**YAKINDU Research Report**

# Model Driven Development

Model Based Software Engineering makes systematic use of (formal) engineering models as primary engineering artifacts throughout the overall engineering life cycle.

Model Driven Software Development (MDSD) is a generic term for techniques that transform formal models into executable software.

Simplification by abstraction:

• hide complexity

• focus on domain relevant aspects

• Generator encapsulates technical details

Transformation to target platform using generators (or interpreters)

• avoid redundancy (DRY: Don‘t Repeat Yourself)

• enforce architecture and design guidelines

• improve quality / reduce error rate / fix errors once

• improve productivity

• More flexibility for variants

## Abstraction Levels

## When is MDD applicable?

• Same solution for the same problem

• Schematic technical implementation

• Differences can be specified in the model

• Less effort for specification than for implementation

• Critical mass will be reached

• Extent of models

• Frequency of change

• Number of target systems

• Many variants

# Yakindu

Yakindu is an open-source-toolkit for the model-driven development of embedded systems. Through the systematic use of models, it aims at an integrated development process as well as an increase in quality and maintainability. With it the accompanying increase in efficiency addresses important challenges with the development of increasingly complex embedded-software-systems.

The Yakindu-toolkit supports the development of both reactive, event-driven and data flow-oriented systems with the help of finite-state machines and block diagrams. The continuous support begins with graphical modeling tools, includes integrated validation and simulation that allows for the early assessment of the models and offers efficient code-generators for the generation of source code for a target platform. Technologically, it is based on the Eclipse-platform, integrates itself seamlessly into Eclipse-based workbenches, and extends this in the direction of model-driven development.

## Who is behind YAKINDU Statechart Tools 2?

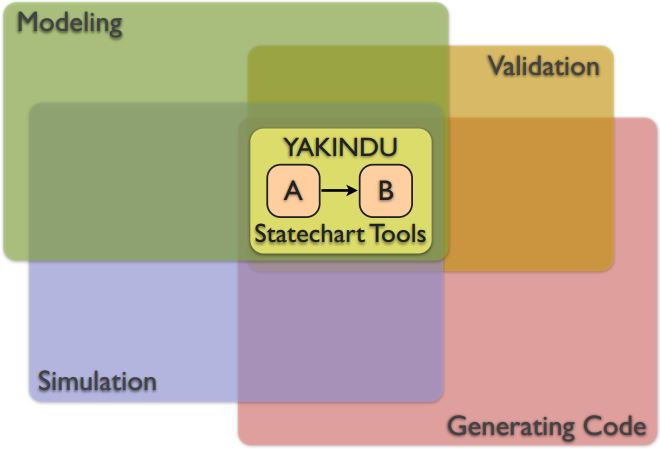
The main part of YAKINDU SCT 2 is an open source project ([www.yakindu.org](http://www.yakindu.org/) ). Most of the developers work for itemis; a well-known consulting company specialized on model-based development.

## Who uses the YAKINDU Statechart Tools 2?

Initially the YAKINDU SCT 2 were designed for the embedded systems industry: automotive, system controls, vending machines etc. However, it brings benefit to everyone who needs to design, simulate and develop behavior. People can use the YAKINDU SCT 2 to generate Java, C, or C++.

## Activities and Tools

# Statechart Tools

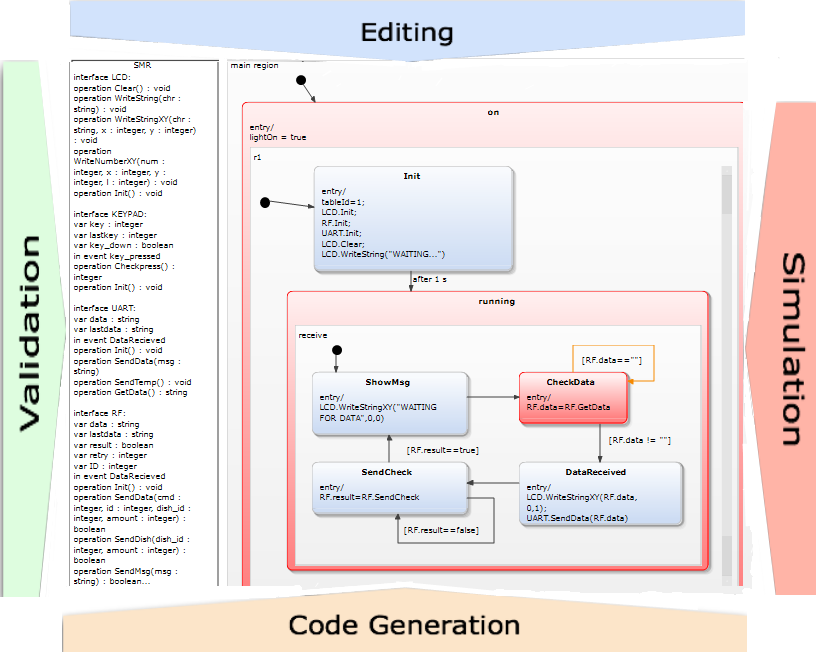
YAKINDU Statechart Tools (SCT) provides an integrated modeling environment for the specification and development of reactive, event-driven systems based on the concept of statecharts. It is an easy to use tool that features sophisticated graphical state chart editing, **validation and simulation of statecharts as well as code generation.

## **YAKINDU Statecharts**

State charts are based on the formalism of state machines that has been well proven for the specification and implementation of reactive event-driven systems. This approach leads to a decomposition of a systems behavior into a set of states that defines the valid reactions to external events along with timing conditions. This results in an intuitive and maintainable description of the overall behavior.

YAKINDU SCT allows modeling based on elements know from Harel-statecharts. All essential concepts such as extended state variables, hierarchical states, sub-statecharts, orthogonal states (also known as And-States or parallel regions), time events, and history states are supported. Event-driven as well as cycle-based execution models are supported.

YAKINDU statecharts are self-contained – they not only contain the definition of states and state transitions, but also the definition of the statechart interface. Therefore, implementations that are generated from the statecharts are complete and provide a well-defined interface that can be easily integrated with any application code.

