



Statechart Tools

User Guide

Alexander Nyßen, Dominik Schmidt, Benjamin Schwertfeger,
Jörn Seger, Axel Terfloth

Version M 0.3

Contents

1	Overview	5
1.1	YAKINDU Statechart Tools	5
1.1.1	YAKINDU and Eclipse	6
1.1.2	Status, Warranty and License	6
1.1.3	Tool architecture	6
2	Installation	9
2.1	Eclipse Installation	9
2.2	Installing the YAKINDU-Plugins	10
2.3	Installing from Zip	12
3	Statechart Modelling and Simulation	13
3.1	The YAKINDU Perspective	13
3.2	Example Step by Step	14
3.2.1	Example State Machine	14
3.2.2	Creating a new Project	14
3.2.3	Defining a State Machine	16
3.2.4	Checking the State Machine	19
3.2.5	Project type and dependencies	19
3.2.6	Create check files	20
3.2.7	Editing check files	21
3.2.8	Simulating a State Machine	22
3.3	Using Example Projects	25

4	C-Code Generator	27
4.1	Introduction	27
4.2	Example Scenario	28
4.3	Starting the workflow	28
4.4	Code Integration into an Existing Project	30
4.5	State Machine Access	31
4.6	Operating System and Drivers for the Example Device	32
5	Java-Code Generator	35
5.1	Introduction	35
5.2	Setting Up The Example Project	36
5.3	Generating Source Code	38
5.3.1	Create Xtend Project	38
5.3.2	Manage Dependencies	40
5.3.3	Create MWE Workflow File	42
5.3.4	Execute MWE Workflow	45
5.4	The Generated Source Code	46
5.5	Integrating Generated Java Code	47
6	UML Transformation	51
6.1	Introduction	51
6.2	UML2 model	51
6.3	The example project	51
6.4	How does it work	53
6.4.1	Name mapping	53
6.4.2	New elements	53
6.4.3	Limitations	54
6.4.4	Transformation Cartridge	54
6.5	Extending an UML2 state machine	55

Chapter 1

Overview

1.1 YAKINDU Statechart Tools

YAKINDU is a tool kit for model based development and the statechart tools are the first modules provided by this project. The tools apply the concept of state machines that are well understood and formal enough to describe behaviour unambiguously. The statechart tools support editing, validating, simulating state machines and generating code from state machines. The tools are provided as Eclipse-plugins and integrate tightly into the IDE.

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 current 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.

The distribution described by this document contains:

- statechart meta model
- statechart editor
- Eclipse YAKINDU perspective
- instant validation
- statechart simulator
- C-code generator
- Java-code generator
- Model transformations from UML
- Eclipse integration
- User guide

- examples

Future versions will add :

- Code generation for different target languages like PLC and others.
- Testing infrastructure

Take a look at the roadmap on the YAKINDU website for details.

Even though the main aim of YAKINDU is to support the development of embedded systems, state machines are a general concept that is also widely used in other domains like the field of enterprise systems. Thus, the YAKINDU Statechart Tools can be of value for software development in general.

1.1.1 YAKINDU and Eclipse

The YAKINDU statechart tools are completely based on Eclipse technologies and especially those from the Eclipse Modeling Project (EMP) and openArchitectureWare (oAW). This user guide will refer to those technologies wherever necessary. For a overview please take a look at the web pages <http://www.eclipse.org/modeling/> and <http://www.openarchitectureware.org/>

1.1.2 Status, Warranty and License

The version M0.2a described by this document is the first version of the YAKINDU statechart tools. It is freely available and will be provided without any warranty. The next version will include the source code and will be published as open source under the Eclipse Public License (EPL).

1.1.3 Tool architecture

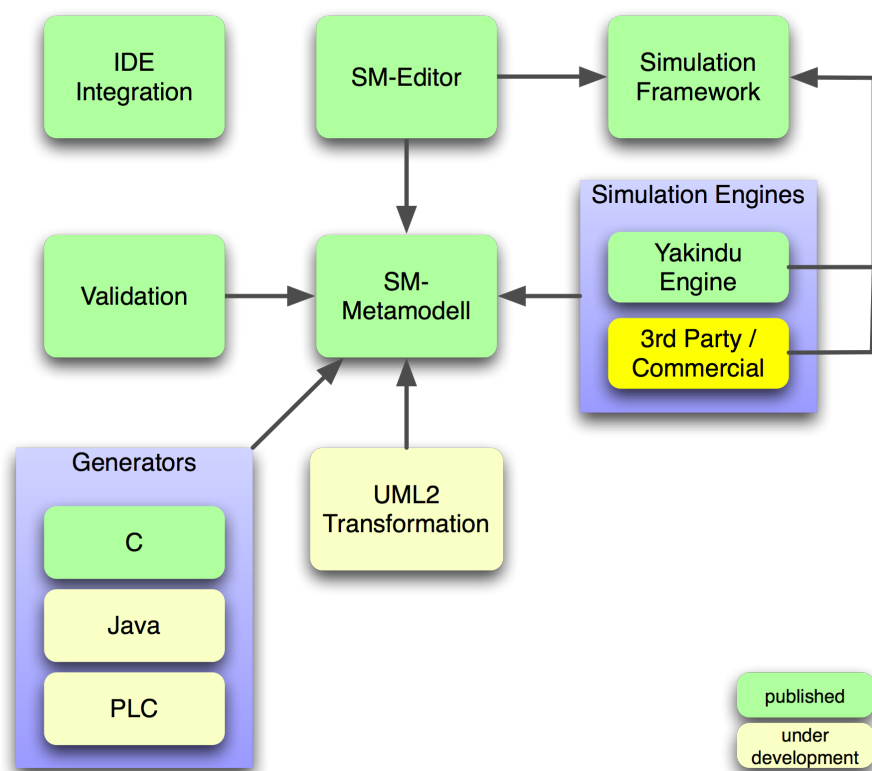


Figure 1.1: Tool architecture

Chapter 2

Installation

2.1 Eclipse Installation

The YAKINDU Plugin installation follows the usual eclipse installation process.

However, before you start, be sure to have the Eclipse environment (3.5/Galileo) installed. You can download eclipse distributions from the Eclipse download site.

<http://www.eclipse.org/downloads/>

Please be aware that there are many different distributions supporting different features. When installing the YAKINDU features the required features will be installed automatically if they are not already installed. You can also download the itemis oAW distribution that already contains several required plugins.

<http://oaw.itemis.com/downloads/>

The YAKINDU statechart tools require several Eclipse plugins:

- EMF
- GEF
- GMF
- Xtend
- Xpand
- Xtext
- MWE
- and some others. . .

Additionally you may want to install additional features like the CDT (C/C++ Development Tools) or the JDT (Java Development Tools). You can install them with the same mechanism as described in the next section.

2.2 Installing the YAKINDU-Plugins

To install the YAKINDU plugins, open the **Software Updates and Add-Ons** dialog which can be found at Help→ **Install New Software...** (like 2.1). In the window press the **Add ...** button and

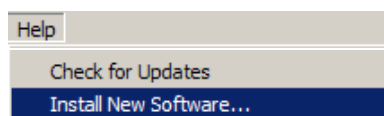


Figure 2.1: Menu to select software updates

enter the update site URL `http://updates.yakindu.com/galileo/daily/` into the *Location* area (see Figure 2.2). After accepting the new URL the update site will be queried.

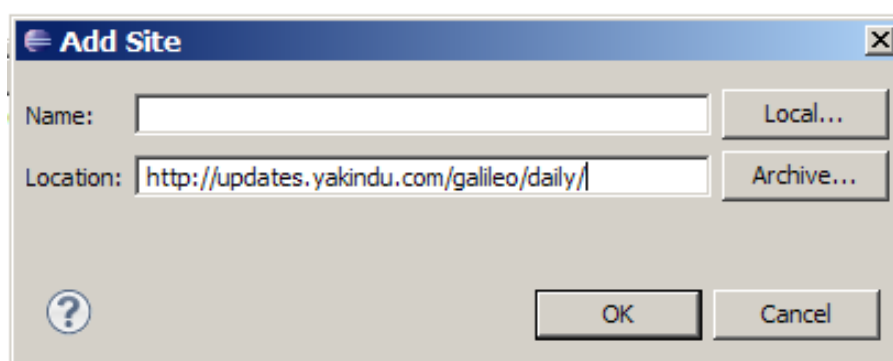


Figure 2.2: Update site for YAKINDU

If everything is correct, you will find a new entry *YAKINDU Update Site* in the list in the *Available Software* tab. Before continue it is necessary to activate update sites for MWE, Xpand and Xtext. This is done by clicking on **Manage Sites...** and selecting the Galileo update site.

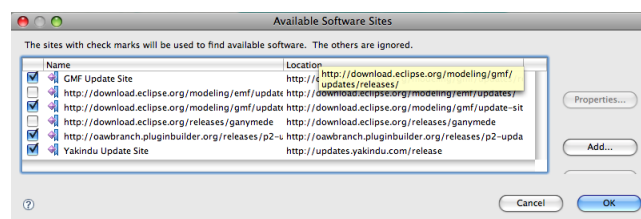


Figure 2.3: Select update site for dependency resolving

For quick start check the Feature *YAKINDU Feature*, *MWE SDK*, *Xpand SDK* and *Xtext SDK* below the tree of *YAKINDU Update Site* and *Galileo Update Site*. After select the Features press

the **Install** button. For the first installation many dependencies are downloaded and you have to wait some minutes.

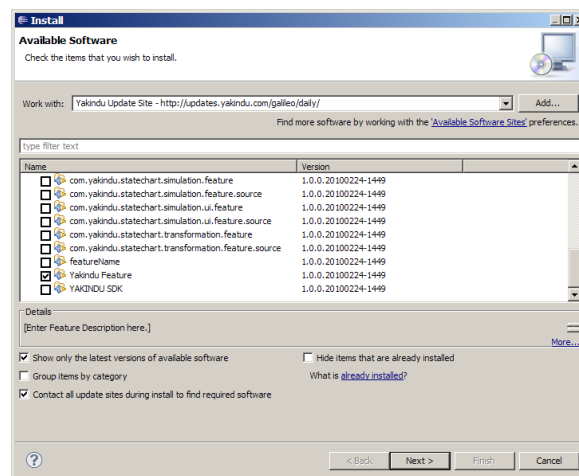


Figure 2.4: Software Updates and Add-Ons Install Dialog

In the next steps you have to confirm the selected Features (see Figure 2.5) and accept the License after reading it. Confirm the Security Warning by clicking the "Ok" button. The next steps are automatic and are finished by an information box from eclipse, asking you to restart. Answer with Yes, because YAKINDU becomes active after restart.

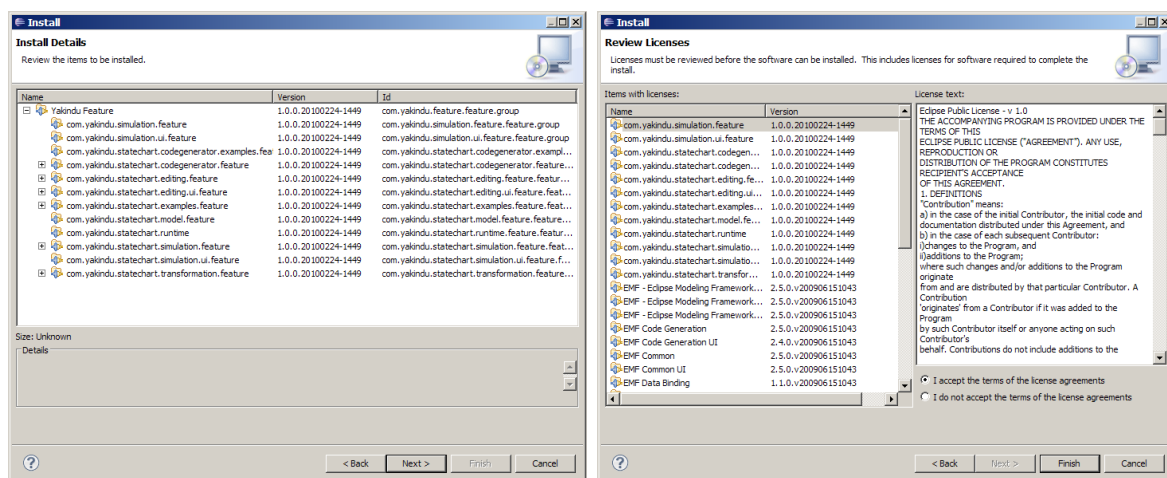


Figure 2.5: Confirm selected features and the license

After a few seconds, your installation is ready to run the quick start 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 directly accessible. Although it's also possible to add some of this views to your favourite perspective and edit statecharts in parallel to your all day work.

It is highly recommended that you update your plugins after installation via *Help* → *Check for Updates*. So you get the actual version of the YAKINDU-Statechart Tools.

2.3 Installing from Zip

The YAKINDU-Web Site also provides a zip file with all plugins for download. The dependencies to GMF must be satisfied manually. The update sites for eclipse are the following and the required features are mostly the sdks of the plugins mentioned before:

- EMF Update Site: <http://download.eclipse.org/modeling/emf/updates/>
- GMF Update Site: <http://download.eclipse.org/modeling/gmf/updates/releases/>
- Galileo Update Site: <http://download.eclipse.org/releases/galileo>
- Xtext <http://xtext.itemis.com/downloads/>
- and some others...

Chapter 3

Statechart Modelling and Simulation

Now follows a quick introduction into the YAKINDU tools and the usage. For a deeper understanding you can read the later chapters.

3.1 The YAKINDU Perspective

The YAKINDU tools comes with an easy perspective for quick starting and using it. The perspective can be activate by clicking on **Window** → **Open Perspective** → **Other** and selecting the YAKINDU perspective (Figure 3.1). After activation the screen looks like figure 3.2.

In the middle the main view on screen is the editor. It is still empty, but you will use it most often. On the left is also a default eclipse view, the project explorer. You will need it for the next step. The other views will be described later, when they are needed.

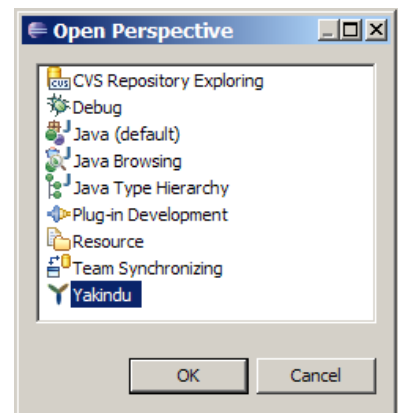


Figure 3.1: Selecting the YAKINDU Perspective

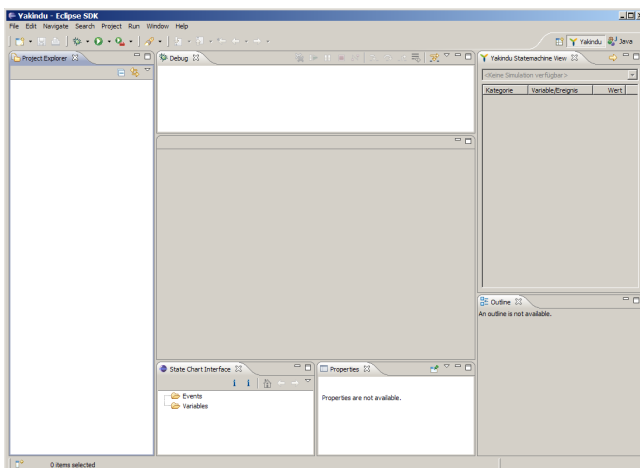


Figure 3.2: The YAKINDU Perspective

3.2 Example Step by Step

When YAKINDU is properly installed, we can start a small example project. This example project gives an idea of how powerful the YAKINDU tool-chain 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 tool-chain, an example was chosen, that is simple enough to give an impression of the usage but is not too far-fetched.

3.2.1 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.

