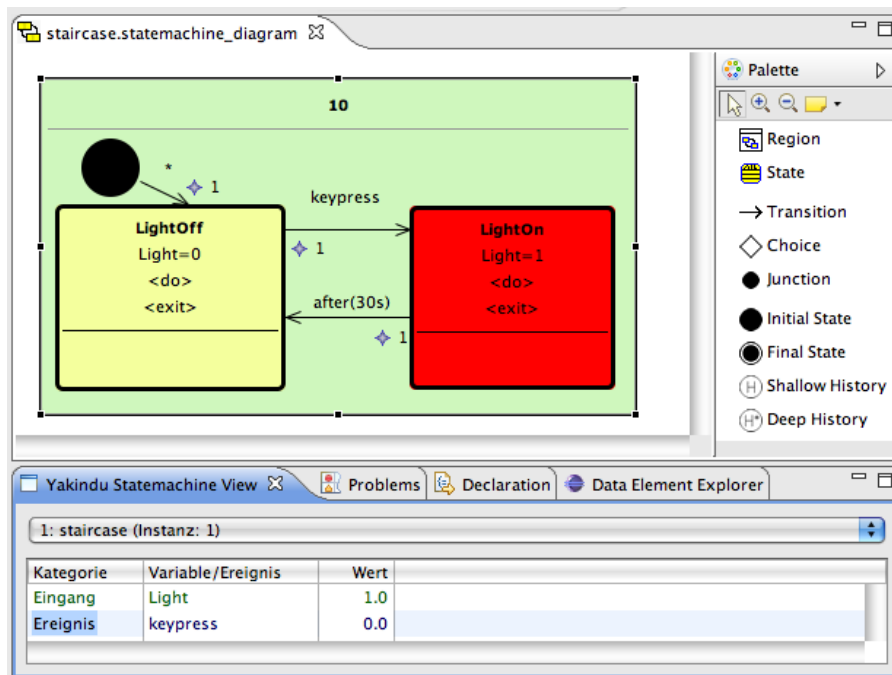


Figure 17: Trigger *keypress* was activated and the state *LightOn* has been reached

The same procedure is valid to set a value of a variable. The difference between a variable and a trigger is that a trigger value is reset to zero as soon as the trigger has taken effect, e.g. a transition was successful. A variable value is only changed when it is actively changed by the user or an action e.g. in *<entry>* *<do>* or *<exit>*.

4 Detailed usage information

4.1 Regions

A region is a sub-state machine that runs concurrently to the other regions. To serialize the processing of the sub-state machines within the regions, the regions possess a priority. The region with the highest priority is processed first.

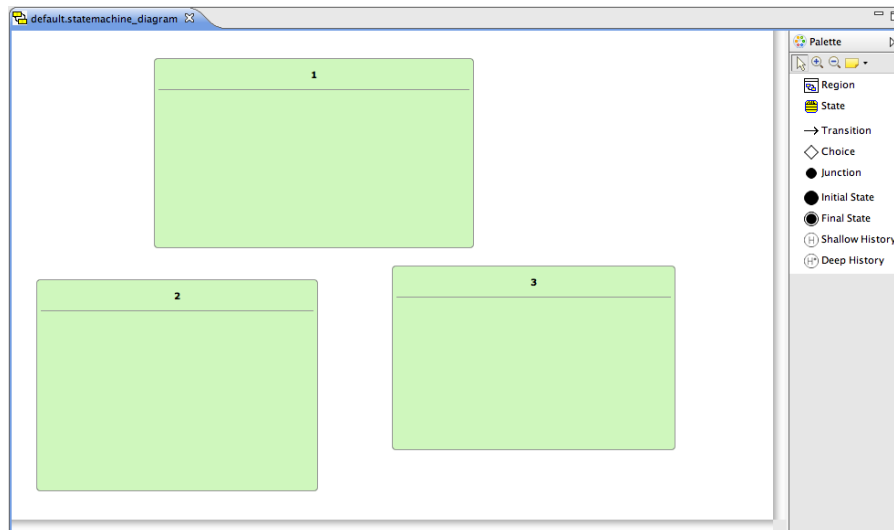


Figure 18: Three regions within a State Machine Diagram.

In the actual Yakindu version, the regions are not hierarchically aggregatable.

4.2 States

The states are the main elements of a state machine as they specify the possible states the state machine consist of. Every state has a unique name, which must be set by the user.

The parameters of this state are:

- **Entry:** The actions, which are performed on the arrival in this state are specified in the *< entry >* area.
- **Do:** The actions, which are performed while the state does not change, is specified in the *< do >* area.
- **Exit:** The actions, which are performed when a status is left, is specified int the *< exit >* area.