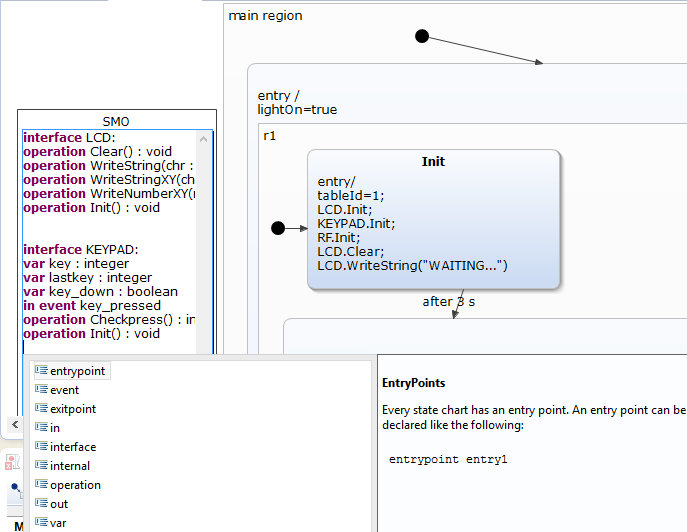
The following graph shows these features and their relation to each other:

## **Editing**

YAKINDU statecharts feature an intuitive combination of graphical and textual notation. While states, transitions, and state hierarchies are graphical elements, all declarations and actions are specified using a textual notation. The usability of the statechart editor is optimized for working with these statecharts.

Every model element, including the textual parts, are visible on the canvas and can be edited directly in the diagram or alternatively using specific form based view without diving into any modal dialog. As a result, everything is available at the user’s fingertips and not hidden in dialogs.

Editor with inline text editing syntax highlighting, completion and error marker.

The editor also includes IDE-like editing support for the textual parts, which includes syntax highlighting, completion and validation.

## **Validation**

The validation of statecharts includes syntax and semantic checks of the complete state chart. Examples of validations are the detection of unreachable states, dead ends, and references to unknown events. These validation constraints are live checked during editing. In case a constraint is violated, this is visualized by warning and error markers, which are attached to the faulty model elements. By this the user gets direct and immediate feedback on the validation state of the statecharts. This assists in detecting problems early on and avoids time-consuming error resolution.

## **Simulation**

In addition to the structural validation, checking the dynamic semantics is crucial. It is not possible to determine the correctness of a statechart just by visual examination. Thus, the user must be able to execute the statecharts he is working on. The integrated statechart simulation engine addresses these needs. The user can execute statecharts directly within the modeling environment. Active states are directly highlighted in the statechart editor and a dedicated simulation perspective features access to execution controls (start, stop, pause, resume), inspection and setting variables, as well as raising events. The tight integration of modeling and simulation environment enables the user to rapidly switch between design and validation tasks without any hurdles.

The simulation engine uses a virtual time space that supports scaling of the simulation time. This allows the simulation of the statechart with very tight or extremely broad timing.

## **Generating Code**

YAKINDU SCT includes code generators for Java, C and C++. The code generators follow a ‘code-only’ approach and do not rely on any additional runtime library. The generated code provides a well-defined interface and can be integrated easily with any client code. The generated code is also readable and structured in such a way that allows for very efficient execution.

## **Customizations**

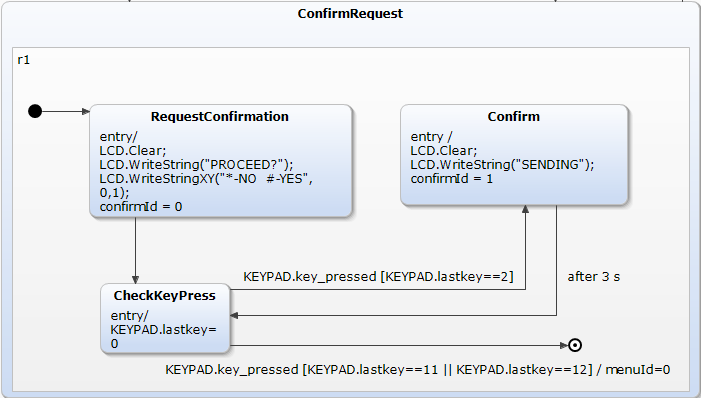
Even though YAKINDU SCT is ready to use out of the box, it is open and designed for extensibility, so it can serve as a basis for more specific statechart modeling and implementation solutions. Nearly every aspect of the tool is customizable. Typical cases of customization are platform specific extensions of the standard code generators.

The ability to extend or change the statechart language provides a very powerful way to build specialized statechart modeling solutions with extremely reduced effort compared to developing a state chart tool from scratch.

## Finite State Machines

A finite number of states defines a system. The behavior of the system depends on the current state; behaves differently to events depending on the state. The current state is determined by the history of the state machine.

**An example**



**Variables**

• Hold quantitative values

• Accessible

• Modified by actions

**Events**

• Trigger transitions

• And actions

**Transitions**

• Switch between states

• Triggered by events

• Guarded by Boolean expressions

• execute actions

• „takes no time“

**States**

• Behavioral equivalence classes

• Execute actions on entry, exit & continuously (do)

• Stable between events

Run to completion step is the atomic operation of a state machine which transforms from one stable state to the next. The system does not remain between states.

## Statecharts semantics

Statecharts semantics using an own, simple meta model and close to UML state machines. However, YSCs are self-contained with an interface well defined by events and variables. Core execution semantics are cycle-driven and not event-driven, so they allows processing concurrent events and event driven behavior can be defined on top. Time is an abstract concept for statecharts. Time control is delegated to the environment. Model interpreter and different flavors of generated code follow the same core semantics.

