**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?

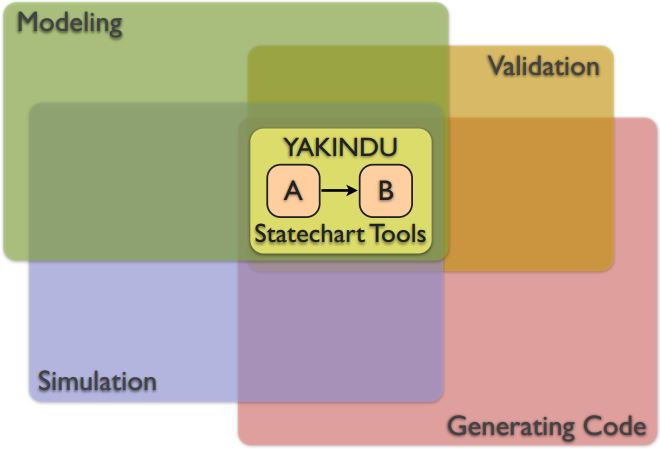
The main part of YAKINDU SCT 2 is an open source project ([www.yakindu.org](http://www.yakindu.org/) ). Most of the developers work for itemis; a well-known consulting company specialized on model-based development.

## 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

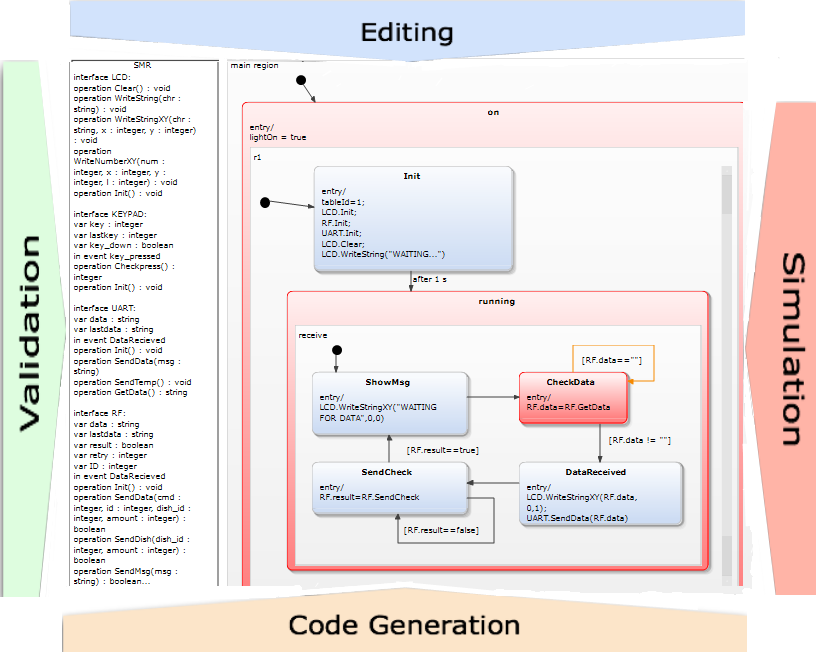
YAKINDU Statechart Tools (SCT) provides an integrated modeling environment for the specification and development of reactive, event-driven systems based on the concept of statecharts. It is an easy to use tool that features sophisticated graphical state chart editing, **validation and simulation of statecharts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

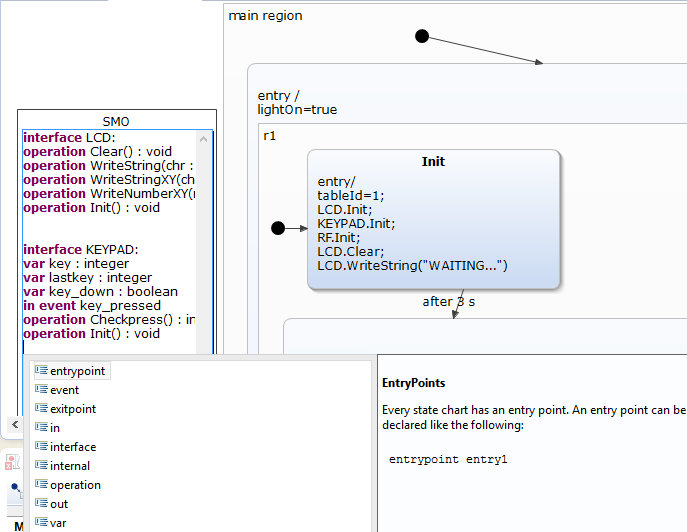
The following graph shows these features and their relation to each other:

## **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**

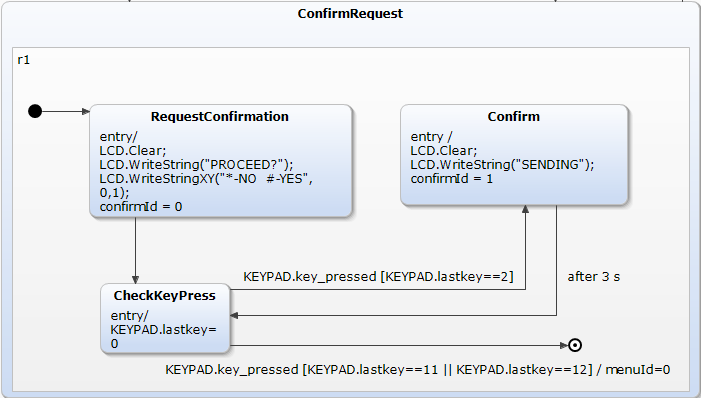
Even though YAKINDU SCT is ready to use out of the box, it is open and designed for extensibility, so it can serve as a basis for more specific statechart modeling and implementation solutions. Nearly every aspect of the tool is customizable. Typical cases of customization are platform specific extensions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

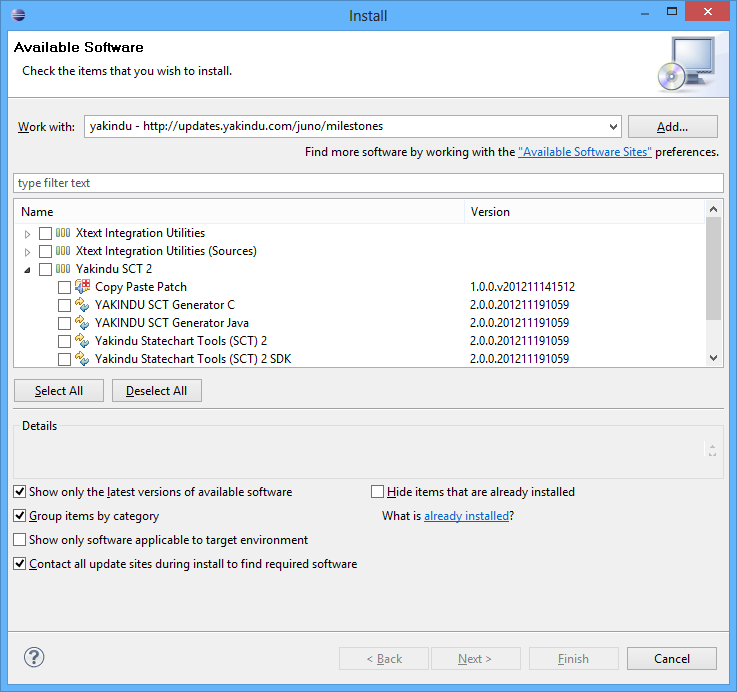
The **Yakindu Statechart Tools 2** are built upon Java and Xtext. Therefore, you need to have installed a *Java Runtime Environment* and Xtext installed. The easier way to get this is to install the *pre-configured Xtext contribution*.

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

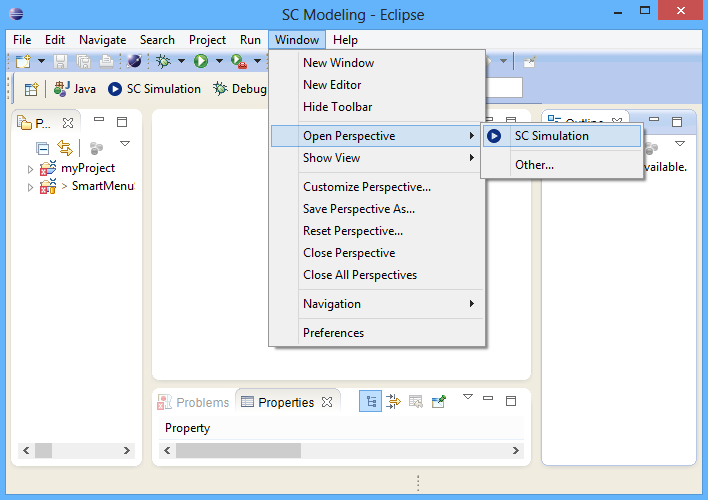
When the installation finished the wizard will ask to reopen Eclipse. The restart is important to make the newly installed software work correctly.