3.2.2 Creating a new Project

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

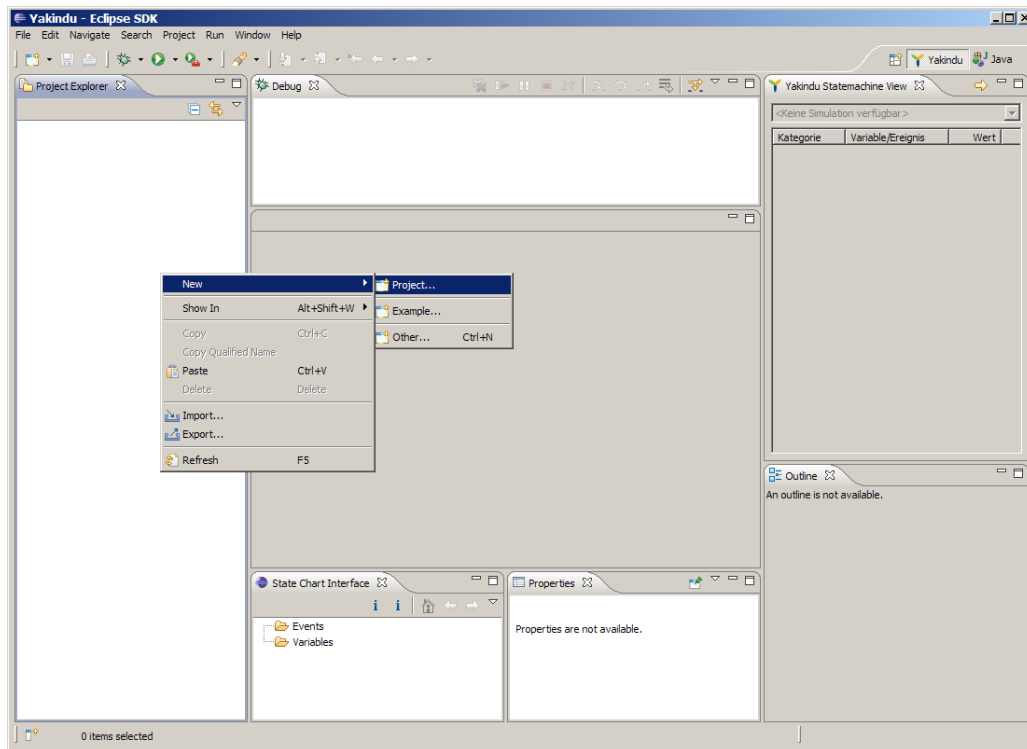


Figure 3.3: Creation of a new project

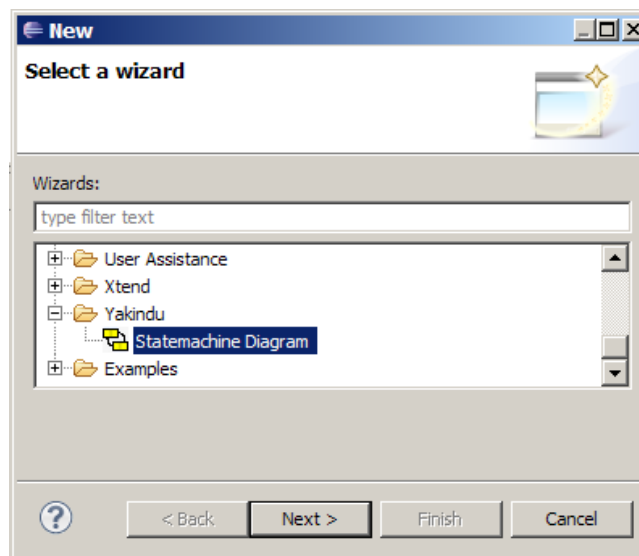






Figure 3.4: Wizard to create a new State Machine

To start a new project, open the menu (**File**→ **New**→ **Project**) and create a new Java Project. In our example, the project is called „*Example*”. However, this procedure only creates a default Java project. For simplicity you should deny the question for changing to Java perspective, because we won't use anything from Java.

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 **YAKINDU** folder as shown on figure 3.4.

Figure 3.5 displays the *New Statemachine Diagram* window, in which the name of the state machine model is set. In our example, the name is changed from **default.statemachine** to **staircase.statemachine**.

The wizard has then created two sources: the **staircase.state-machine** and the **staircase. statemachine_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.statemachine_diagram** file.

Now you should have a new visual editors view to create a state machine as shown in figure 3.6. 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** .

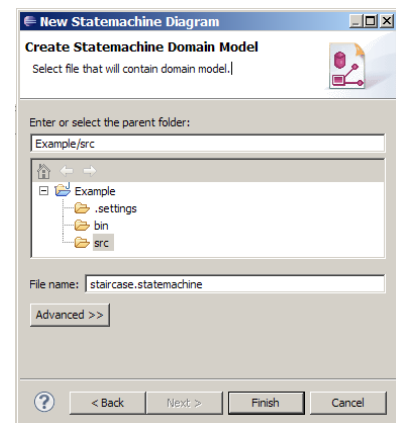





Figure 3.5: Wizard to create a new state machine domain model

3.2.3 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.

To set the priority of the region afterwards, you can open the properties view. If you cannot find it in your eclipse perspective, 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 **LightOn** and **LightOff** 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 region.

After the creation and naming of the states, the actions (**< entry >**, **< do >**, **< exit >**) should be

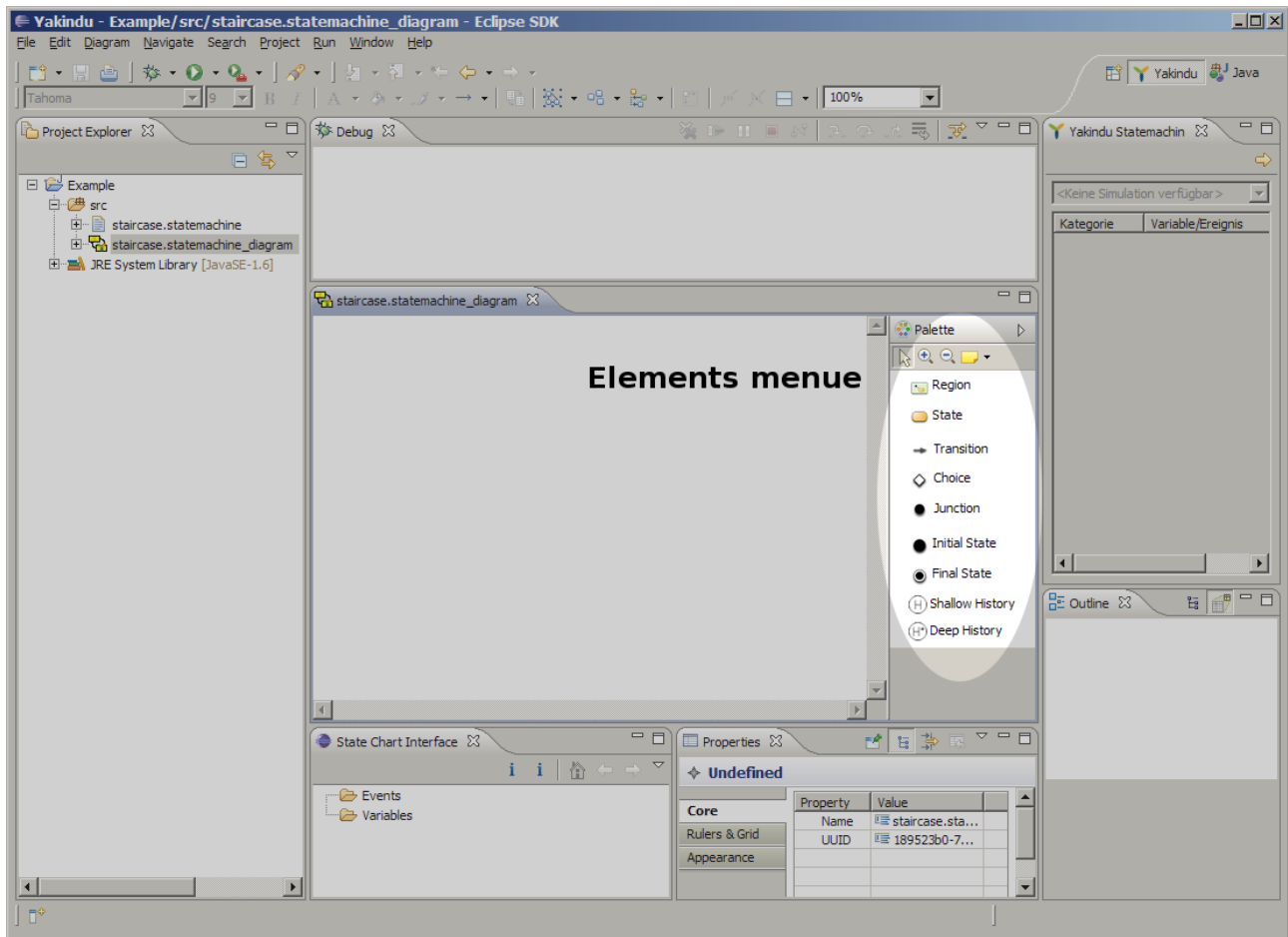


Figure 3.6: YAKINDU State Machine editor

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 **State Chart Interface**. 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 3.8). 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;`.

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.

An expression can consist of one or more *triggers*, *guard operations* and *actions*, which can also be

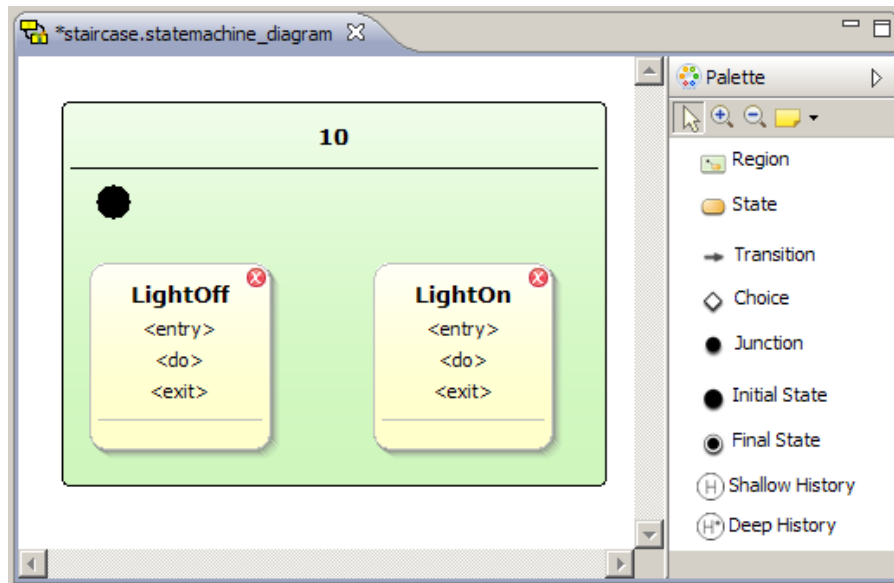


Figure 3.7: Creation of the **Initial**, **LightOff** and **LightOn** state

mixed. Additionally a transition must have a priority, to define the order, in which the expressions of different, concurrent transitions are processed.

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 *State Chart Interface* (refer also to figure 3.8). 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*).

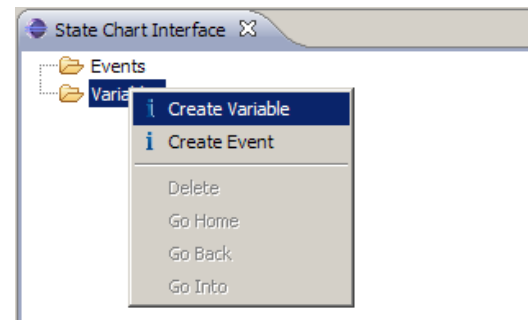


Figure 3.8: Creating a variable in the *State Chart Interface*

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 3.9.

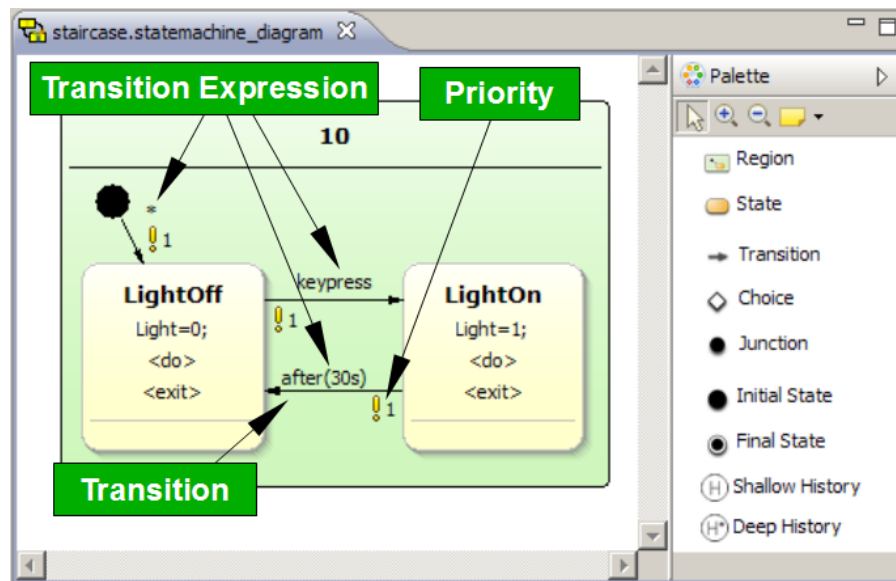


Figure 3.9: Complete State Machine (State Machine Diagram Editor)

3.2.4 Checking the State Machine

The state chart is very common and it is possible to model things, which doesn't make sense for your target platform or you want to have some special naming conventions. We will check now, that every name of a state is at least eight character long. But first we have to prepare our Project for code completion within checks. Later on we can add a new check file and enter the checks.

3.2.5 Project type and dependencies

For this steps it is necessary, that the model is inside an Java project. If it's not yet, you can create a new project beside the first, copy the model and change the target directories (see chapters 4 and 5) for generated files, so they point to the first.

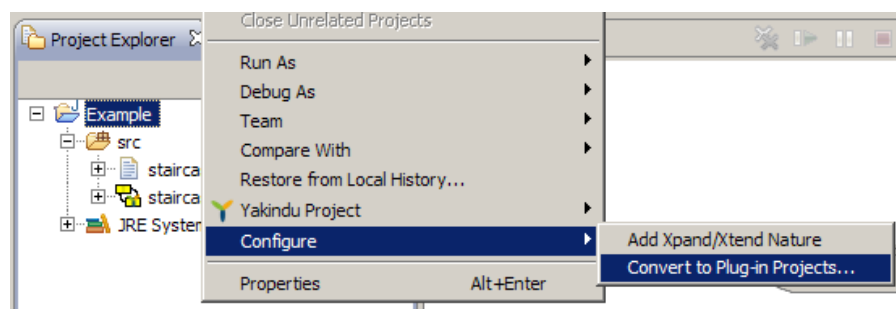


Figure 3.10: Add plugin development nature to your java project

We will add the plugin development nature from eclipse to the project by selecting the project, click with the right button and select **Configure** → **Convert Projects to Plugin Projects...** (Picture 3.10). Your project is already selected in the next window and you can press Finish to complete. As

a result some new files are created in your project. We only need the **MANIFEST.MF** inside the folder **META-INF**. Open this file by double-click and add some dependencies (See picture 3.11). This is done by clicking on **Add** and selecting some plugins. If you design for codegeneration the plugin *com.yakindu.statechart.codegenerator.java* or *com.yakindu...c* are the one of your choice. For platform independent modeling you can select *com.yakindu.statechart.model.expressions*.

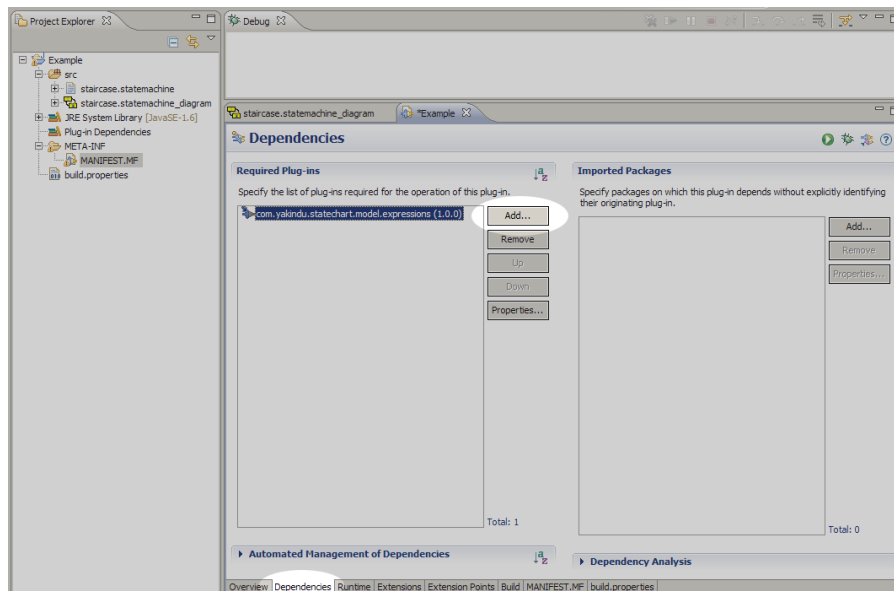


Figure 3.11: Add dependencies to your project

Now the file can be saved and closed. It will not be necessary to edit this file later, except you decide to use an additional codegenerator.