# Getting started

## Installation

### Prerequisites

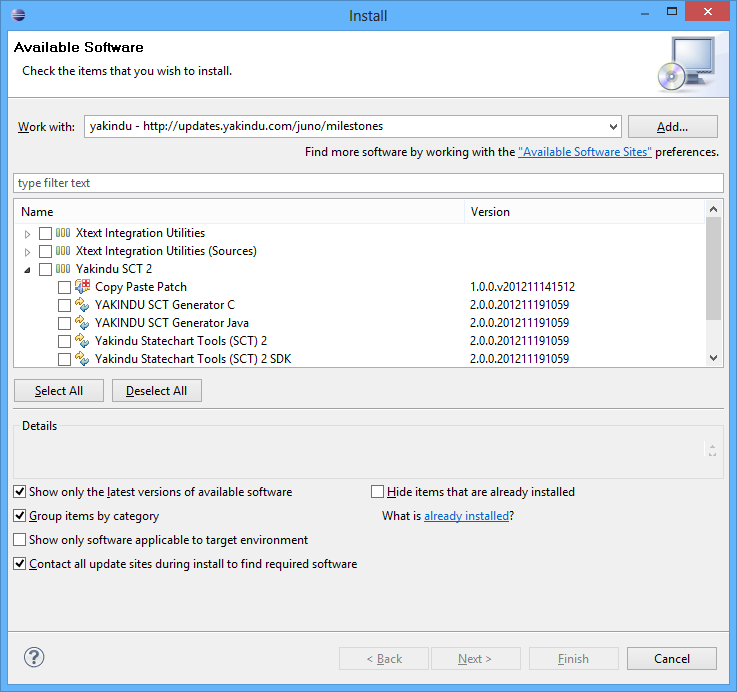
The **Yakindu Statechart Tools 2** are built upon Java and Xtext. Therefore, you need to have installed a *Java Runtime Environment* and Xtext installed. The easier way to get this is to install the *pre-configured Xtext contribution*.

The **Yakindu Statechart Tools 2** need **Eclipse Indigo 3.7** or higher and **Xtext 2.0.1** exactly (no higher) to work correctly.

### Installing the YAKINDU-Plug-Ins

You install the Yakindu Plug-Ins from the update site: *http://updates.yakindu.com/juno/milestones*

* Click **Help** > **Install new software...** and **Add...** the update site Yakindu SCT2 milestones - *http://updates.yakindu.com/juno/milestones*
* Check all to install the YAKINDU SCT2
* Click **Next** to start the installation and click **Next** at the next step
* Accept the license agreement and click **Finish**
* The software will be installed

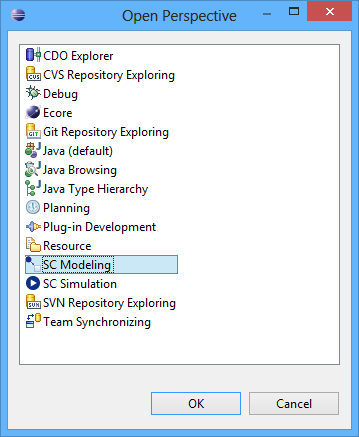
When the installation finished the wizard will ask to reopen Eclipse. The restart is important to make the newly installed software work correctly.

## First steps: my first state machine

In the following, you will create your first state machine with the YAKINDU Statechart tools and simulate it.

### Creating a New Project

For modeling purposes, the YAKINDU statechart tools offer a workbench perspective. Such a perspective is a bunch of editors and views that are organized in a pre-defined order on the screen. Open the Yakindu **SC Modeling** perspective by clicking **Window** > **Open Perspective** > **SC Modeling**. This perspective is optimized for statechart modeling. It consists of

* *Project Explorer* on the left
* *Outline* View on the right
* YAKINDU Statechart Editor at the top
* *Problems* and *Properties View* at the bottom
* In the Eclipse workbench, all elements are organized in projects. So first, to do is to create a project. Therefore click **File** > **New..." > \*Project**. In the wizard click **Next** and insert a project name. Click **Finish**.
* Now you see your project in the project explorer.

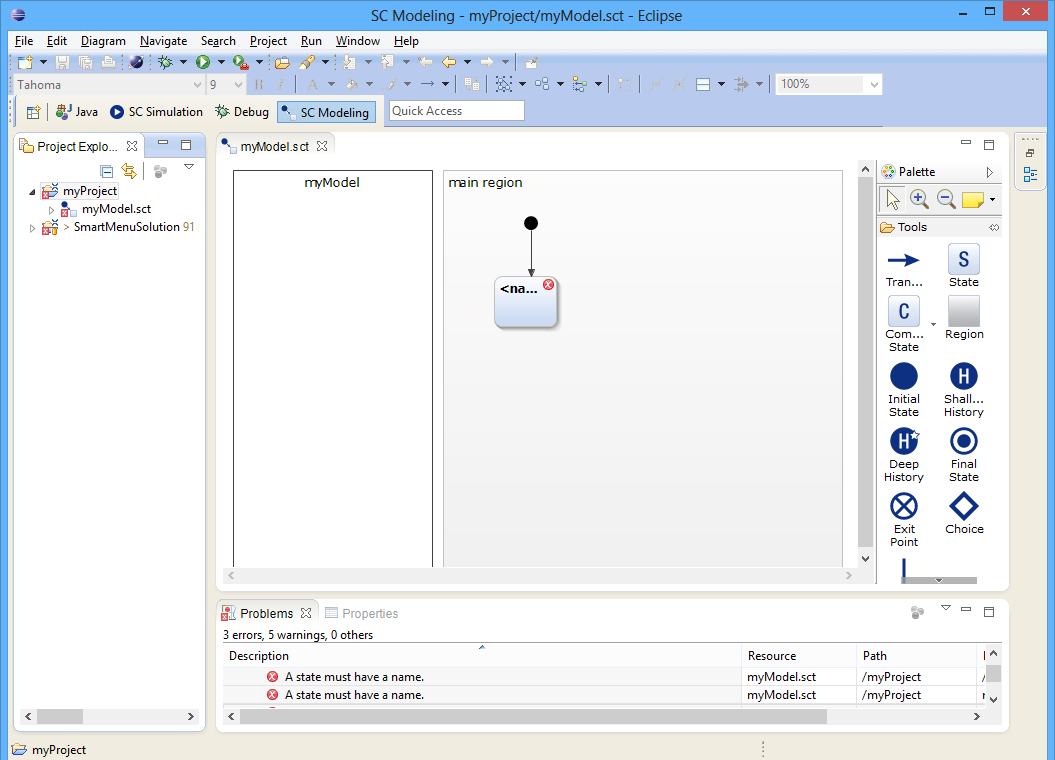
### Defining a State Machine

In the new project now, create a new statechart model:

1. Click **File** > **New** > **Other...** > **YAKINDU** > **YAKINDU Statechart Model**
2. Click **Next** and name the sct file
3. Click **Finish**
4. The YAKINDU statechart editor opens on the statechart model. It already has an initial state and an unnamed simple state connected by a transition.