Hier fehlt noch: wie genau ist die "expression" die hier einzutragen ist definiert.

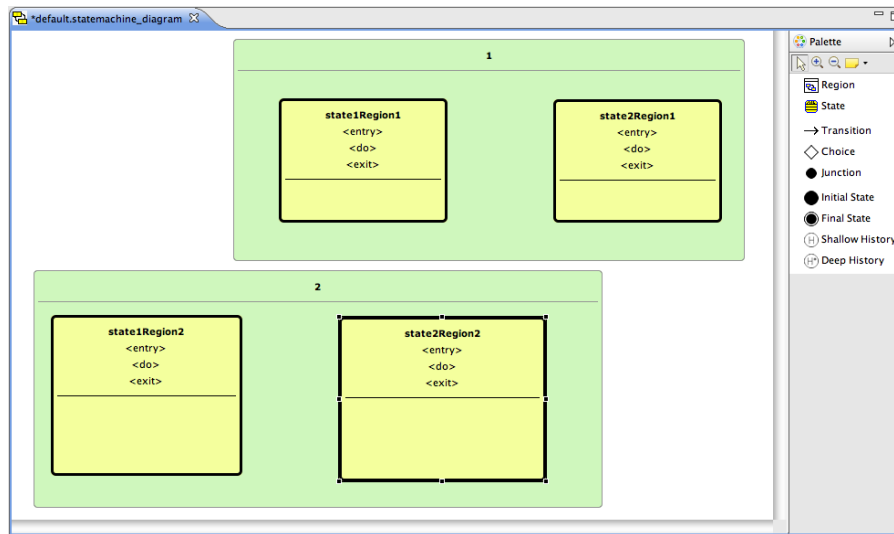


Figure 19: Two regions with two states each

4.3 Pseudostates

Pseudostates are states, that do not have any *< entry >*, *< do >* or *< exit >* areas. These states are used to specify special states like the **Initial State** or the **Final State**.

Actually the following states are defined:

- **Initial State:** The **Initial State** is the state where the state machine or the sub-state machine specified by a region start with. Usually the **Initial State** is connected unconditionally with another normal state.
- **Final State:** The **Final State** is the state where a state machine ends. When the **Final State** is reached, the state machine usually stops processing.
- **Shallow History:** The **Shallow History** carries the last active state within a sub-state machine. When this sub-state machine is re-entered, the last active state is set active again.
- **Deep History:** The **Deep History** works as the **Shallow History** but can also handle nested sub-state machines, so that the correct nested sub-state machine is used when a sub-state machine is entered.
- **Choice:** A **Choice** is an element to branch a transition. Every branch of a transition must have its own transition condition. An example is a transition, that is started with a trigger signal and then splits up to arrive at state X or state Y depending on a guard variable.
- **Junction:** The **Junction** connects two transitions that have the same target state.

4.4 Transitions

Transitions specify the change from one state to another. Every **Transition** must have a condition on which the state switch takes place. The syntax for this condition is:

Trigger1, Trigger2, ..., TriggerN[Guard]/Action1; Action2; ... ActionM;

- **Trigger:** A trigger is activated by an outside event. If more than one trigger is specified all triggers are connected by **OR**.

A special trigger is specified by the *after(< duration >)* keyword. This keyword defines a duration after which a transition is performed. The *< duration >* is an integer number with a specifier (e.g. *after(12s)*).

- **Guard:** A guard is a boolean expression. Example: $A=B$
- **Aktion:** An action consists of instructions, that should be performed during transition. This action should be short to have a reliable flow.

Additionally every **Transition** needs a unique priority. The reason is that every **Transition** have to be unique, so that the state machine always behaves in the same way independent of the sequence the **Transitions** are created for example.

4.5 Variables and Events

The state machine is driven only by triggers and guarded variables, as discussed in section ???. These two types of variable data can only be used within a transition expression, if the variables and events (which results in triggers) are defined.

To define a variable or an event, you open the **Data Element Explorer** and right-click on either **Events** or **Variables**.

The dialog **Create Variable** requests at least a name for the variable. Additionally, the variable can have a **Port**. The **I/O-type** specifies whether the variable is a local, input or output variable. At last, the **Data Type** can be chosen between integer (int), floating point (double) or boolean (bool).

Events are very similar to variables. The main difference during event creation is that you can not specify a **Data Type** but a **Trigger Type** which can hold the values *raising*, *falling* or *either*.

4.6 Interfaces

To interact with the state machine the system designer can add interface functions within an actions expression. These functions must follow the **Action Function Pointer** definition (i.e. the function is returning void and does not have any parameter). To meet the MISRA requirements, a function pointer must be a constant pointer, which means that the interface function must be bound to the state machine during compile time. So there is no late binding available in this case.