3.2.6 Create check files

All user defined check files reside in a folder names checks. Within this folder all **.chk**-files are evaluated and considered for checking. The format of check files is described in the reference of eclipse modeling project and the old openArchitectureWare-project¹.

Create a new folder in the root of your project by right-click on **Examples** and selecting **New→Folder**. The folder must be named *checks* and the other options can be ignored. Simply finish and create your check files inside this folder.

Check files are created by selecting the folder "checks" and choosing in the drop-down-menu the entry **New→Other**. In the next window the entry **Check file** is found inside the category **Xtend**(see Figure 3.12).

¹<http://oaw.itemis.de/openarchitectureware/662/ressourcen>

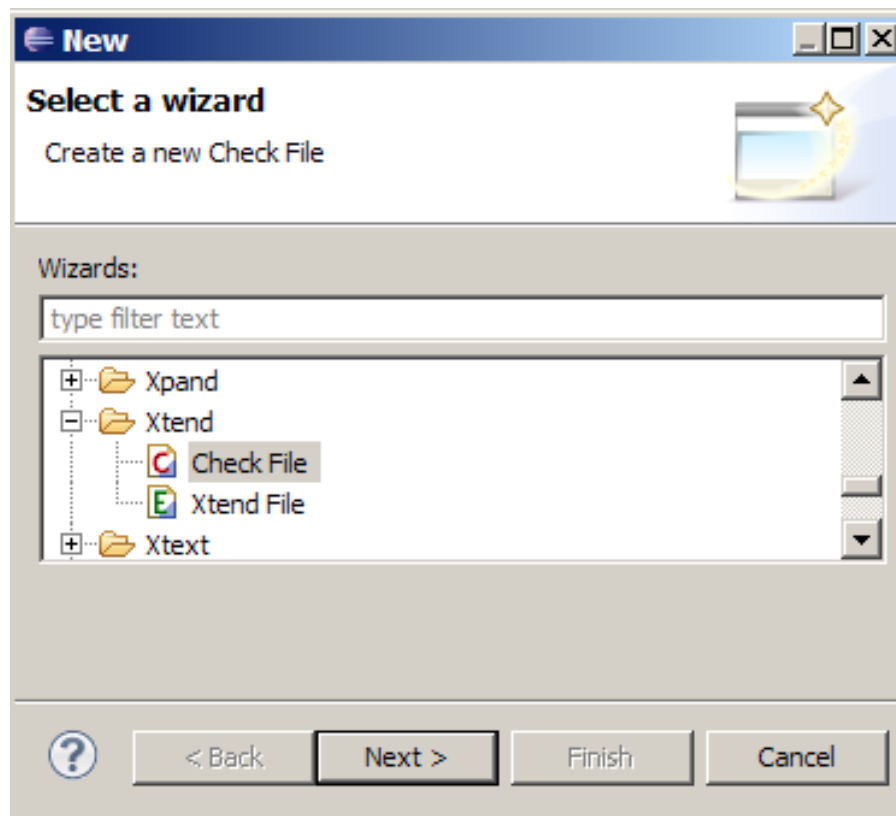


Figure 3.12: Add a new check file to your project

3.2.7 Editing check files

The syntax of check files is described in the oAW-Reference, but the short introduction here is enough for the first experience. If you try the code completion (`Strg+space`) the first time in the editor, you are asked to add the Xtend nature to your project (See figure 3.13). Answer with yes, to get code completion.

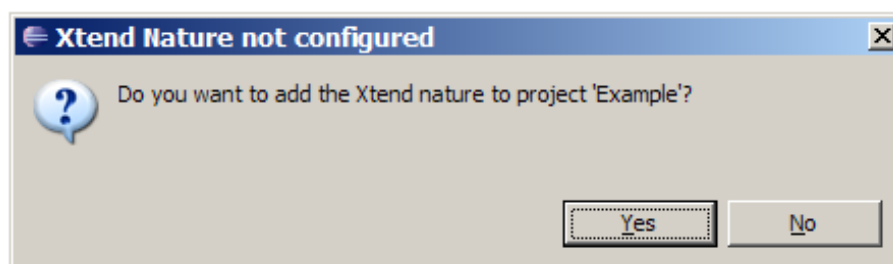


Figure 3.13: Add the Xtend nature to your project for code completion

Every check file starts with an `import` of the required and used models. For the statemachine it is sufficient to add the model `statemachine`. After this we want to add a check for a state. This is done by defining a context for the model element `State` and the severity of `WARNING` or `ERROR`. Because short names are not nice and we spelling names we define an error with the error message "State names must have at least 8 characters". After a colon the boolean expression follows.

The expression defines the default, not the error. In our case names are longer or equals to eight characters. In figure 3.14 the result of "state.chk" is printed.

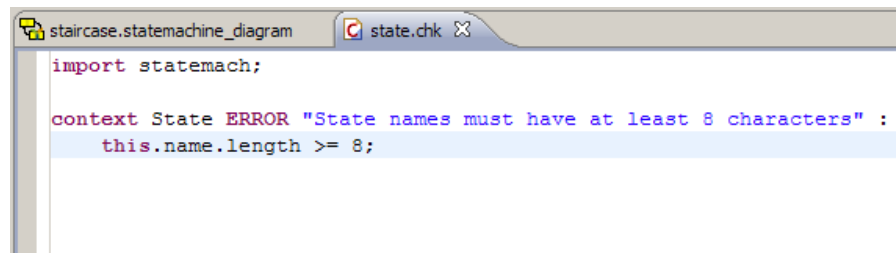


Figure 3.14: An example check file for names longer than eighth characters

After saving the file and opening the statechart again, the statechart is validated after some seconds. As a result you can see a red cross in the upper right corner of state LightOn. That's because we added a check, that state names must have at least eight characters and LightOn needs only seven characters. If you put the mouse above the red cross and wait some seconds, the error message is shown in a small box.

3.2.8 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 3.15. Choose the entry **Run Configuration ...** to open the configuration dialog.

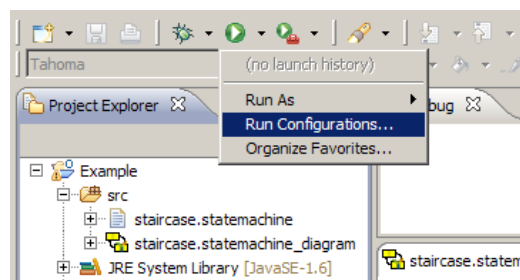


Figure 3.15: Starting the Simulation creation dialog

In this dialog you add a new **YAKINDU Simulation** called *Example* by clicking the 'New' button in the upper left corner. Then the configuration dialog is presented. After choose your *Example* project and the staircase state machine as your model file the dialog looks like figure 3.16. As your simulation engine, please choose *YAKINDU Statechart Simulator*.

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

On Simulation start-up make sure, that the debug view is open (refer figure 3.18). In this view, 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.

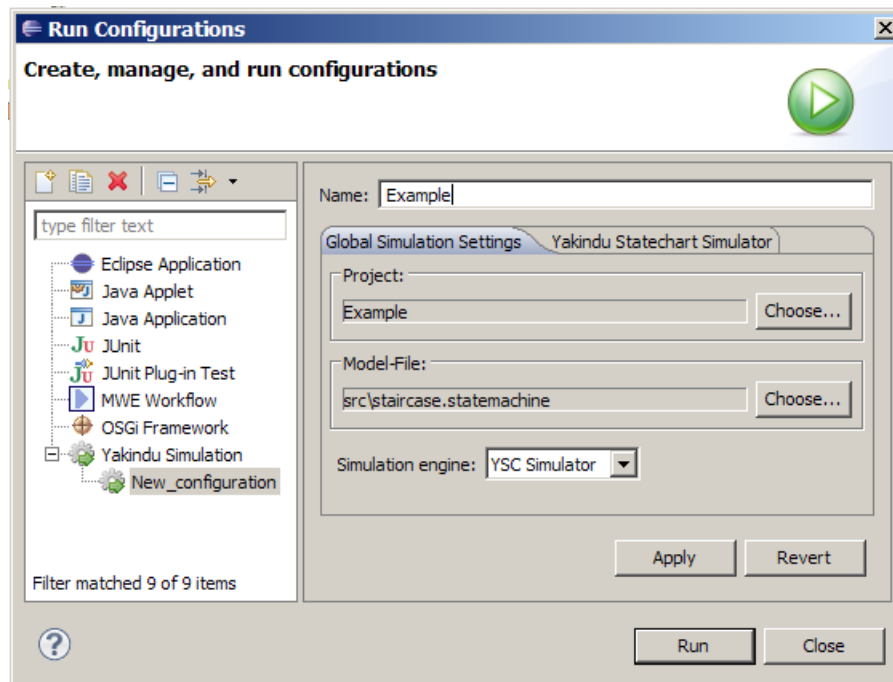


Figure 3.16: Dialog to create a new simulation environment

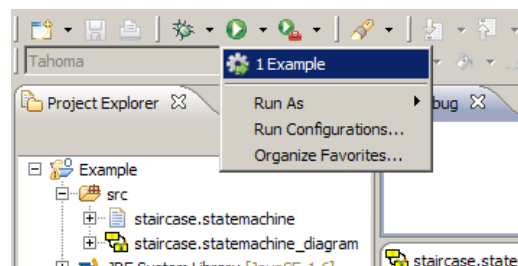


Figure 3.17: Restart a simulation in the menu

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 change the events and variables, you must have started the simulation explicit by a click to the run button in the debug view, or by clicking on the single step button.

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

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 Statechart View during simulation time. To simulate a keypress, just click the

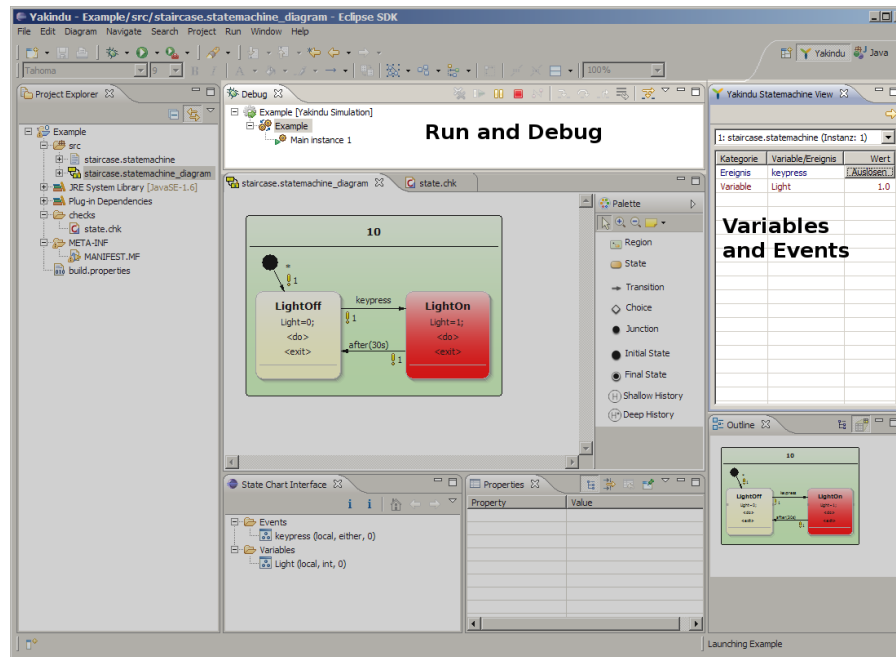


Figure 3.18: Running a simulation

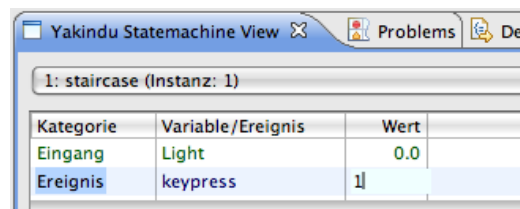


Figure 3.19: Change trigger value during simulation

"fire" button near the *keypress* event in Statemachine view. In this case, the value changes and the transition is followed. After the transition has been performed, the trigger for the transition is reset.

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 >*.

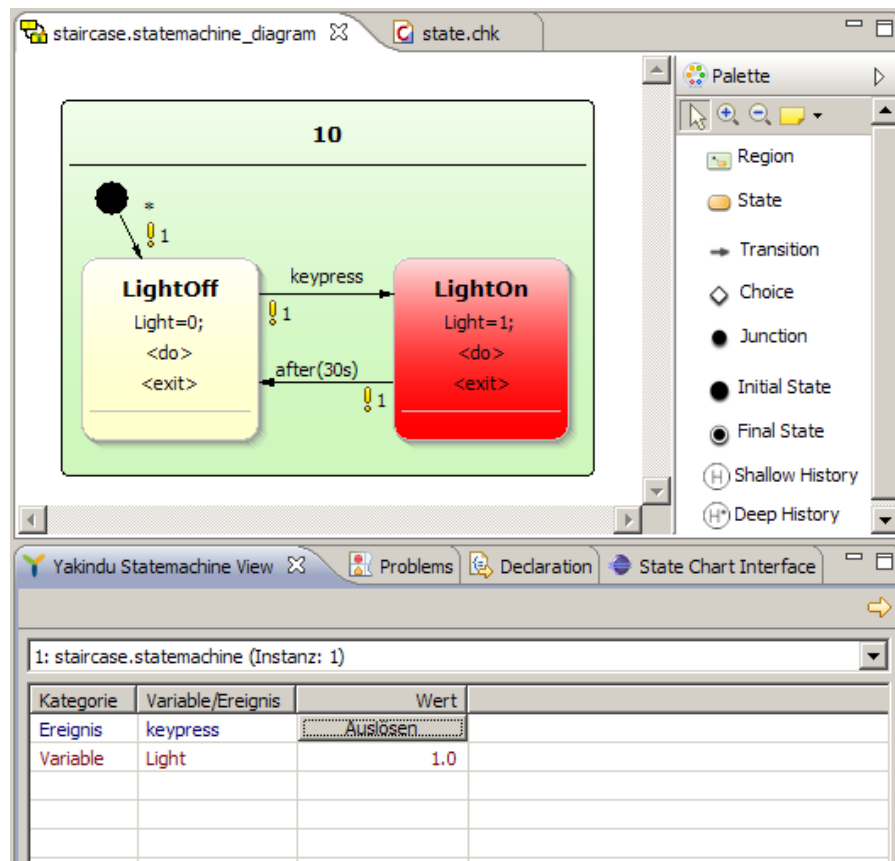


Figure 3.20: Trigger *keypress* was activated and the state *LightOn* has been reached

3.3 Using Example Projects

Some example projects are included in the YAKINDU release. They can be installed by creating the **Statemachine example project**. This is done by selecting **File** → **New** → **Statemachine example project**. This Wizard (see Figure 3.21) creates after all three new projects in your workspace, called "Safe", "StaircaseLighting" and "TrafficLight". You can read a short description of them, if you select one. After you click finish the new projects are listed in your project explorer.

Taking a deeper look at the files inside the examples (Figure 3.22), the two projects Safe and StaircaseLighting contain only two files. The *.statemachine* file contains the model of your state machine, and the *.statemachine_diagram* file is for modelling. Both files are necessary to simulate and visualize the simulation. If no visualization is needed, the model itself is enough. How to simulate is described in section 3.2.8.

The files inside the third project, the TrafficLight model, includes also model files inside the model directory, but also an workflow under workflow and an complex simulation environment for practical usage with a microcontroller. This set up and handling is described in the next section 4

Now we will start with a short hands-on example to demonstrate how development is done and what especially the elements and views are for. This step is orthogonal to-the-ready to use examples in this section.