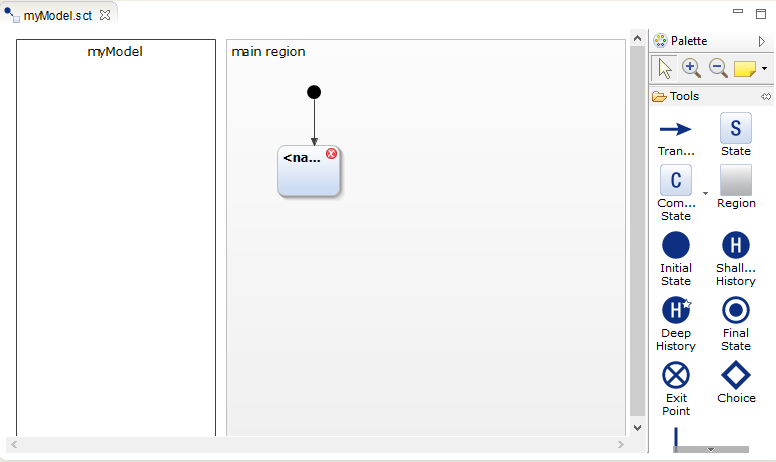
The newly created model has a problem. The new state has a red dot with a cross. This is an error marker. If you look at the problems view (the **Problems** tab), you see more details to that problem. In that case, it says: “A state must have a name”.   
To solve the problem:

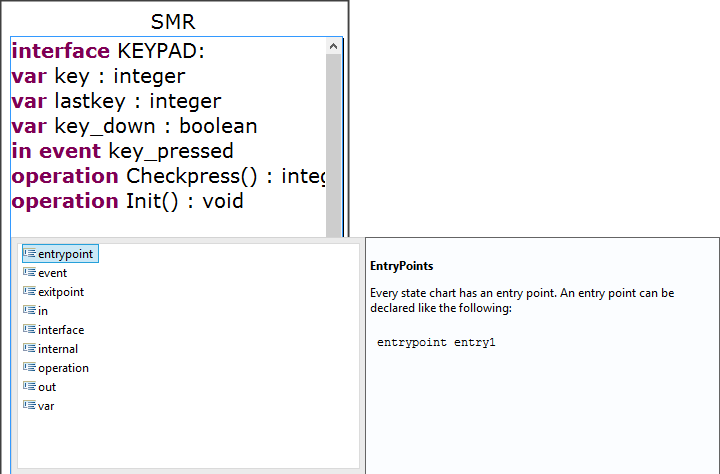
1. Click in the name field of the state and type the name ‚off’.
2. Click **File** > **Save**.

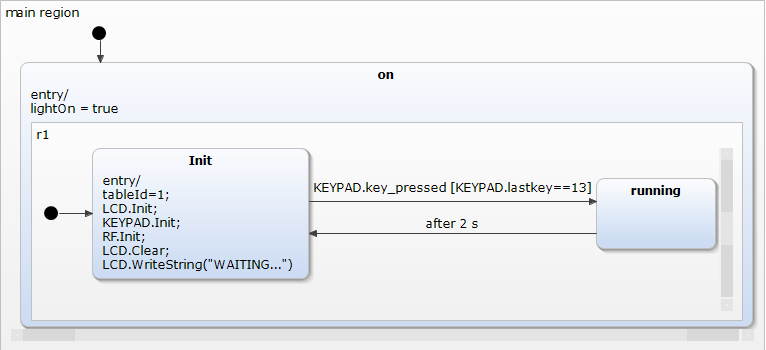
The error marker and the entry in the problems view vanish.

With the **YAKINDU Statechart editor,** you create or modify state models. The editor automatically opens on .sct files.

The editor consists of three parts:

* the graphic canvas to design the state machine
* the palette on the right that owns the elements to design the state machine
* ****a textual modeling field on the left

YAKINDU statecharts are self-contained – they not only contain the definition of states and state transitions, but also the definition of the statechart interfaces. To define those interfaces, open the direct edit mode by double clicking onto the statechart definition block on the left. The editor also owns comfortable functionality like syntax highlighting, code completion, live validation. The following image shows an example of code completion in the text fields of the editor:

1. Add an initial state
   1. Click on the **Initial State** in the palette
   2. Draw an Initial State in the main region
2. Add a composite state
   1. Click on the symbol **Composite** **State** in the palette
   2. Draw a composite state icon in the main region
   3. Name the composite state 'On'
3. Add an **entry trigger** that marks actions that are carried out on entering a state or state machine.
   1. Click on the composite state icon in the main region
   2. Type *entry/* into the Properties View at the bottom
4. Add a **variable**
   1. In the declarations view, add the statement internal: *var lightOn : boolean*
   2. To the state ‘On’ add the text *lightOn = true* below the entry trigger
5. Add more states
   1. Click on the symbol **State** in the palette
   2. Draw state icons in the main region
   3. Name the states as 'Init' and ‘running’
6. Draw a transition with an **Event** and a **ReactionTrigger** from ‘Init’ to ‘running’ state.
   1. Click on the symbol Transition
   2. Draw a line from *Init* to *running* state.
   3. In the declarations view, add the statement internal: *in event* *KEYPAD.key\_pressed*
   4. To the transition add the text *KEYPAD.key\_pressed [KEYPAD.lastkey==13]*
7. Draw a transition from ‘Init’ state to ‘running’ state.
   1. Click on the symbol **Transition** on the palette.
   2. Draw a line from ‘running’ to ‘Init’.
   3. Add the following statement to the transition: *after 1s.*

### Simulating the State Machine

• Model interpreter allows interactive simulation that helpful for finding errors.