## 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.

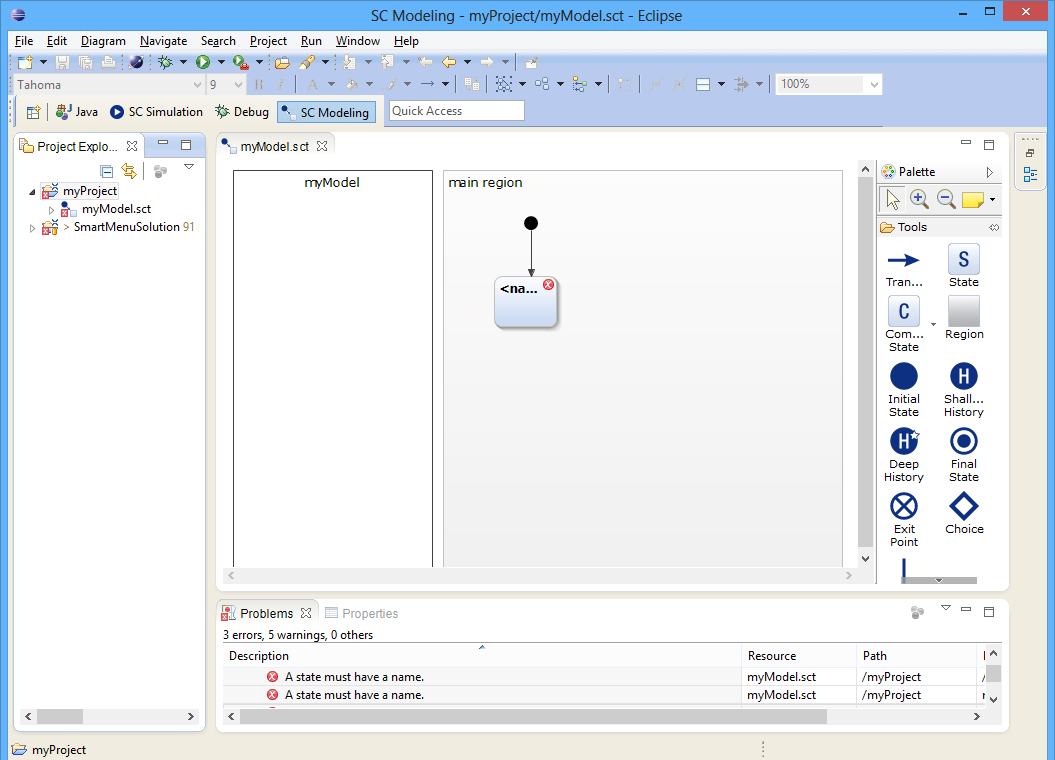
### Defining a State Machine

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.

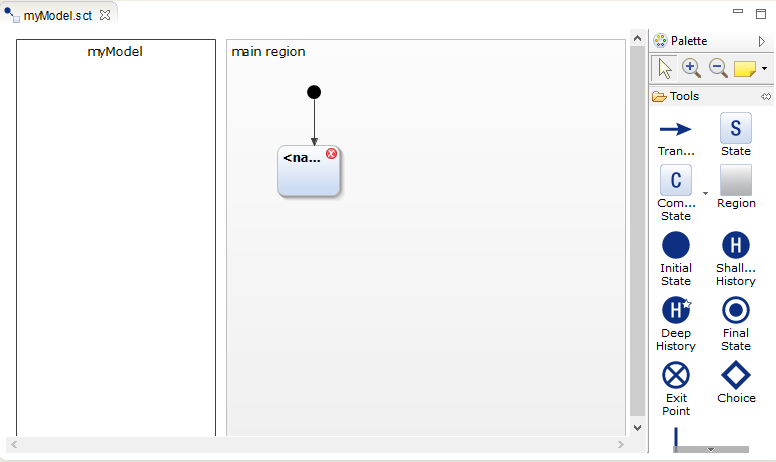
The newly created model has a problem. The new state has a red dot with a cross. This is an error marker. If you look at the problems view (the **Problems** tab), you see more details to that problem. In that case, it says: “A state must have a name”.   
To solve the problem:

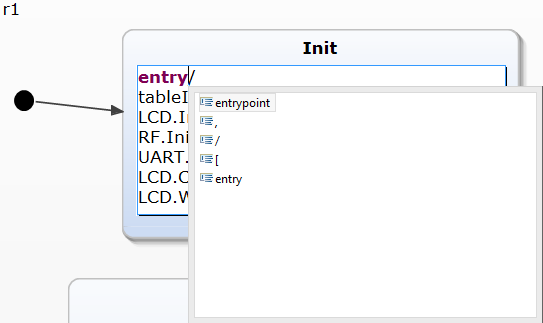
1. Click in the name field of the state and type the name ‚off’.
2. Click **File** > **Save**.

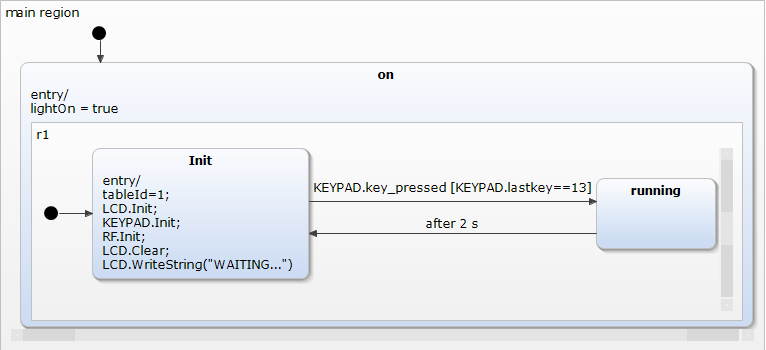
The error marker and the entry in the problems view vanish.

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

You can use the textual modeling field for declaration internal behavior of states and events or variables. 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** (refer to YAKINDU SCT 2 Reference for more information)
   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. (refer to YAKINDU SCT 2 Reference for more information)
   1. Click on the symbol Transition
   2. Draw a line from off to one 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.

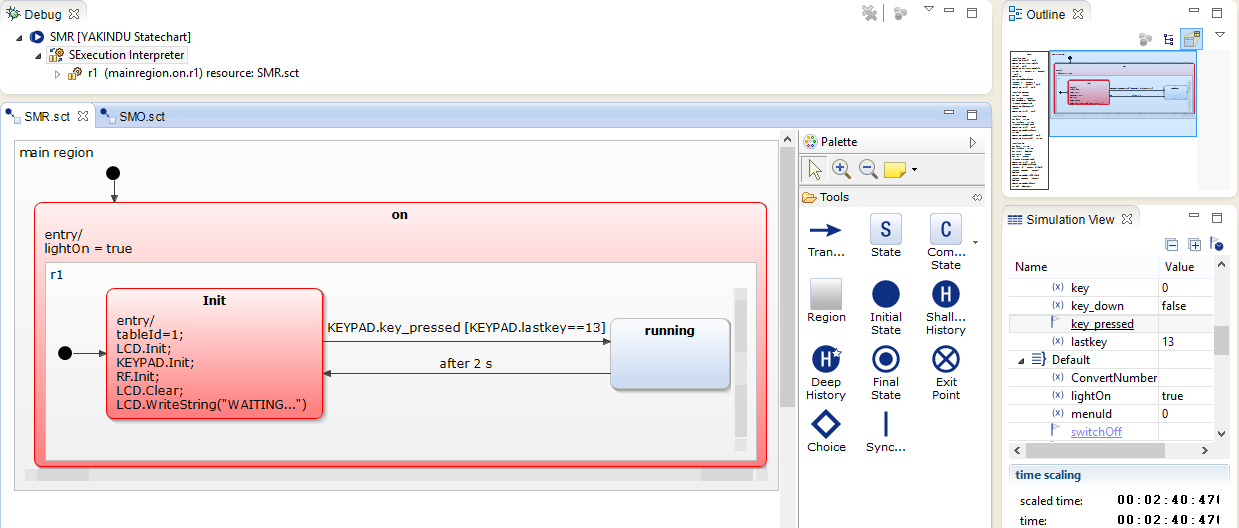
• API is for simulation engines, interacts with devices, and visualizing device state. Currently, it does not have real debug capability.