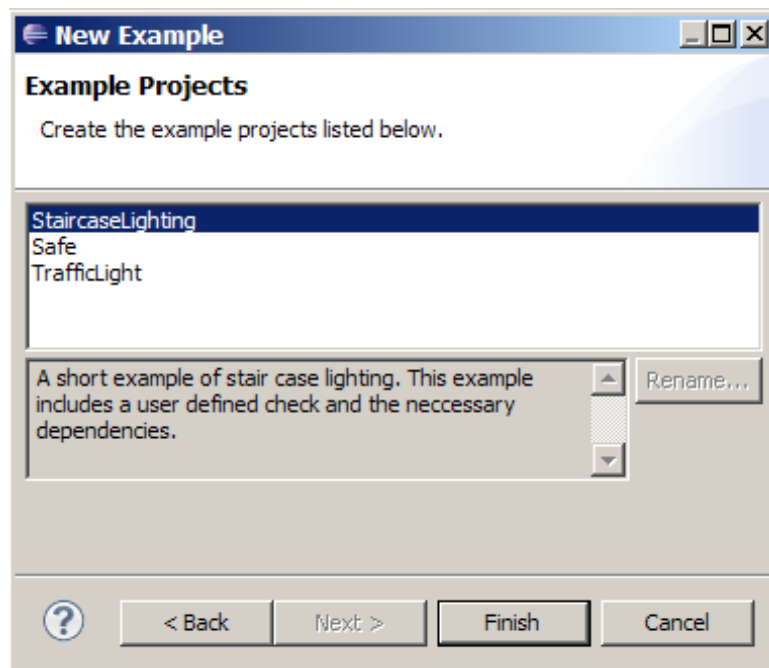


Figure 3.21: The state machine examples

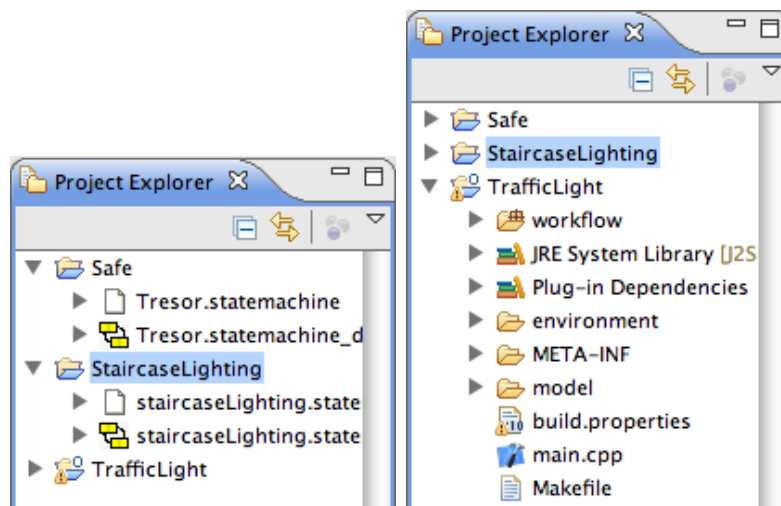


Figure 3.22: Example projects and files inside

Chapter 4

C-Code Generator

4.1 Introduction

The creation of a state machine is shown in YAKINDU Tutorial in detail. However, what is left is how the state machine models can be used to create source code. The purpose of this document is to give an overview how the code is created and how the created code can be integrated into an existing project.

The C source code generator, shipped with the YAKINDU release, is optimized for small embedded systems with certain restrictions, like small RAM/ROM, ANSI-C restrictions and MISRA rules (i.e. no heap usage, no function pointers). These restrictions are mandatory for many tasks e.g. in the automotive area.

Currently, the **YAKINDU C source code generator** is under heavy development as the other YAKINDU features, too. So the interfaces are not fixed yet and will probably change in the near future.

This document guides through an example scenario on an **Display3000** development board. It uses an Actmel ATmega128 CPU, 128 kByte Flash and 4 kByte RAM.

The Traffic Light example, which is discussed in this section, is included into the YAKINDU examples and can be installed as described in section 3.3.

The example scenario is a simple pedestrian traffic light. A pedestrian can press a button to indicate, she/he wants to cross the street. Then a blinking white light indicates, that the traffic light has recognized the request. After a few seconds, the traffic light for the street turns to red and the pedestrian traffic light turns to green. Then the pedestrian traffic light turns to red and the street traffic light changes to green again.

The state machine to model this behaviour is shown on figure 4.1. This state machine must not be created but is shipped with the example project and can be found in the `workspace/Traffic_Light/` directory.

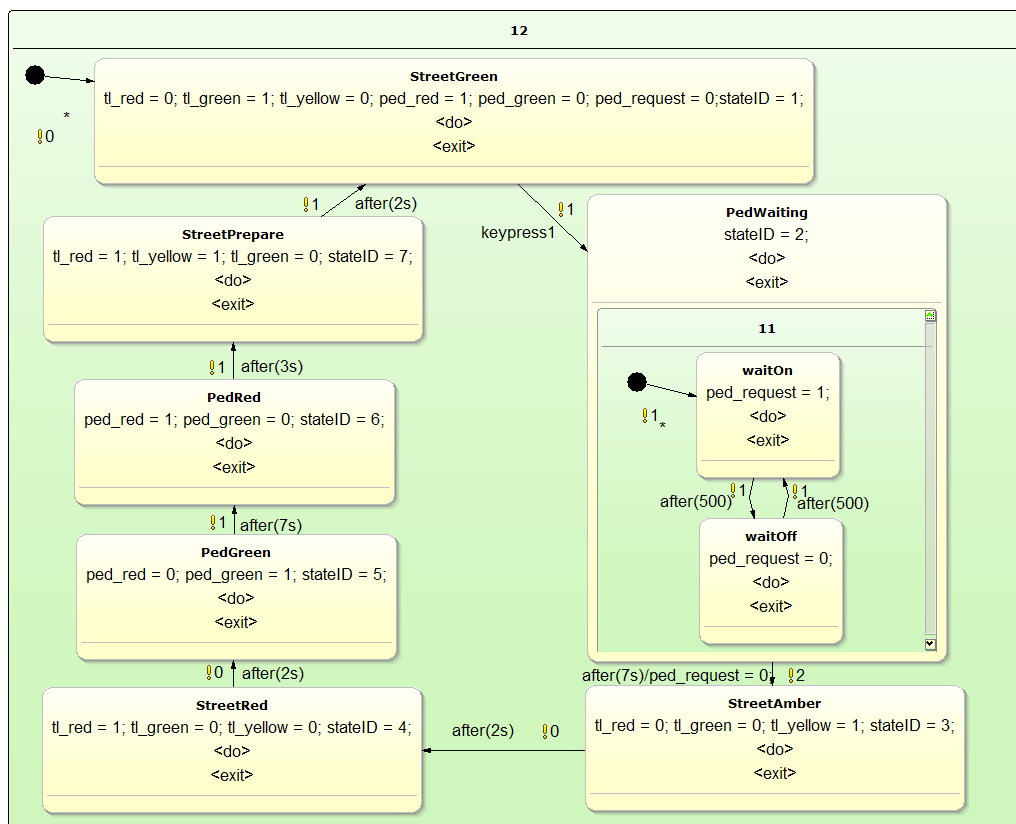


Figure 4.1: Statemachine for the Traffic Light Example

4.3 Starting the workflow

The source code is created by starting the MWE workflow. This workflow can be found in the *traffic light* example directory:

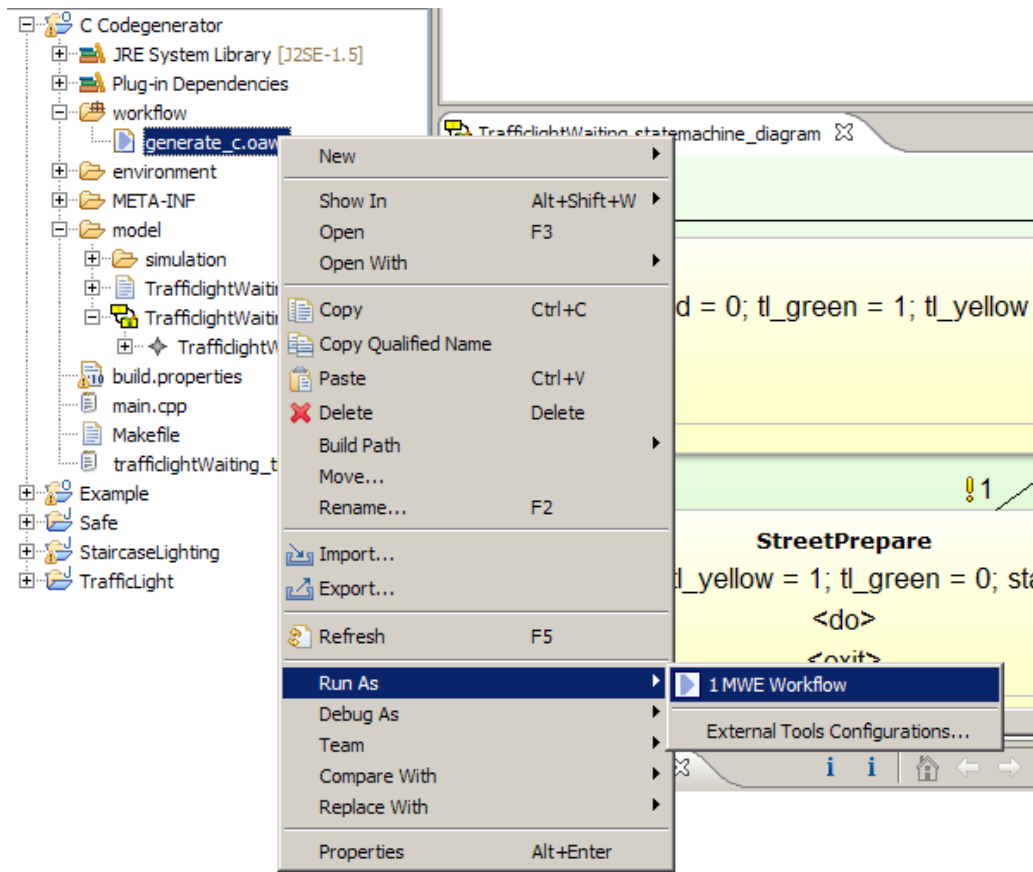


Figure 4.2: Starting the oAW-workflow to create the sources

The generated C source code can be found within the new directory `c-src-gen`:

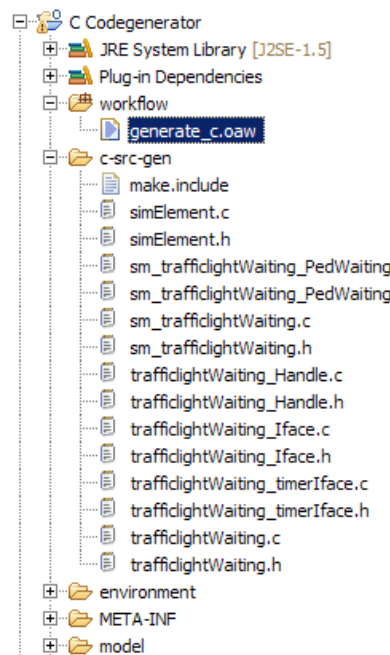


Figure 4.3: The sources are placed into the *c-src-gen* directory

4.4 Code Integration into an Existing Project

Aside the source code for the state machine and the code for the interfaces, the code generator creates a file called `make.include`. This file can easily be included into an existing makefile by the line:

```
include c-src-gen/make.include
```

The files, which are needed for the compilation, are added to the environment variable `SM_SOURCE`, so you have to add these sources to your project sources:

```
OBJECTS      = main.o $(ENV_OBJ) $(GRAPHIC_OBJ) $(SM_SOURCE)
```

Additionally you should add the `c-src-gen` directory to the include and source paths, so that the headers and sources could be found.

The example is designed for the Display3000 board, so the **Makefile** was designed to create a binary for this hardware. Therefore it requires the AVR-gcc toolchain to be able to compile. The toolchain contains a compiler, a linker and some other useful tools e.g. to download the data.

So go to your workspace directory with a shell and from there into the base directory of your project and call `make`:

```
workspace$ cd Traffic_Light
Traffic_Light$ make
```

```
[...] compiling [...]  
avr-gcc -Wall -Os -DF_CPU=7456000 -mmcu=atmega128 -I. -I./graphicLib [...]  
rm -f main.hex  
avr-objcopy -j .text -j .data -O ihex main.elf main.hex
```

If the compile was successful, you have a working binary `main.hex` that could be deployed on the target.

To transfer the binary, you can use `make flash`.

4.5 State Machine Access

As mentioned before, the code for the state machine can be found in the folder **c-src-gen**. After running the workflow the following files should be available in this directory:

```
c-src-gen$ ls  
make.include                sm_trafficlightWaiting_PedWaiting.h  
simElement.c                trafficlightWaiting.c  
simElement.h                trafficlightWaiting.h  
sm_trafficlightWaiting.c     trafficlightWaiting_Iface.c  
sm_trafficlightWaiting.h     trafficlightWaiting_Iface.h  
sm_trafficlightWaiting_Handle.c trafficlightWaiting_timerIface.c  
sm_trafficlightWaiting_Handle.h trafficlightWaiting_timerIface.h  
sm_trafficlightWaiting_PedWaiting.c
```

The files completely define the state machine. How to integrate the source into another project, was subject of the previous section.

The files `sm_trafficlightWaiting.*` and `sm_trafficlightWaiting_PedWaiting.*` represents the two regions in the state chart. The header- and C-file called `sm_trafficlightWaiting_Handle.*` contain the main handle, which is called `SM_trafficlightWaiting_Handle`. Please do not access the state machine handle information directly.

The structure carries the information about the current state, the actual transition, that has been activated, the handle for the first level region and the interface handle. The handle is initialized with the function `trafficlightWaiting_init(&sMachineHandle, &interfaceHandle)`. Here the `sMachineHandle` is the state machine handle and the `interfaceHandle` is a pointer to an interface handle. The initialization call initializes the whole state machine and returns the interface handle pointer.

For convenience all a state machine user needs to include is `trafficlightWaiting.h`. This header includes all other necessary information.

4.6 Operating System and Drivers for the Example Device

To let the state machine run on the Display 3000 development board, the system needs a minimal **operating system** (OS) and some drivers for input and output. This operating system is found in the `environment` directory. This cooperative OS is written in C++ and contains an input driver for the 6 keys on the hardware board and an output driver for the display and a LED board that is connected to the *port A*.

Because of copyright restrictions, the display driver is not included into this example. The calls to the display interface can be included by adding `-DWITH_DP3000_GL` to the compiler options and updating the include path defined in variable `COMPILETT` of the Makefile to point to the right AVR-Libraries.

The transfer of the LED data uses a simple software driven SPI-like interface with three connections (data, clock and inherit).

The rest, like shifting the data, is done by the hardware board.

Following files belong to the operating system:

```
definitions.h  event.h      scheduler.cpp  task.cpp
event.cpp     prioQueue.h  scheduler.h    task.h
```

The `key*` files contain the driver for a debounced key input. The `output*` files contain the driver code to create the output. To create the cycles for the state machine, this behaviour is placed in the files `statemachine*`.

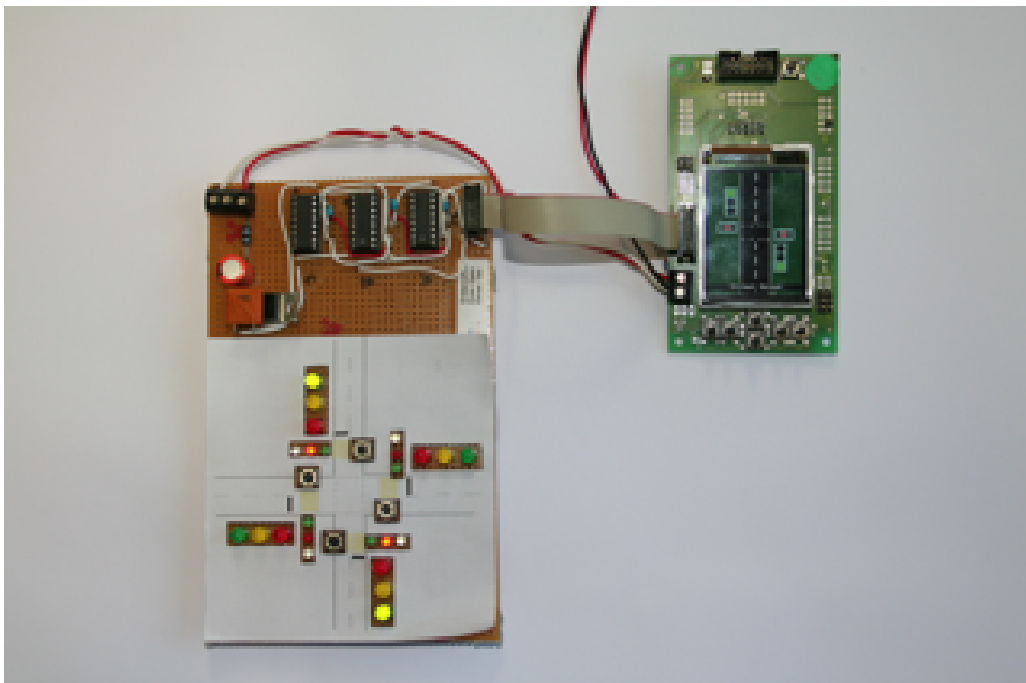


Figure 4.4: Display3000 Development board with additional Hardware

After generation of source code by the workflow, it is possible to call the Makefile inside the root directory of this example. By default this File calls the commands **avr-gcc** and **avr-g++**. If you

like to send the binary to the micro controller, it is necessary to check the `PORT` variable inside the Makefile and **avrdude** needs to be on classpath. If everything is correct, `make flash` will send the binary to the controller.