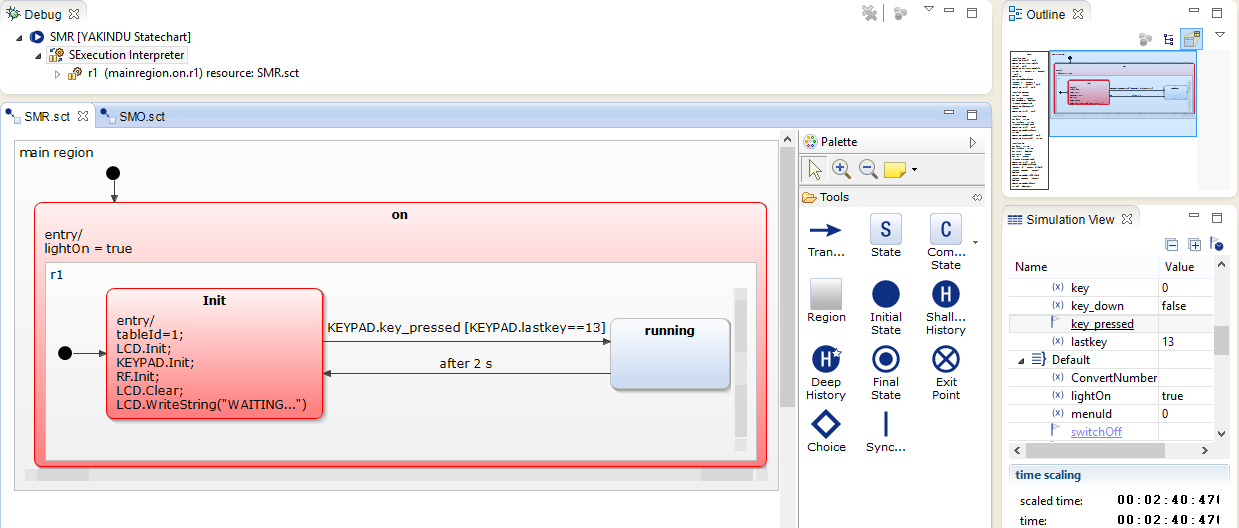
• API is for simulation engines, interacts with devices, and visualizing device state. Currently, it does not have real debug capability.

YAKINDU SCT 2 comes with a perspective to simulate the models. To simulate a state machine open the Yakindu **SC Simulation** perspective by clicking **Window** > **Open Perspective** > **SC Simulation**. This perspective is optimized for simulation purposes and consists of:

* *Project Explorer* on the left
* *Outline* view on the right top
* Simulation View on the right bottom
* Debug view at the top
* YAKINDU Statechart Editor

The simulation view is an interactive view to watch and control the state machine simulation. To get it open the simulation perspective.

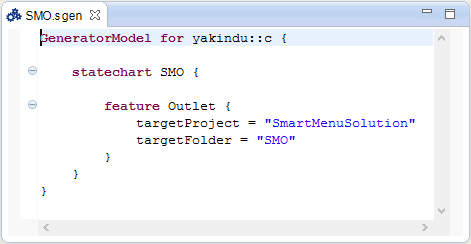
1. Start the simulation by clicking **Run** > **Run**.
2. The simulation starts and state ‘Init’ gets active (red).
3. Raise event‚ *key\_pressed*: Click on‚ *key\_pressed* in the simulation view and enter a value 13 for **variable** lastkey
4. State ‘running’ is active for 2 seconds, and then state ‘Init’ is active again.
5. Stop the simulation: **Run** > **Terminate**.

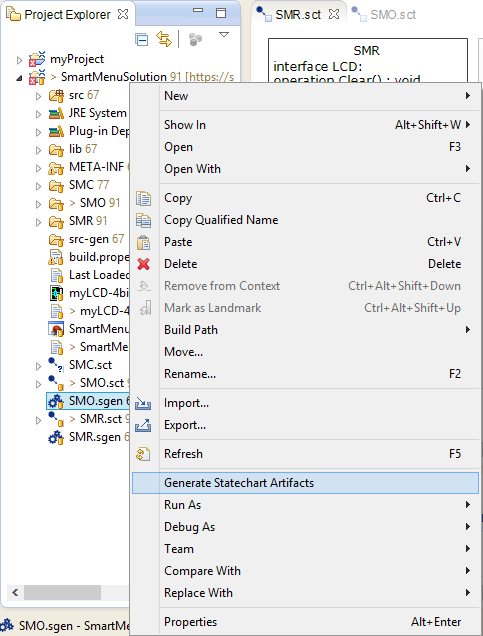
****

During the simulation in the statechart editor, the active state gets a red color. The previous transition is green. The simulation view shows the events that trigger the states. You can change values here to trigger events. Beneath the table, there are two clocks. One for the virtual time and one for the real time and a slider to change virtual time.

### Generator model for C

All generators can be customized with a generator model. This is a textual model file where generator features, like i.e. the outlet path, can be specified. To get started with the generator model, we included a new Eclipse wizard that creates a basic configuration file with default values.

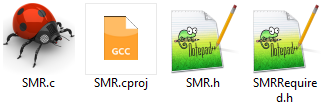
With the feature **Outlet** you define the folder the source files will be generated in. The following screenshot shows an example configuration for the C code generator. 

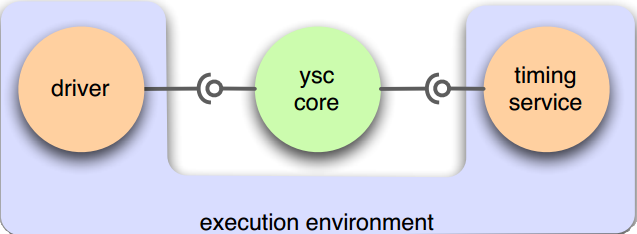
To generate code, right click on the .sgen configuration file then select Generate Statechart Artifacts.