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, 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.

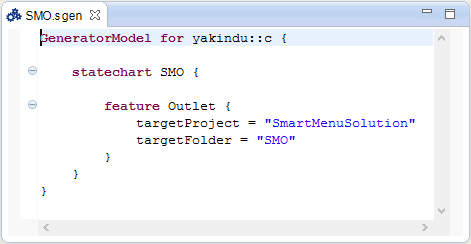
With the feature **Outlet** you define the folder the source files will be generated in:

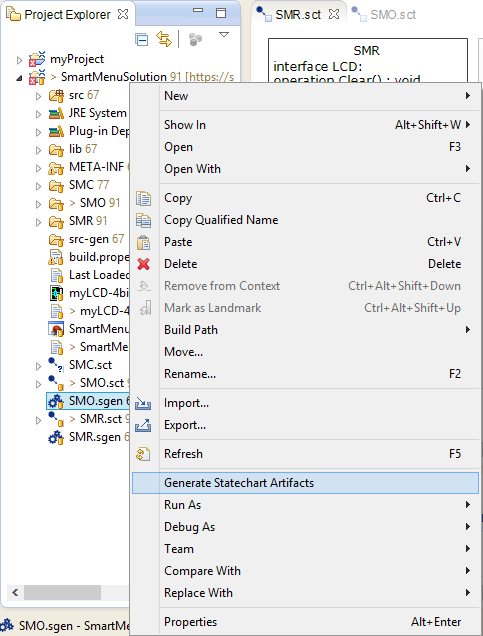
feature Outlet {

targetProject = "SmartMenuSolution"

targetFolder = "SMO"

}

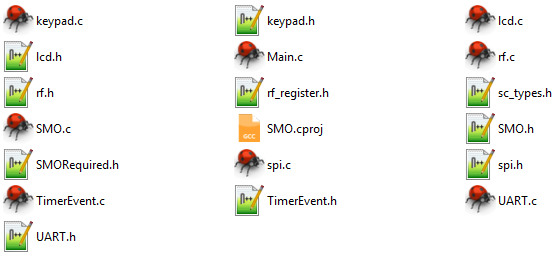
The following screenshot shows an example configuration for the C code generator. 

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.

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

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.