Chapter 5

Java-Code Generator

5.1 Introduction

The creation of a YAKINDU Statechart is shown within the YAKINDU Tutorial in full detail. However, what is left is how the state machine models can be used to create Java source code. This will be demonstrated in the following, making use of the *TrafficLight* demo project deployed with the YAKINDU Statechart tools.

5.2 Setting Up The Example Project

The reference example used within this documentation is the *TrafficLight* example project deployed with the YAKINDU distribution. To set it up in your local workspace, use the YAKINDU Example Project Wizard as depicted by Figures 5.1 and 5.2.

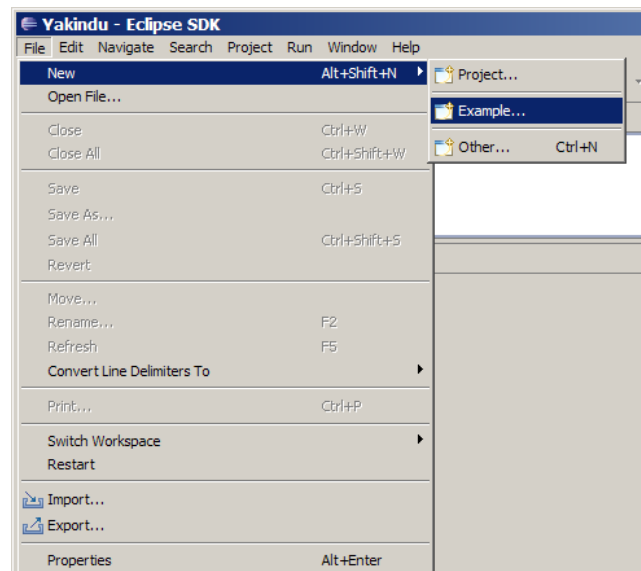


Figure 5.1: Opening the Example Creation Wizard

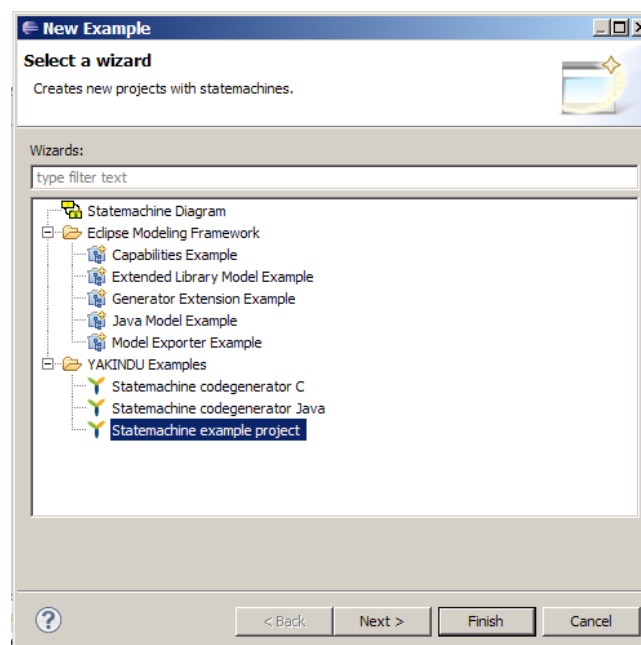


Figure 5.2: Creating Statechart Example Projects

After successful completion, the Example Project Wizard will create the YAKINDU Statechart example projects within the workspace, out of which one - *TrafficLight* - will serve as demonstration

example in the following. As outlined within Figure 5.3, it denotes a state chart to control a simple traffic light with a corresponding pedestrian light.

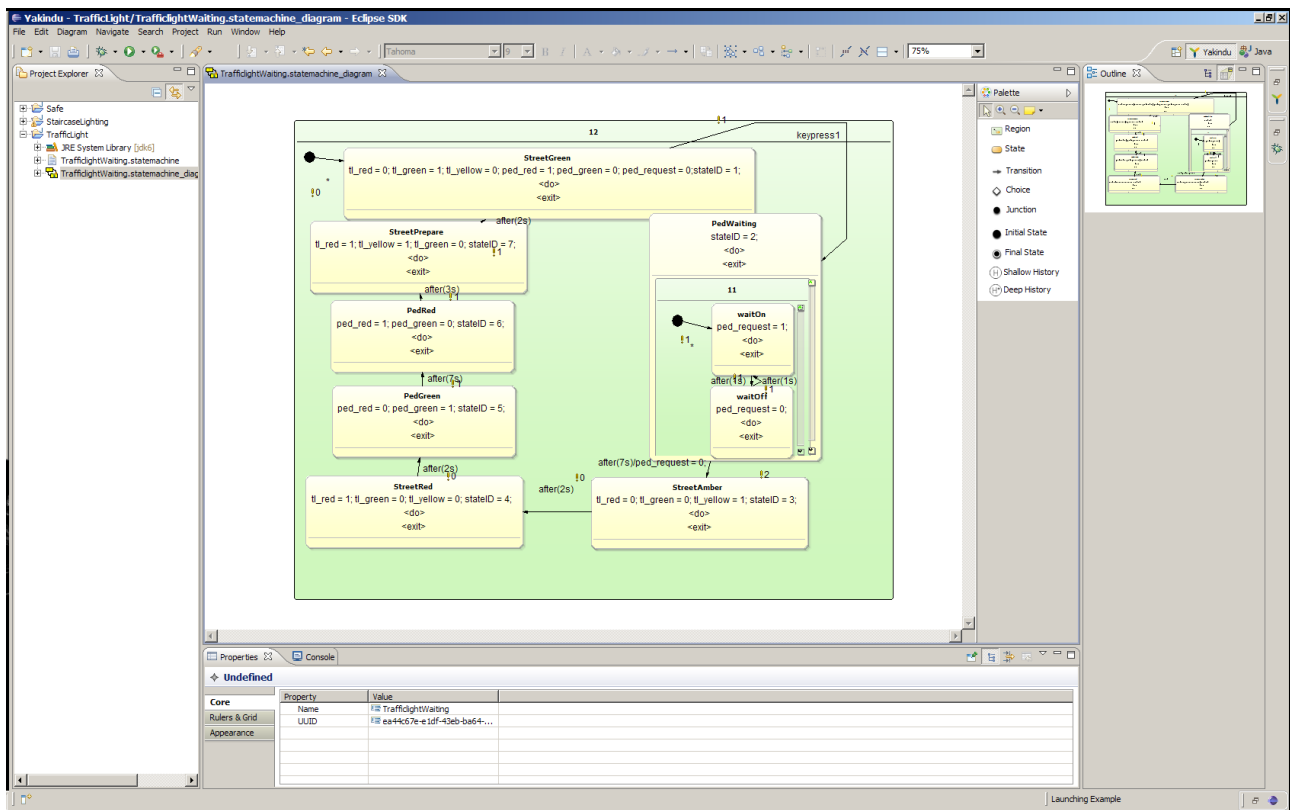


Figure 5.3: Traffic Light Example Project Contents

5.3 Generating Source Code

5.3.1 Create Xtend Project

The starting point to create Java source code from a YAKINDU Statechart model is to set up a Java project to contain the generated code. For the sake of simplicity, we will instead create a new Xtend project for this purpose (Xtend projects also possess the Java as well as the PDE Plugin project nature), as this allows to easily manage dependencies and execute MWE workflows. As outlined by Figures 5.4, 5.5, and 5.6, creating an Xtend project is easily done using the respective project creation wizard. Here, we choose *TrafficLightJavaDemo* as the name of the project.

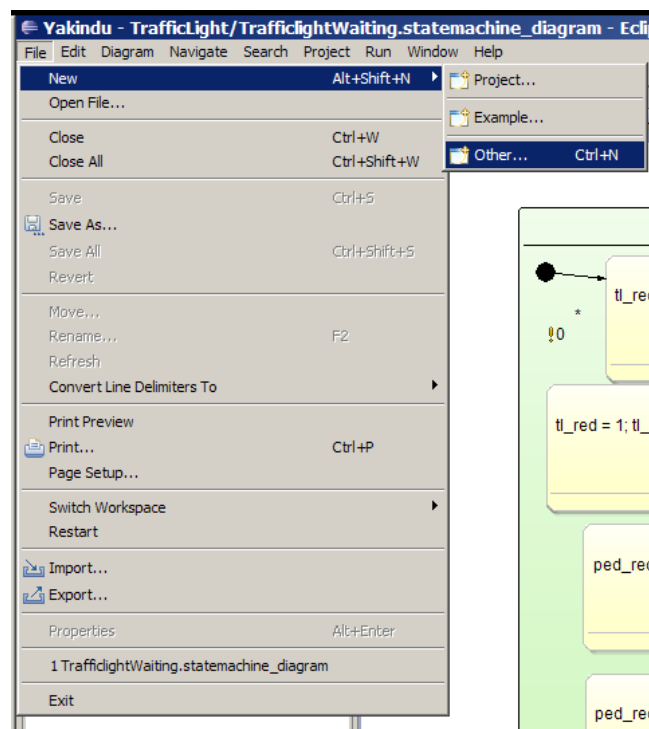


Figure 5.4: Open New Wizard

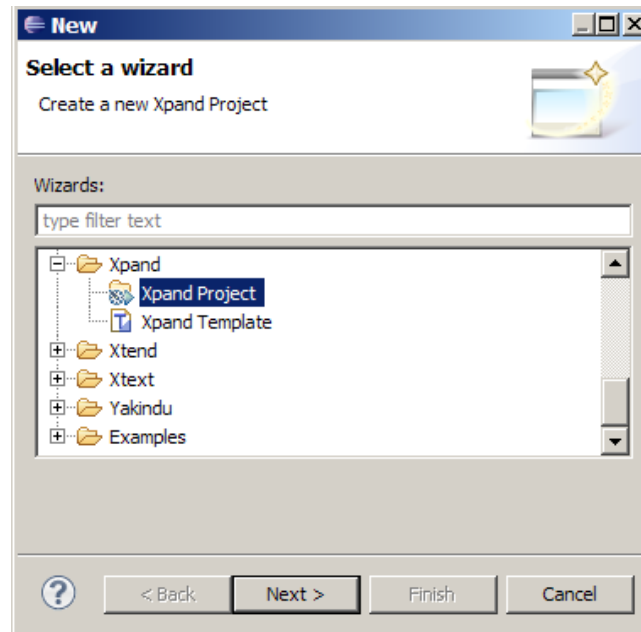


Figure 5.5: Selecting to Create a new Xpand Project

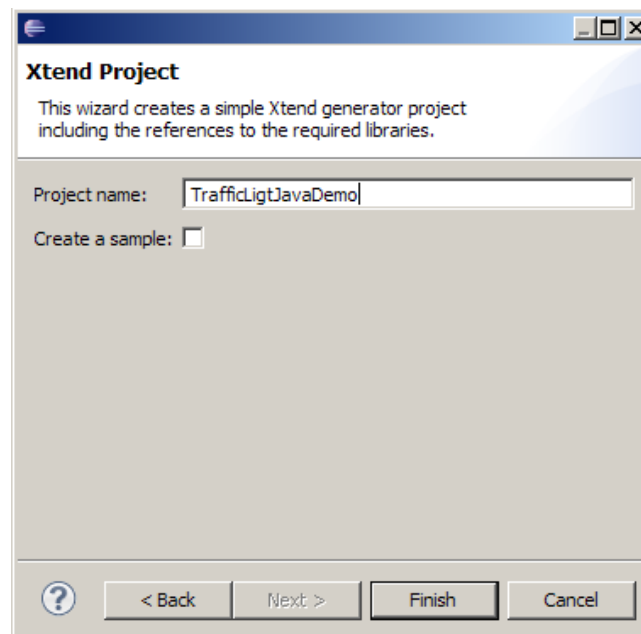


Figure 5.6: Xtend Project Creation Wizard

5.3.2 Manage Dependencies

Having created the *TrafficLightJavaDemo* project by means of the Xtend project wizard, the next step is to make the YAKINDU Java code generator plugin visible within the local project's classpath, so the Java code generation cartridge file, which is deployed by the code generator plugin, can be referenced from a local MWE workflow (to be set up as the next step). As we have created a Xpand project (which is implicitly also a PDE plugin project, as the Xpand project wizard adds the PDE plugin project nature to it), we may simply do so by adding the YAKINDU Statechart Java code generator plugin to the list of dependent plugins of our local project. As outlined by Figures 5.7 and 5.8, this can be simply achieved by opening the manifest editor on the MANIFEST.MF file located within the META-INF folder of the project, selecting the Dependencies tab (depicted by Figure 5.7, then by choosing to *Add* a new dependency and selecting the YAKINDU Statechart Java code generator plugin in the resulting dialog (depicted Figure 5.8). Also the following Plug-Ins are required:

- org.eclipse.jface.text
- org.eclipse.emf.mwe.utils
- org.eclipse.emf.ecore.xmi
- org.antlr.runtime
- org.eclipse.xtext
- org.eclipse.xtext.log4j
- com.ibm.icu
- org.eclipse.core.runtime
- org.eclipse.jdt.core