### C-Code Generator and Drivers

In this part, we will see 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.

Now, we will use our generated code to compile to assembly code for the Smart Menu development board. As mentioned before, the code for the state machine can be found in the folder SMO. After running the workflow, the following files should be available in this directory:

The files completely define the state machine.

Then we will integrate the generated statechart implementation with some client code. Here is the example functions for a timing service, which the generated code will integrate with.

void TimerInit(){//initial function

TCCR0 = 0b011;

TCNT0=131;

TIMSK=(1<<TOIE0);

tmrCount=0;

sei();

}

void TimerSet(const sc\_eventid evenId,const sc\_integer max) {//function sets a Timer for an event

bool check=false;

for (int i=0;i<tmrCount;i++) {

if (tmrEvent[i].EventId==evenId) {

check=true;

tmrEvent[i].count=0;

tmrEvent[i].max=max;

tmrEvent[i].enabled=true;

}

}

if (!check) {

tmrEvent[tmrCount].EventId=evenId;

tmrEvent[tmrCount].count=0;

tmrEvent[tmrCount].max=max;

tmrEvent[tmrCount].enabled=true;

tmrCount++;

}

}

void TimerUnSet(const sc\_eventid evenId) {//function unsets a Timer for an event

for (int i=0;i<tmrCount;i++) {

if (tmrEvent[i].EventId==evenId) {

tmrEvent[i].count=0;

tmrEvent[i].enabled=false;

\*(sc\_boolean\*)evenId=bool\_false;

return;

}

}

}

We also need driver for devices that are being used. This is an example of a keypad driver:

#define KEYPAD\_DDR DDRF

#define KEYPAD\_PORT PORTF

#define KEYPAD\_PIN PINF

uint8\_t scan\_code[4]={0x0E,0x0D,0x0B,0x07};

uint8\_t ascii\_code[4][4]={7,8,9,13,

4,5,6,14,

1,2,3,15,

11,10,12,16};

uint8\_t KEYPAD\_Check() {

uint8\_t i,j,keyin;

for (i=0;i<4;i++) {

KEYPAD\_PORT=0xFF-(1<<(i+4));

\_delay\_us(10);

keyin=KEYPAD\_PIN & 0x0F;

if (keyin!=0x0F)

for (j=0;j<4;j++)

if (keyin==scan\_code[j]) return ascii\_code[j][i];

}

return 0;

}

void KEYPAD\_Init()

{

//definition for connections

KEYPAD\_DDR=0xF0;

KEYPAD\_PORT=0x0F;

}

There are functions in the generated code we need to overwrite. These functions are operations that we define in the state chart editor. Here is an example for overwriting function sMRIfaceRF\_sendData:

sc\_boolean sMRIfaceRF\_sendData(const sc\_integer cmd, const sc\_integer id, const sc\_integer dish\_id, const sc\_integer amount) {

unsigned char num;

unsigned char mod;

//clear buffer

for(uint8\_t i=0; i<sizeof(bufferout); i++) bufferout[i] = 0;

//Commands: 1-ordering, 2-delete, 3-help, 4-bill

bufferout[0]=cmd+'0'; //convert cmd from int to char

//change table id or kitchen

num=id;

for (int i=1;i>=0;i--) {

mod=num % 10;

num/=10;

bufferout[i+1]=mod+'0';

}

//convert dish code

num=dish\_id;

for (int i=2;i>=0;i--) {

mod=num % 10;

num/=10;

bufferout[3+i]=mod+'0';

}

//convert amount

num=amount;

for (int i=1;i>=0;i--) {

mod=num % 10;

num/=10;

bufferout[6+i]=mod+'0';

}

//Set Address for Data

nrf24l01\_settxaddr(nrf24l01\_addrtx);

uint8\_t writeret = nrf24l01\_write(bufferout);

\_delay\_ms(1);

if(writeret == 1) {

return true;

} else {

return false;

}

}

# Yakindu Statechart tool concepts

## Modeling

Yakindu Statechart Models are based on statecharts as defined by David Harel and are close to UML state machines. Thus, they support all structural model elements as defined by the UML specification, which are States (orthogonal and hierarchical), Regions, Transitions and Pseudo States (History, Deep History, Initial, Final, Choice, Join / Fork).   
In addition, Yakindu Statechart Models specify interfaces that define the interaction of the state machine with its environment. Besides some more advanced concepts like Entry / Exit Points, an interface consists of in and out Events as well as Variables including types. These well-defined statechart interfaces are especially useful in the context of component models and product line engineering.

For defining interfaces and modeling, the dynamic aspects of statecharts (triggers, guards and actions) SCT provides a statically typed, textual action language. It tightly integrates into the graphical editor and supports the user with code completion, syntax highlighting, cross-referencing and validation during modeling.

Declarations of interfaces, events, variables etc. are done in a textual modeling field in the editor. The language expressions that define actions are directly added to the elements like states or transitions.

## Simulation

Yakindu Statechart Models can be executed via an integrated simulation engine. The simulation engine supports two different types of execution semantics:

* cycle based
* event driven

The default execution semantic is **cycle based** which executes a statechart cycle within a fixed period and thus allows processing of concurrent events. In contrast, the **event based** approach executes a statechart cycle as soon as an event occurs.

During simulation, the currently active states and the transition path including the previous states are highlighted in the editor for visual debugging. An additional view shows the variable values and allows raising events.

Apart from that, the engine uses a **virtual time** during simulation. The user can provide a time scaling factor that is multiplied with the real time. This is especially useful during debugging, if the statechart model contains very tight or long running time triggers.

## Code Generation

Yakindu Statechart Tools currently support the generation of Java, C and C++ Code. All generators can be customized with a **generator model**. This is a textual model file where generator features, like the execution type (event or cycle based), or the interface styles (static or generic), can be specified. The code generation process can be executed either with a builder that starts code generation on resource change or manually with a context menu action.

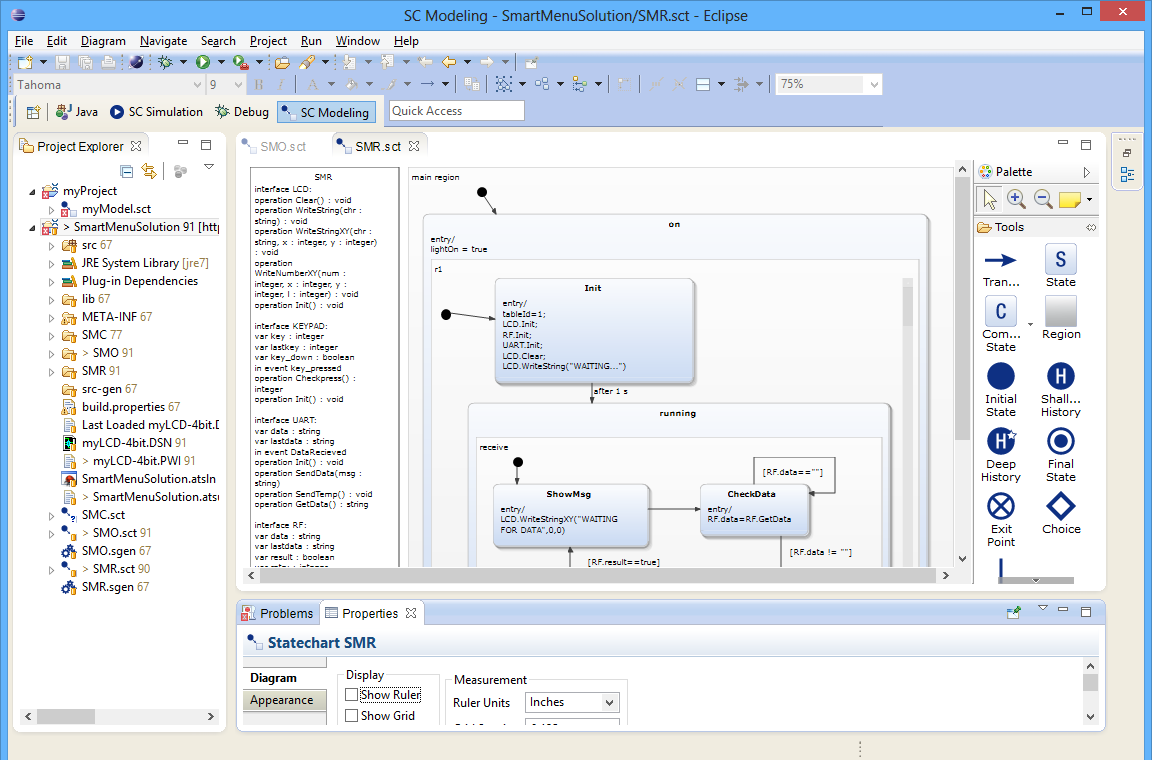
## Domain specific adaptability

Yakindu Statechart Tools were developed with a domain specific adaptability in mind. Several extension points allow the customization of all mentioned aspects. It is possible to contribute a custom type system or custom action languages, which may be a dialect of the textual description language or a complete new language.

In addition, the different code generators can be customized or new code generators for other target languages can be plugged in easily.

## Yakindu SC Modeling perspective

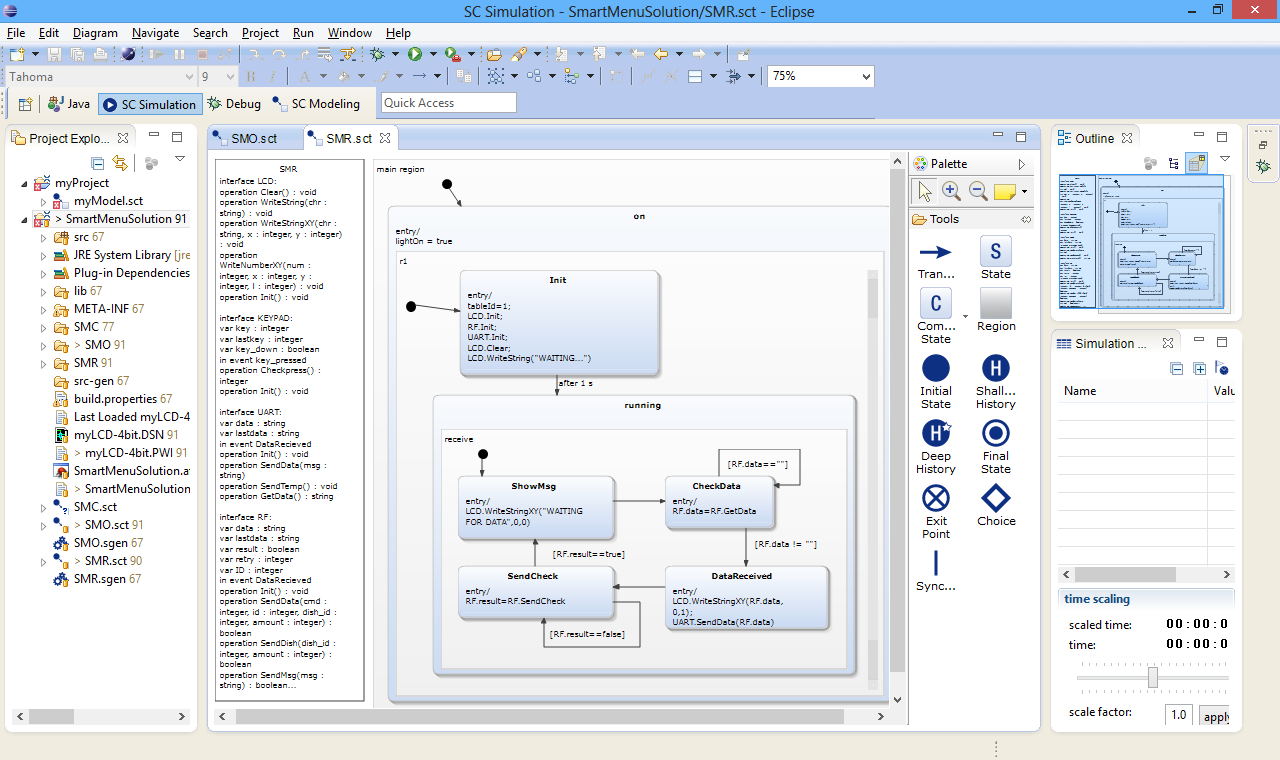
For state machine modeling purposes open the Yakindu **SC Modeling** perspective by clicking **Window** > **Open Perspective** > **SC Modeling**. This perspective is optimized for statechart modeling. It consists of