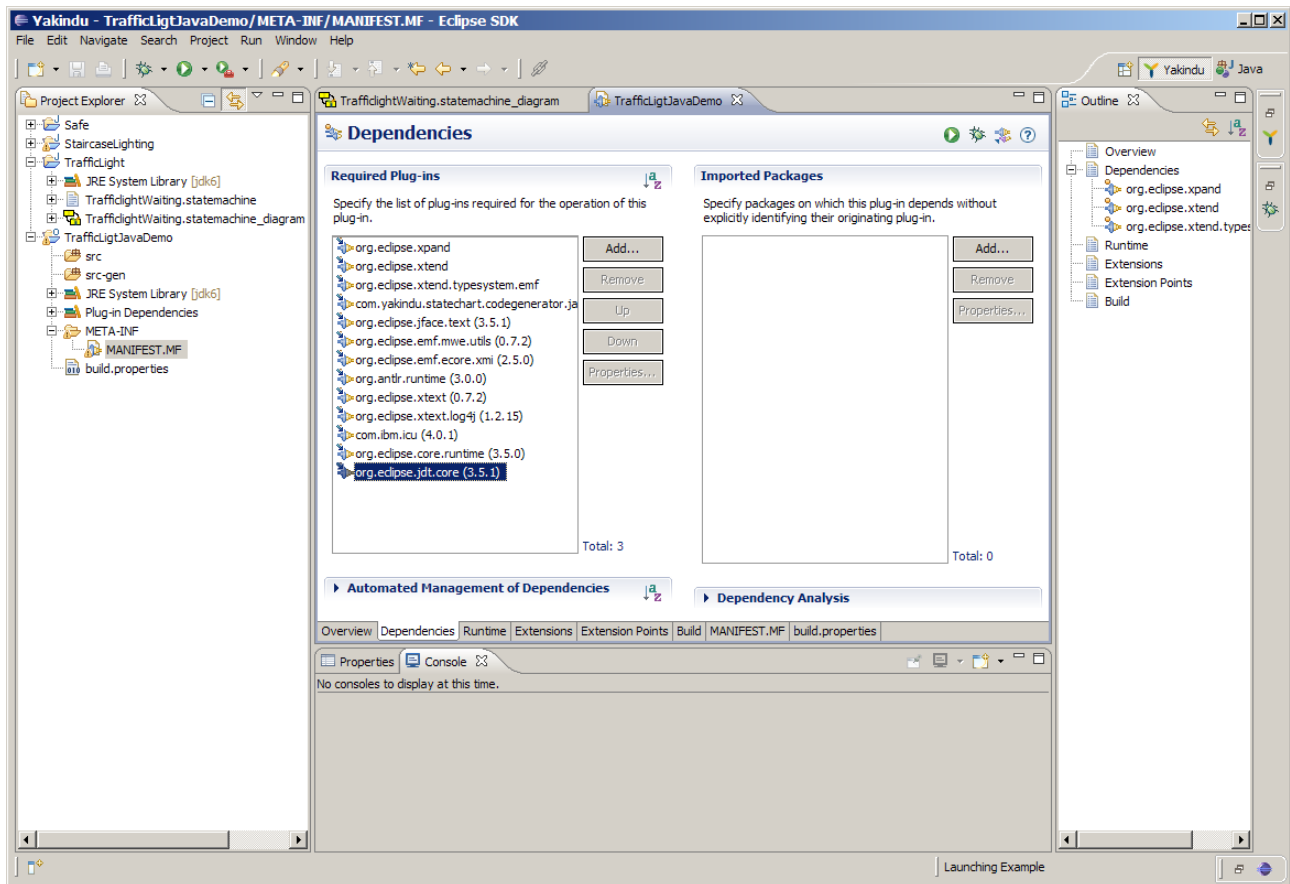


Figure 5.7: Manifest Editor

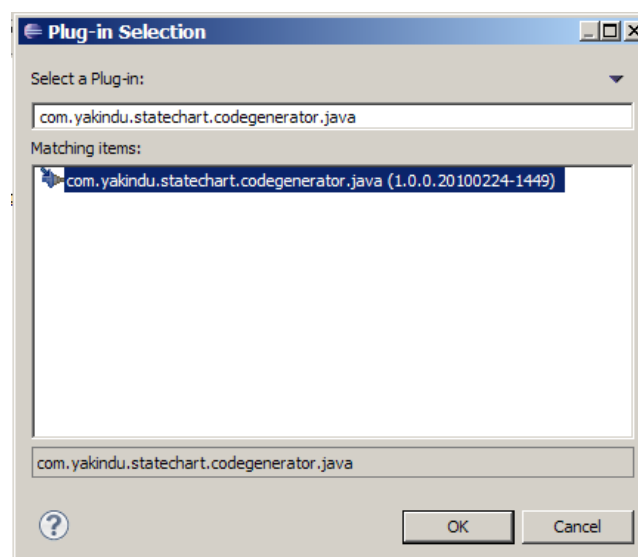


Figure 5.8: Adding Plugin Dependency

5.3.3 Create MWE Workflow File

As the YAKINDU Java code generator plugin provides an OAW workflow cartridge that may most simply be called from within a local MWE workflow file, the next step is to now set up such an MWE workflow file. This can be done as depicted by Figures 5.9, 5.10, and 5.11, choosing `generate.wme` as the name of the workflow file.

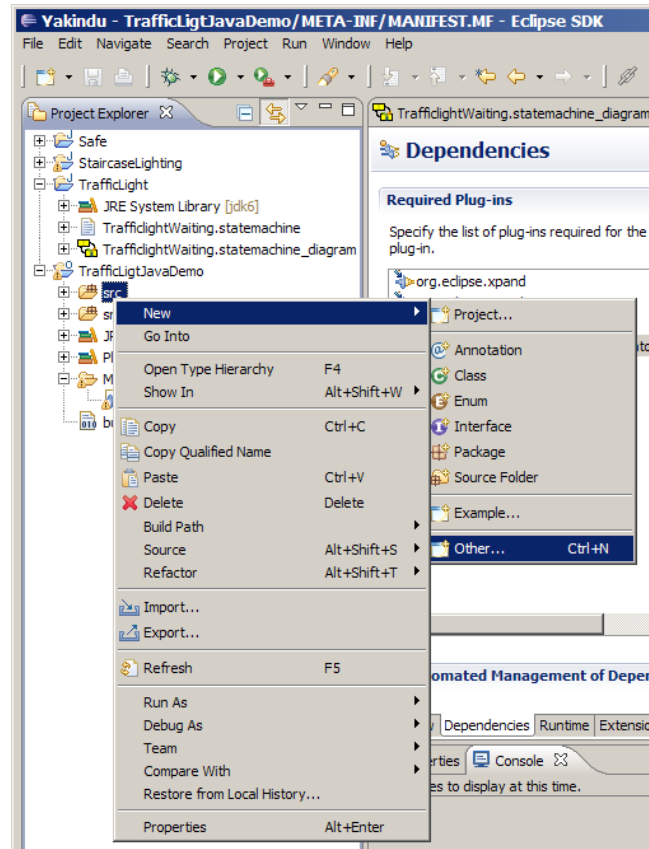


Figure 5.9: Opening New File Creation Wizard

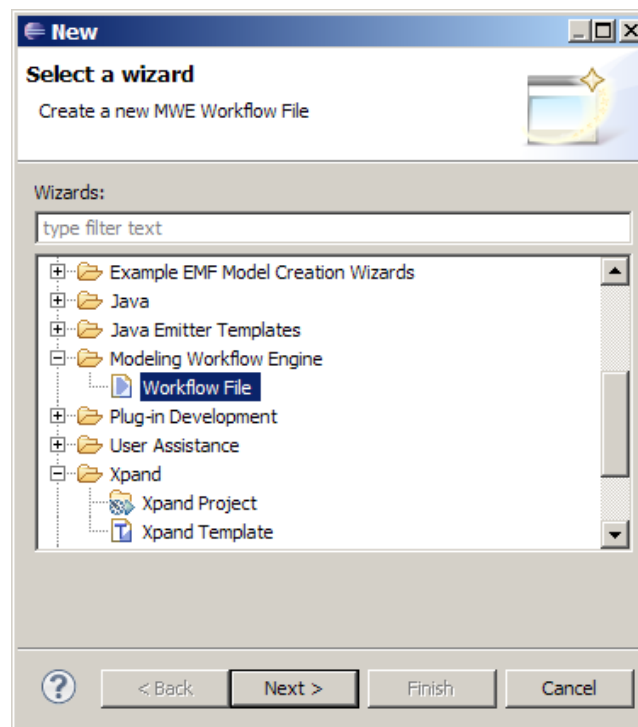


Figure 5.10: Selecting to Create a new MWE Workflow File

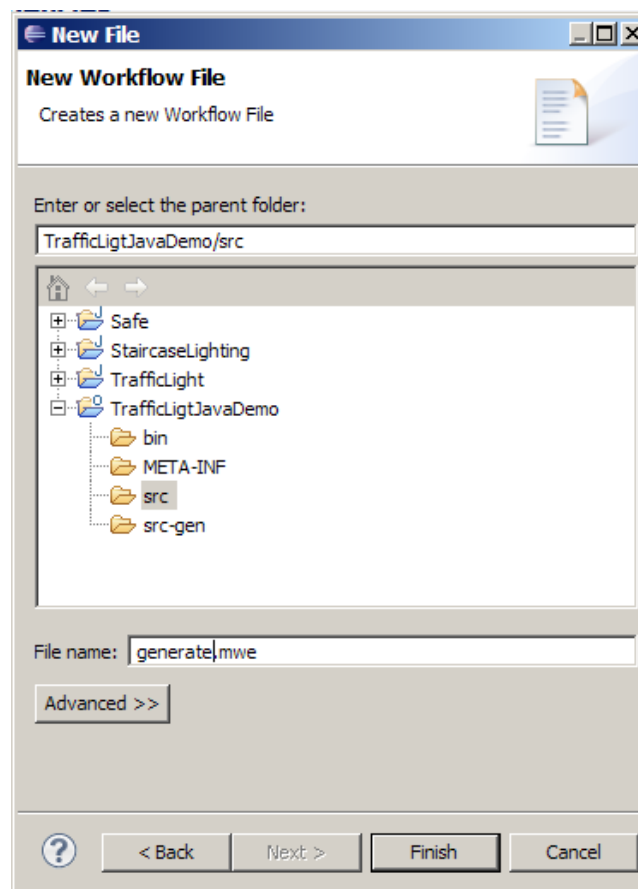


Figure 5.11: Specifying Name of MWE Workflow File

The contents of the workflow file has to be added as outlined by 5.12. Basically, the location of the input YAKINDU Statechart model (property `model`), as well as that of the output folder (property `src-gen`), the Java source code has to be generated into, have to be specified by means of workflow properties, being then passed to the YAKINDU Statechart Java code generator cartridge (*generate_java_defensive.oaw*), which performs the actual code generation.

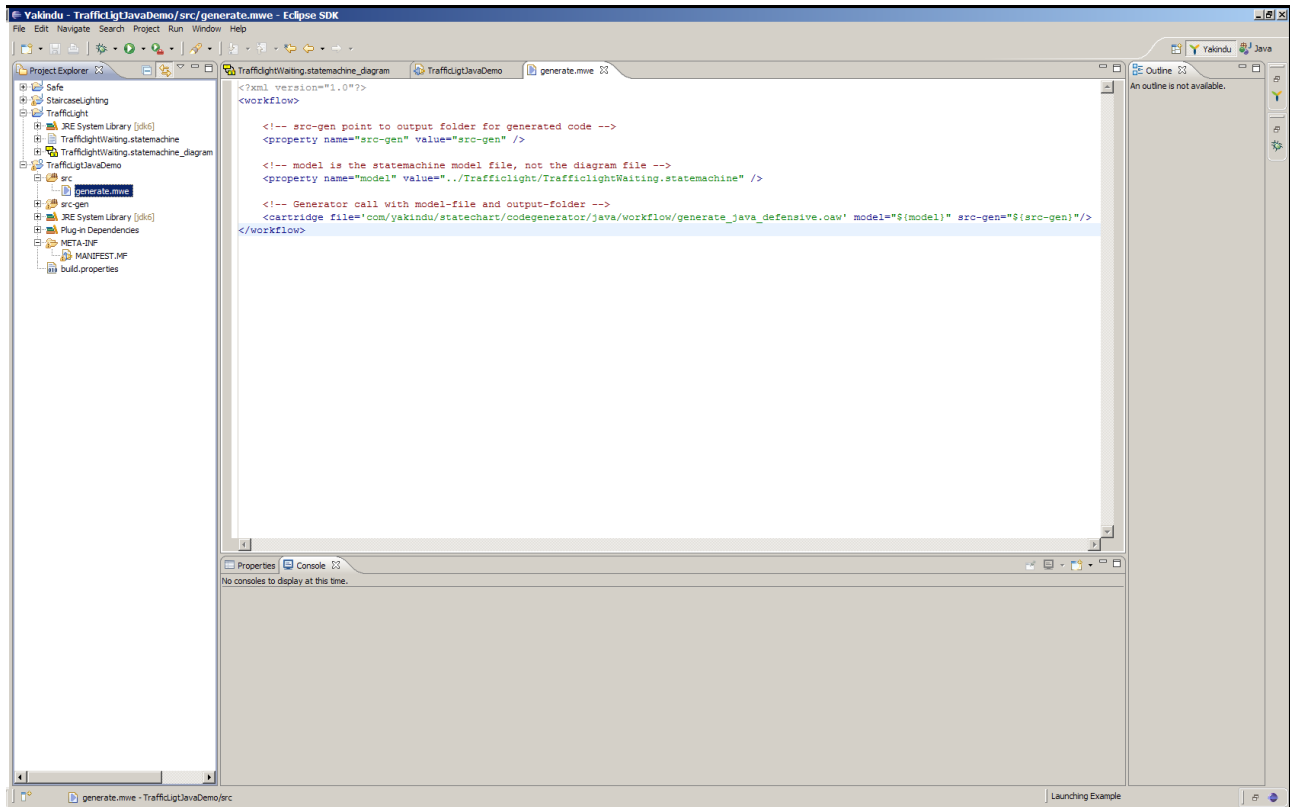


Figure 5.12:

In fact, the YAKINDU Statechart Java Codegenerator provides a lot of workflow cartridges, namely *generate_java.oaw*, *generate_java_me.oaw* and *generate_java_lejos.oaw*. Each has a defensive-version, which basically generate the same sort of Java source code, with the restriction that the *defensive* version generates additional defensive code, being used to check pre- and post-conditions as well as invariants at runtime. The Java-Codegenerator generates normal JavaSE-Code. The JavaME-Code is designed for mobile devices, which doesn't supports all Javaclasses. The third workflow cartridge generate Java-Code for leJOS, which can deploy on a LEGO Mindstorm.

5.3.4 Execute MWE Workflow

Having specified all relevant workflow properties, and having specified to call the code generator cartridge, the MWE workflow may then be simply executed as depicted by Figure 5.13. It will generate Java source code into the selected folder (here, the local `src-gen` folder within the *TrafficLightJavaDemo* project).

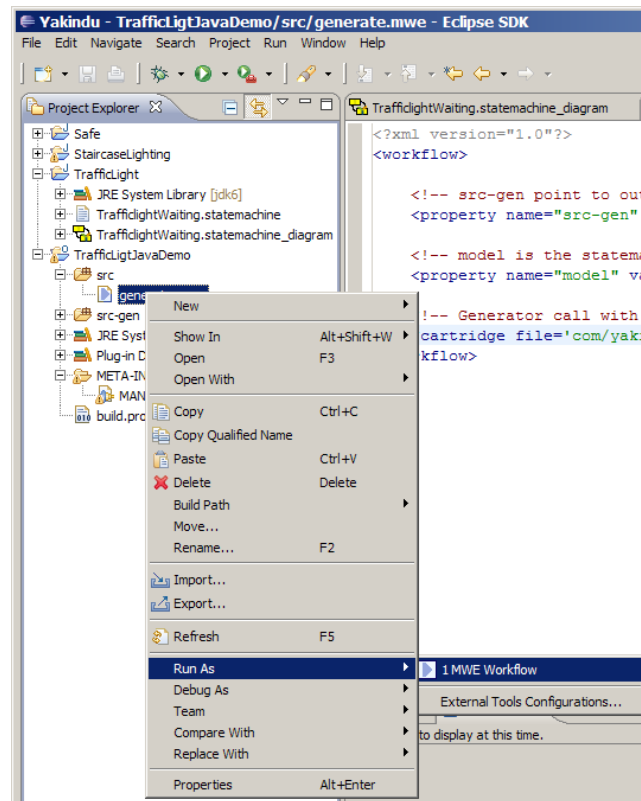


Figure 5.13: Contents of MWE Workflow File

5.4 The Generated Source Code

The Java source code generated by the YAKINDU Statechart Java code generator consists of a set of generic classes (located within the `com.yakindu.statechart` package) and a model specific state chart implementation class, named according to the respective Statechart, here `TrafficLightStatechart` within `trafficlight` package), which is outlined by Figure 5.14. It is the single class that may be directly used by clients to integrate the generated code with manually written or other generated code.

To obtain a new instance of the state chart implementation, the generated implementation class offers a static method named `createInstance()`. It returns a completely initialized state chart instance, which may be started (via `enter()`), cyclically executed (via `runCycle()`), and stopped again (via `leave()`).

Feeding the state chart with external events is done by passing respective event constants (declared within the state chart implementation class) into the state chart instance (via `setEvent()`) prior to executing a new cycle (via `runCycle()`). Setting variable values is similarly possible via respective selector methods generated into the implementation class (a pair of getter and setter for each variable of the state chart). On each call to `runCycle()`, the state chart reacts on all events currently raised (since the completion of the prior cycle) and reads the variable values as they are set when calling `runCycle()`. After `runCycle()` has been completed, all events raised prior to its execution will be cleared and the variables reflect the values as they result from the reaction of the state chart.