* *Project Explorer* on the left
* *Outline* View on the right
* YAKINDU Statechart Editor at the top
* *Problems* and *Properties View* at the bottom

## Yakindu SC Simulation perspective

To simulate a state machine use the Yakindu **SC Simulation** perspective by clicking **Window** > **Open Perspective** > **SC Simulation**. This perspective is optimized for simulation purposes and consists of:

* *Project Explorer* on the left
* *Outline* view on the right top
* Simulation View on the right bottom
* Debug view at the top
* YAKINDU Statechart Editor at the bottom



## YAKINDU Statechart Editor

With the YAKINDU Statechart editor, you create or modify state models. The editor automatically opens on .sct files. To create a new YAKINDU statechart model click **File** > **New** > **Other...** > **YAKINDU Statechart model**. Give it a speaking name and click **Finish**. The newly created model opens in the statechart editor. In addition, the SC Modeling perspective opens.

The editor consists of three parts:

* the graphic canvas to design the state machine

The palette on the right that owns the elements to design the state machine

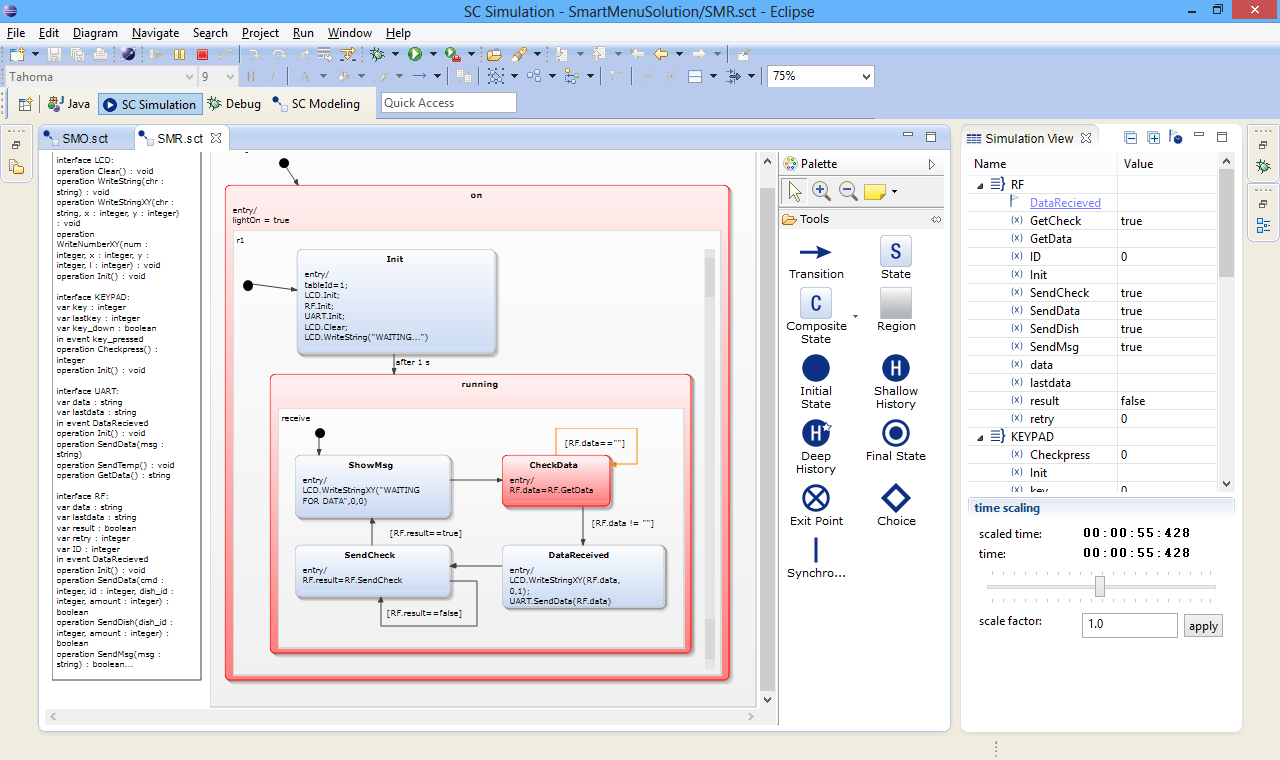
* a textual modeling field on the left

You can use the textual modeling field to define internal behavior of states and declare events or variables. The *reference* explains this language.

Elements that cause problems get warning and error markers attached. For more details about the problems, open the **Problems View**.

## Simulation View

The simulation view is an interactive view to watch and control the state machine simulation. To get it open the simulation perspective.

Start a simulation by clicking **Run** > **Run**.

During the simulation in the statechart editor, the active state gets a red color. The previous transition is green. The simulation view shows the events that trigger the states. You can change values here to trigger events. Beneath the table, there are two clocks. One for the virtual time and one for the real time and a slider to change virtual time.

You can also select an event driven or cycle based run configuration.

## YAKINDU SCT Nature

YAKINDU model projects are associated with the **YAKINDU SCT nature**. On projects with **YAKINDU SCT nature** .sct and .sgen files are automatically build and checked for problems. Therefore, you get fast feedback during editing. You can toggle the SCT nature by right clicking **Configure** > **Add** or **Remove YAKINDU SCT Nature** on the project.