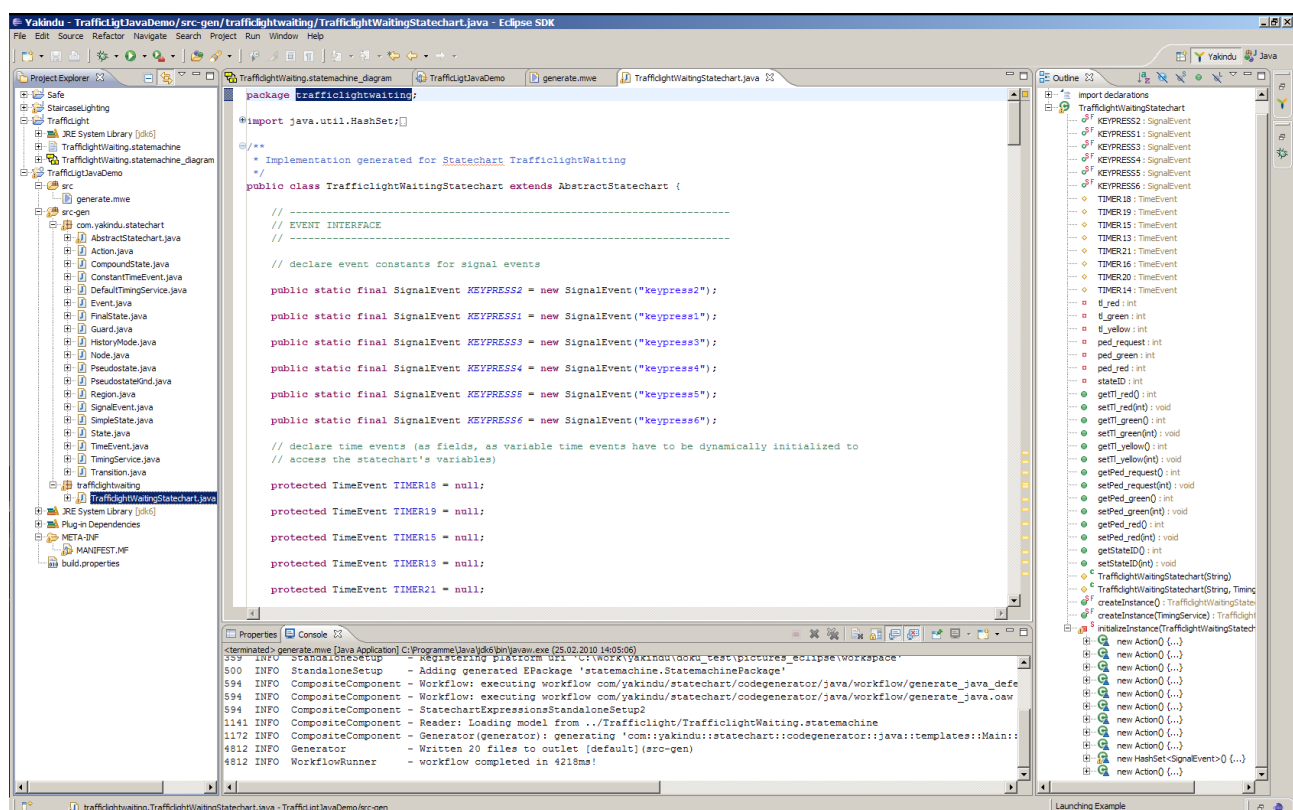


Figure 5.14:

5.5 Integrating Generated Java Code

A demonstration example that reflects the integration of the Java code, generated for the *TrafficLight* example with manually written code is deployed with the YAKINDU Statechart tools and may be created using the example wizard as outlined by Figures 5.15 and 5.16.

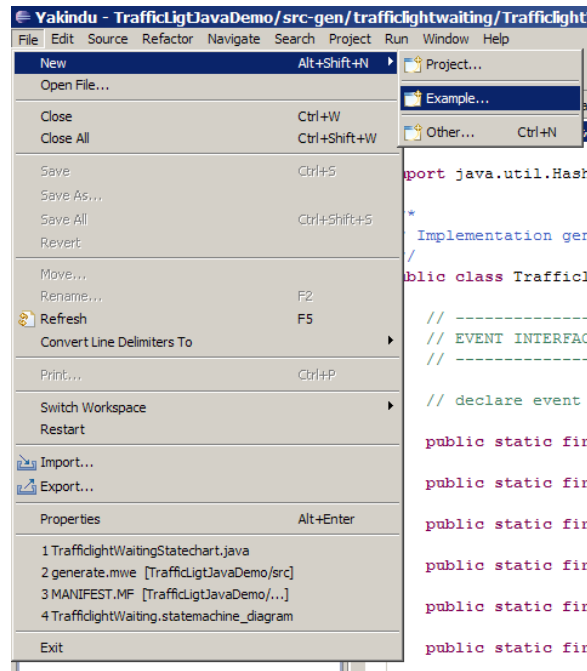


Figure 5.15: Opening the Example Wizard

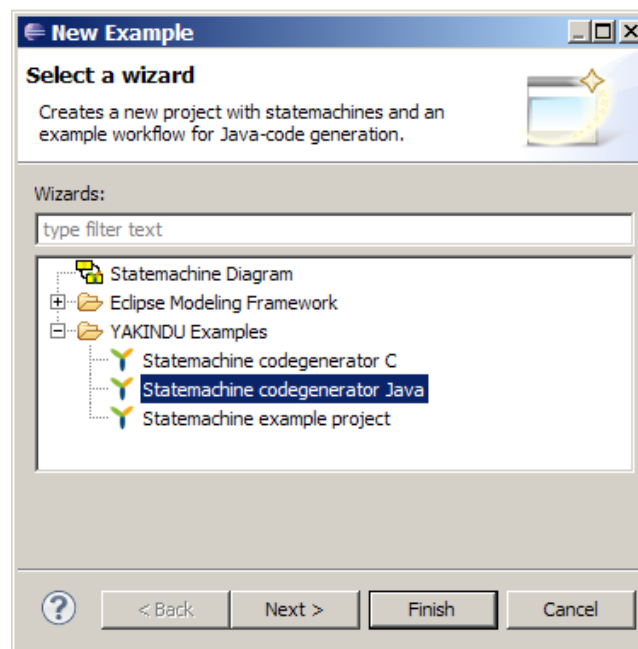


Figure 5.16: Creating a new Java Code Generator Example Project

As depicted by Figure 5.17 it contains some manually written code (within the `src` folder) that realizes a `Draw2d` visualization of a traffic light, being adapted (within the `CrossingDemo` main class) to the generated source code of the *TrafficLight* example (contained in the `src-gen` folder of the example project).

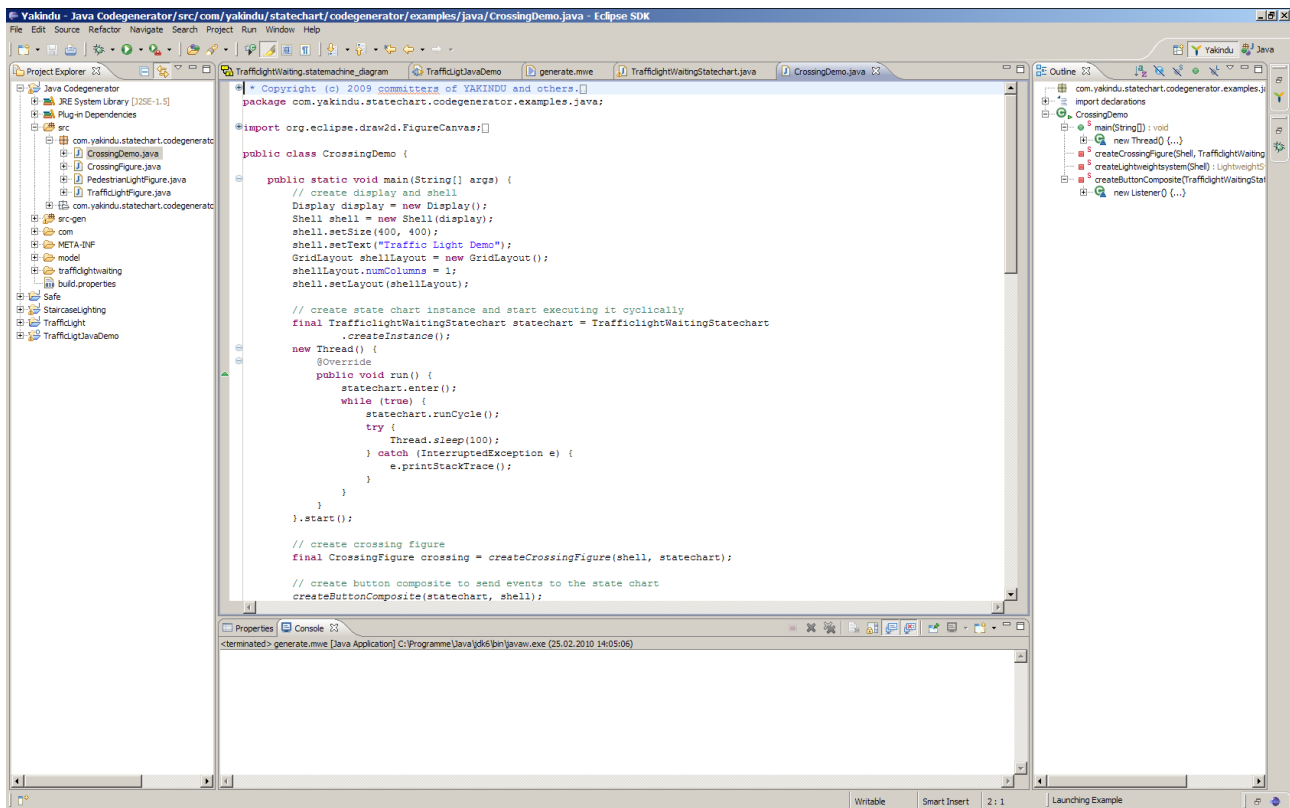


Figure 5.17: Traffic Light Java Demo Project Contents

As outlined in Figures 5.18 and 5.19, the `CrossingDemo` main class within the example project may be executed as a Java application. It shows a visualization of the *TrafficLight* state chart variable values (by means of colors of the traffic and pedestrian lights contained) and offers a button group, which prolongs respectively named events to the *TrafficLight* state chart.

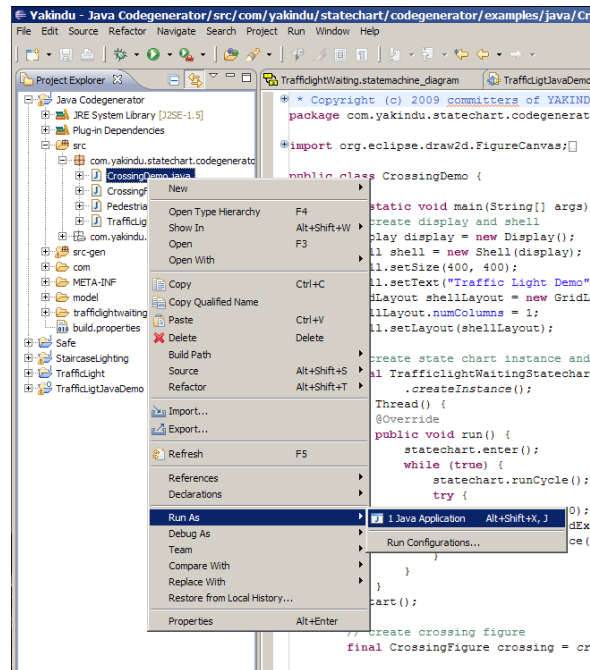


Figure 5.18: Running CrossingDemo as Java Application

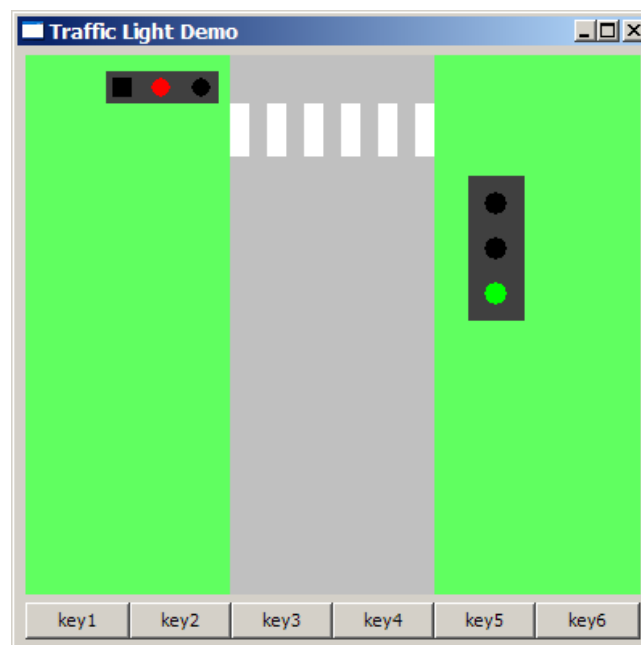


Figure 5.19: CrossingDemo Application

Chapter 6

UML Transformation

6.1 Introduction

In our previous examples we created a YAKINDU state chart with the YAKINDU Statechart Editor. However, it is also possible to import an existing UML2 state machine or create one with your favourite UML2 tool (You may also use the UML2 Editor which ships with the eclipse modelling distribution). The tool of your choice must support "EMF UML2". In our examples we are going to use the open source tool Papyrus UML (<http://www.papyrusuml.org>).

If you want to use an UML2 modelling tool instead of the YAKINDU Statechart Editor or you have an existing UML2 state machine, you need to consider that the YAKINDU state charts are somehow different from the UML2 state machines. To fill this gap you have to extend your UML2 state machine (In future releases another approach may also be available).

6.2 UML2 model

As for the code generators, an example project for transformation of UML2 state machines to YAKINDU state charts is also available. You can open the examples (see Figure 6.1) and compare them with the generated YAKINDU state charts.

Some information, which cannot be modelled in UML2 are generated automatically, or as mentioned before, can be specified somehow else. For example: Every Transition and every Region in a YAKINDU state chart needs a priority. On how to set a priority will be discussed later. Lets have a look at an example project first.

6.3 The example project

The example is available within eclipse under **File** → **New** → **Example...** within the category **YAKINDU Examples**. Select "UML to statechart example" and Finish the dialog. A new project

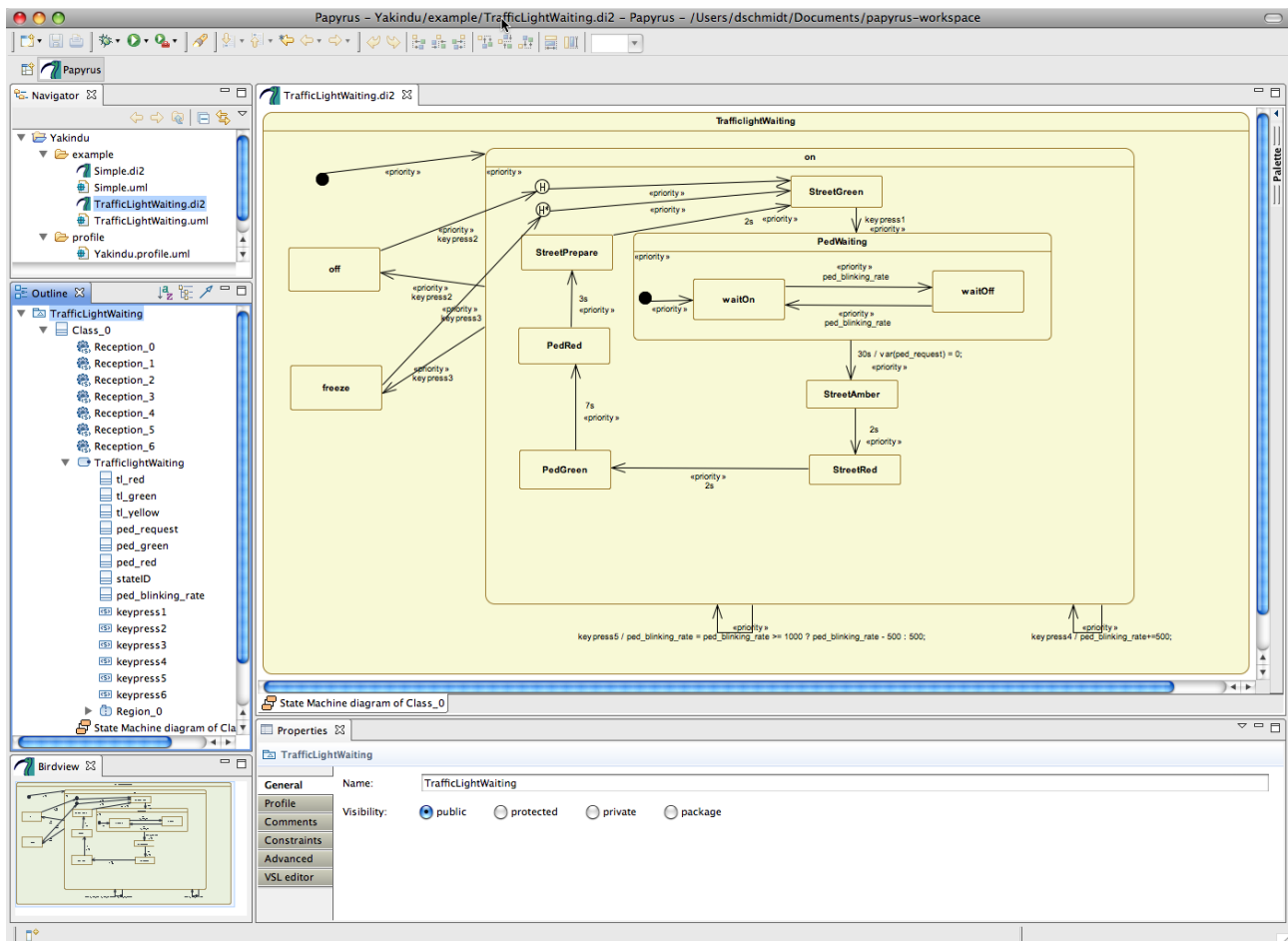


Figure 6.1: UML2 state machine in Papyrus

"UML_Transformation" containing two Papyrus UML diagrams inclusive ".di2" files for Papyrus UML will be created.

To generate the YAKINDU state chart, right-click on **generate_demo.oaw** and select **Run As** → **oAW Workflow** (Compare Figure 6.2). The output of the transformation is saved in the new folder src-gen and the status of the run is available in the Console view of eclipse, which will be automatically opened in the right corner at the bottom of the window.

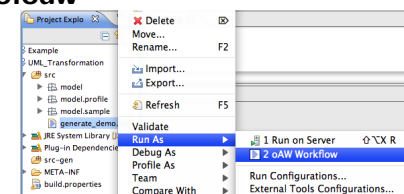


Figure 6.2: Run UML transformation workflow

The result is a *.statemachine file which can be found in the src-gen folder within your project. The output folder, where your statemachines will be generated and the UML2 model can be defined in the generatordemo.oaw. The next step, which should follow, is to generate a diagram for your model. This is not automatically done, but with a right-click on the statemachine file you can initialize a diagram (see figure 6.3). Because the layout information are not stored inside UML2 you have to draft your diagram by hand.

The generated state chart can be edited, simulated and code can be generated from it. These steps were described in previous chapters. If you want to generate code from the state chart see Chapter

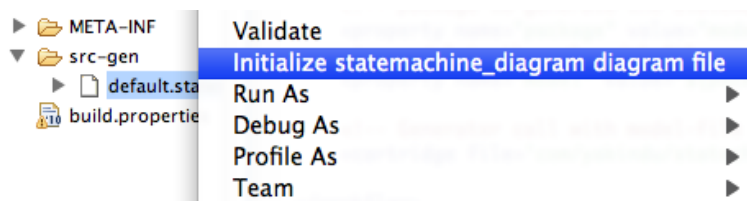


Figure 6.3: Initialize Diagram

4 for C and Chapter 5 for Java. If the model is a valid YAKINDU state chart a simulation is also possible (see Chapter 3.2.8).

6.4 How does it work

The UML2 state machine and the YAKINDU state chart are quite similar. Nevertheless, there are a few differences that need to be cared of.

6.4.1 Name mapping

The naming is the smallest gap. The following table shows the name mapping between the UML2 state machine and the YAKINDU state chart.

UML2	YAKINDU
StateMachine	Statechart
Region	Region
Transition	Transition
Vertex	Node
State	State
FinalState	FinalState
Pseudostate	Pseudostate

Table 6.1: UML2 - YAKINDU Mapping

As you can see, there are only marginal differences. Besides that, YAKINDU state chart also defines new elements which are not present in the UML2.

6.4.2 New elements

The new elements are Variable and Event. As those can't be mapped 1:1, some conventions are needed. The default behavior of the transformation is to ignore those elements. This will result in an incomplete YAKINDU state chart (therefore no simulation is possible). To prevent this, you have to extend your UML2 state machine.

6.4.3 Limitations

There are a few UML Elements that are currently not supported by the transformation. That includes the PseudoStates

- join
- fork
- entryPoint
- exitPoint
- and terminate.

A transformation that encounters such an element will fail.

6.4.4 Transformation Cartridge

The Transformation Cartridge allows you to integrate the YAKINDU Transformation into your oAW Workflow. The cartridge expects the following parameters.

Parameter	Description
umlModel	A path to your UML2 Model file.
srcgen	The output folder where the generated *.statemachine files should be created.
package	The root directory for the transformation.

Table 6.2: YAKINDU Transformation Cartridge Parameters

In your oAW Workflow it may look like this:

```
...
<property name="src-gen" value="src-gen" />
<property name="package" value="model" />
<property name="model" value="${package}/example.uml" />

<cartridge file='com/yakindu/statechart/transformation/uml/transform.oaw'
  umlModel="${model}"
  srcgen="${src-gen}"
  package="${package}" />
...
```

6.5 Extending an UML2 state machine

Since there aren't UML2 Elements called Variable or Event, those have to be created with the existing UML2 elements and the YAKINDU UML2 profile. The profile allows you to give Transitions and Regions a priority and to describe Events as Signals and Variables as Classes.

First you need to import the YAKINDU UML2 profile. Open your *.uml file with Papyrus and click on **UML Editor** → **Load Resource...** and select the YAKINDU UML2 profile. After that select your model and click on **UML Editor** → **Package** → **Apply Profile**. A new window pops up where you add the YAKINDU UML2 profile to your model. Press **Ok** and save the *.uml file. Now that the profile has been loaded you need to apply the stereotypes to various elements. You can do this with the UML Editor which has been used until now or you can do this with Papyrus UML. For our next task we are going to use Papyrus UML.

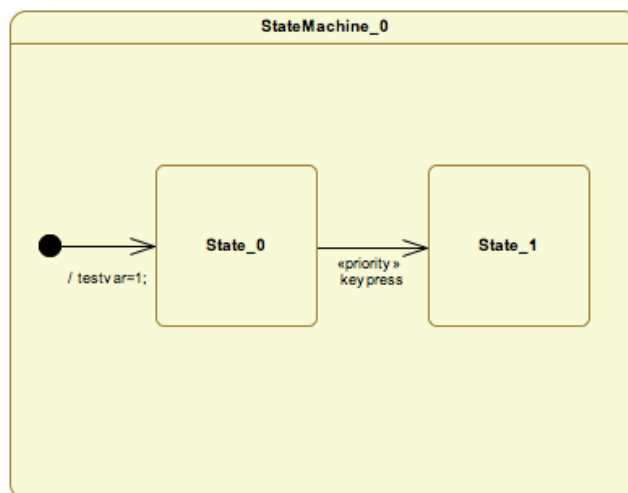


Figure 6.4: UML2 state machine example

Open the related *.di2 file. It should look somehow like Figure 6.4. The YAKINDU UML2 profile allows us to apply the stereotypes Priority, Variable and Event. Priority can be applied to Transitions and Regions, Variable to Classes and Event to Signals. First, we want to give every Transition and every Region a Priority. To do so, select a Transition/Region. In the Properties View you need to click on **Profile**. Press the "+" button and select Priority (see figure 6.5). After that you can change the value for Priority (default: 0) (see figure 6.6). That's it! Repeat this for every Transition/Region in your State machine.

Now we are going to add the Stereotypes Event to our Signals. We are doing this again with the UML Editor because Papyrus seems to have some problems with this Stereotype.

Select a Signal and click on **UML Editor** → **Element** → **Apply Stereotype** (see figure 6.7). Add YAKINDU::Event and press **OK**. In the Properties View you can see now a Property Event with some Attributes. Change those Attributes to your needs (see figure 6.8). Repeat this for every Signal in your state machine.

To add Variables to your UML2 state machine you have to create the corresponding Classes first.

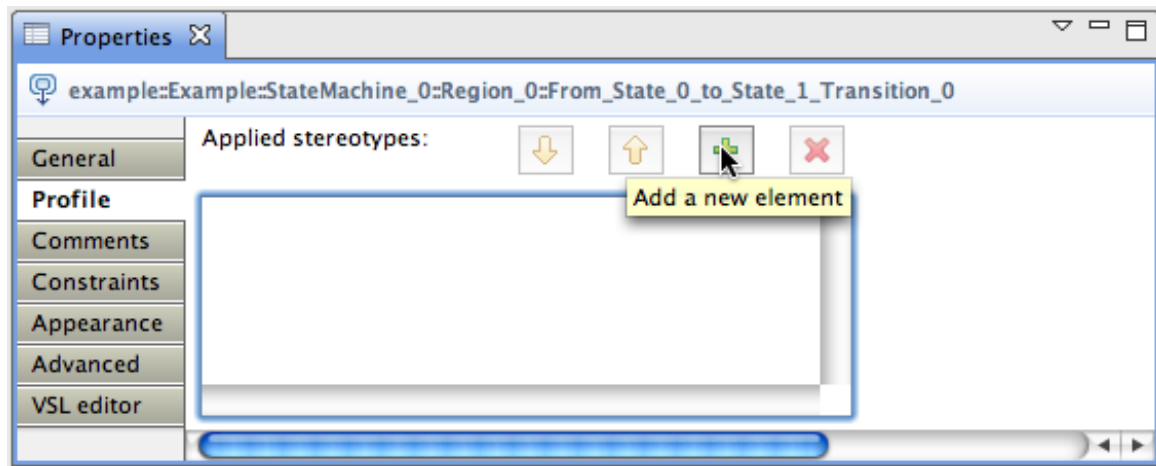


Figure 6.5: Add a Stereotype to an UML2 Element

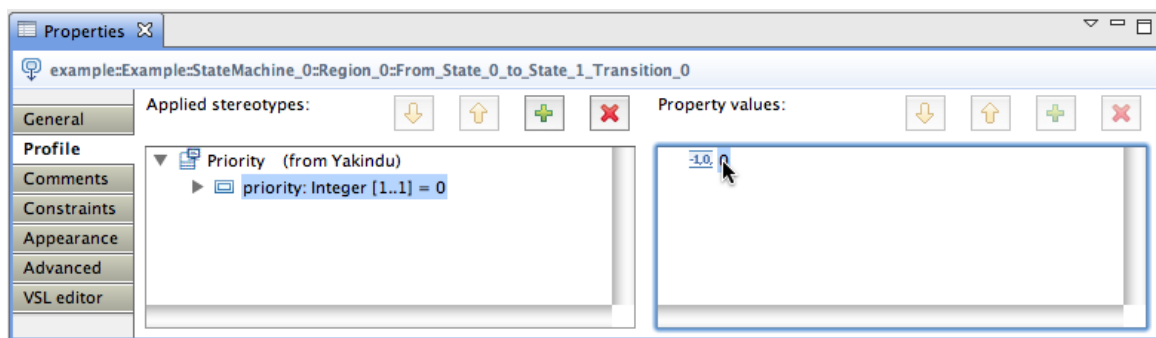


Figure 6.6: Change the Priority

Those have to be named after the Variables in the Expressions of your Transitions (see figure 6.9). If you created a Class, select it and click on **UML Editor** → **Element** → **Apply Stereotype**. Add **YAKINDU::Variable** and press **OK**. In the Properties View you can change now the Attributes of Variable to your needs. Repeat this for every Variable in you Expressions.

It is important to know that Signals and Classes that should be recognized as Events and Variables must be created under the state machine in the UML2 model. This is necessary due to the fact that any number of state machines can be in a UML2 model.

If you followed all the described steps you are now able to generate a valid YAKINDU state chart.

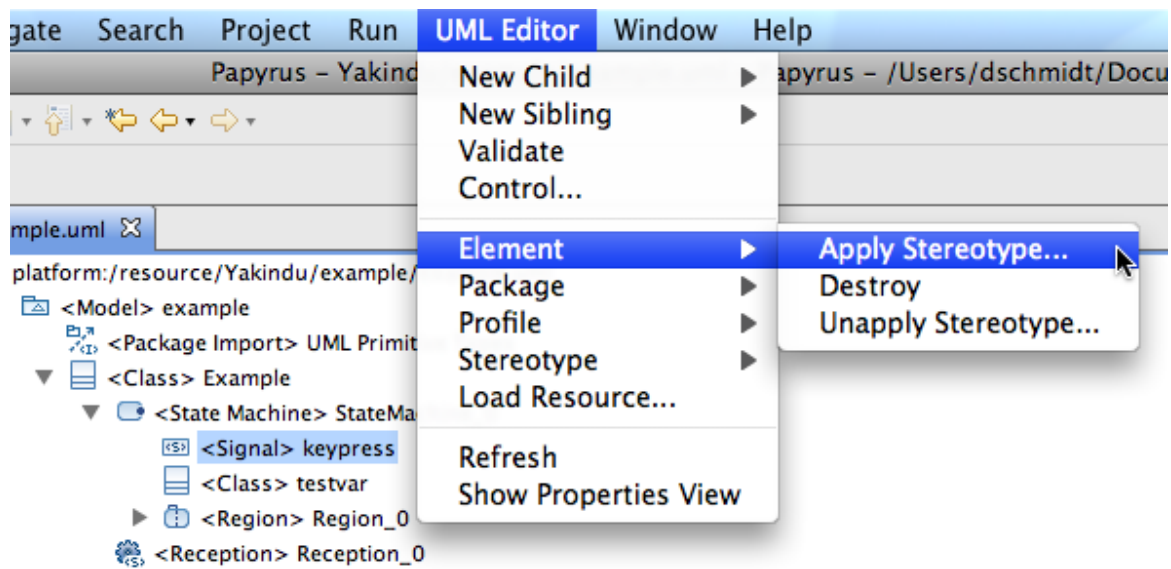


Figure 6.7: Apply Stereotype

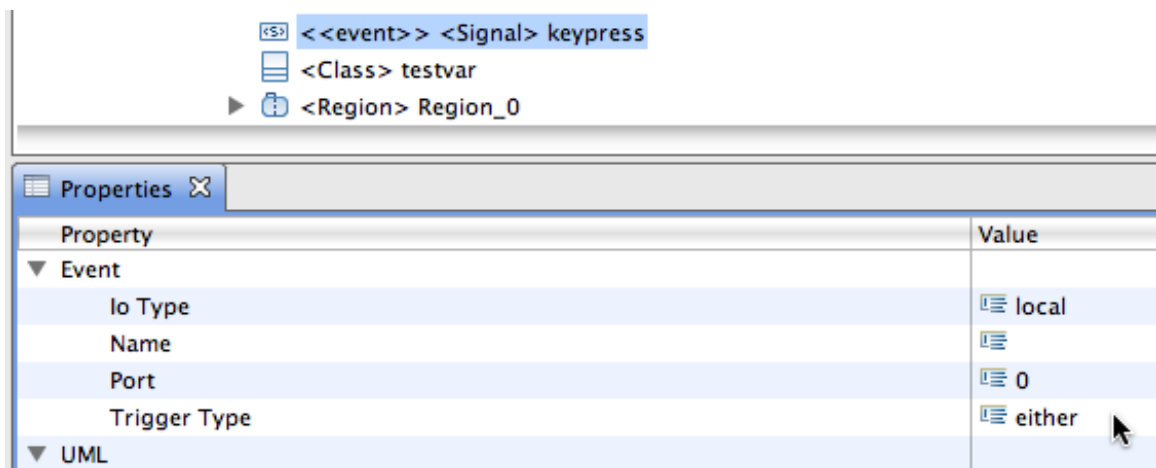


Figure 6.8: Change Attributes

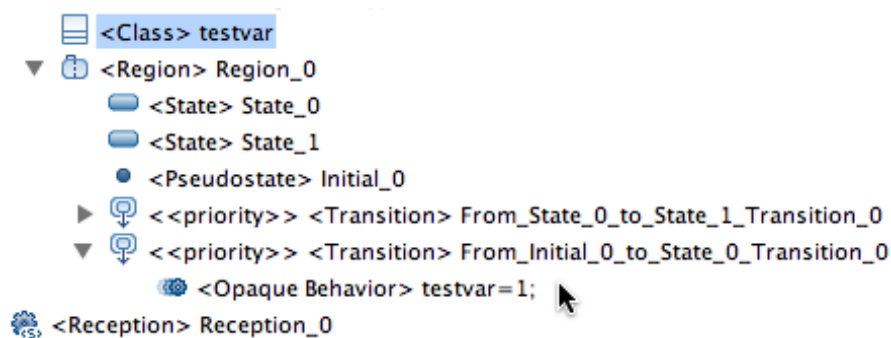


Figure 6.9: Class named after a Variable