# 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.

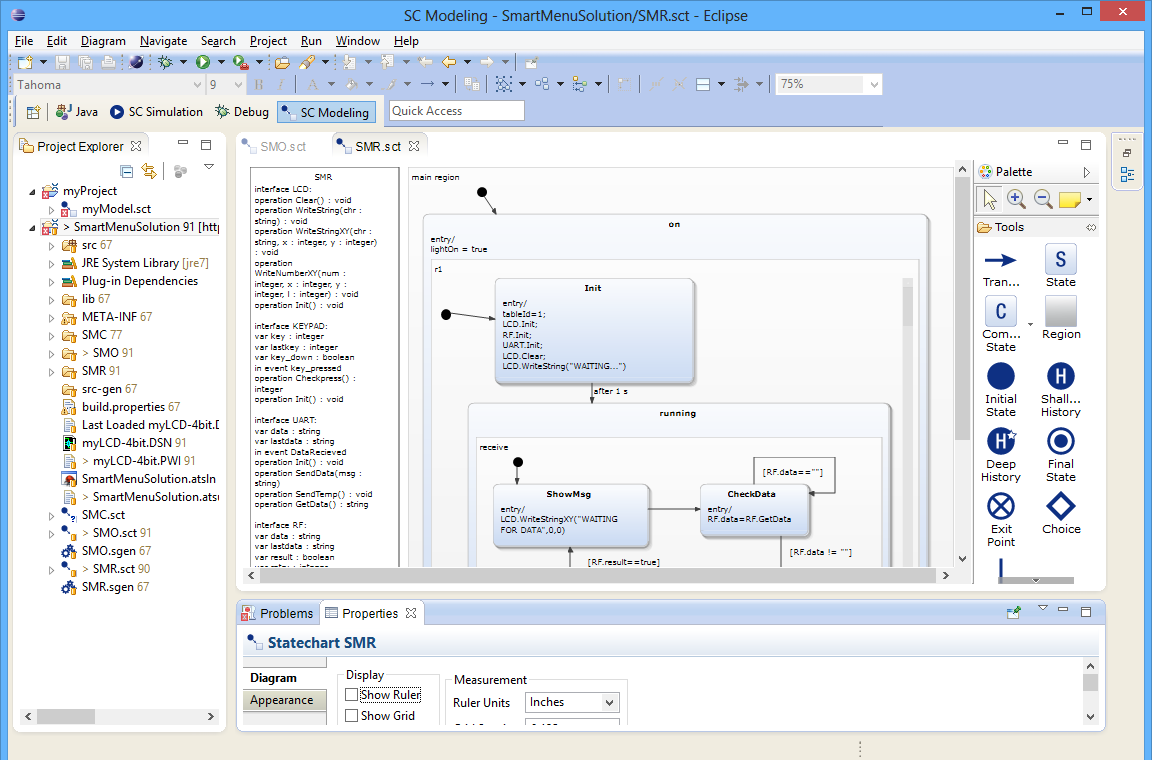
## Domain specific adaptability

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

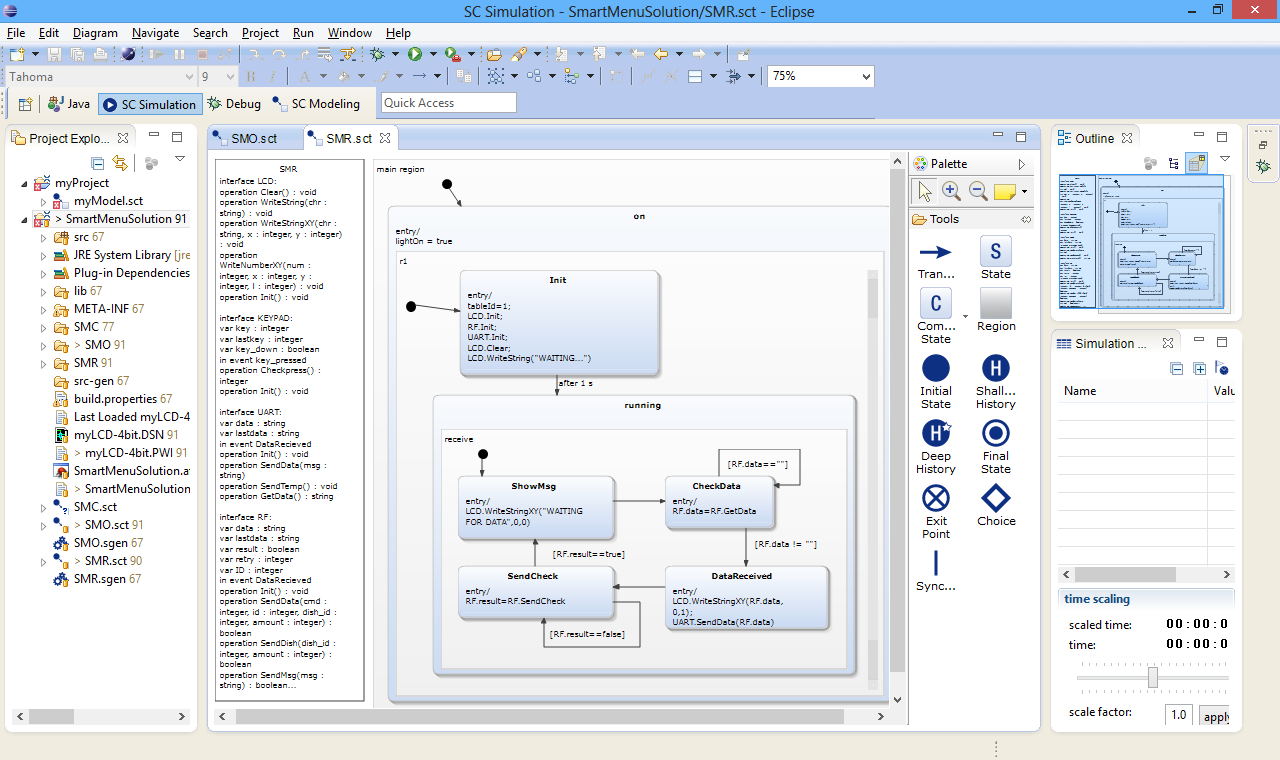
For state machine modeling purposes open the Yakindu **SC Modeling** perspective by clicking **Window** > **Open Perspective** > **SC Modeling**. This perspective is optimized for statechart modeling. It consists of

* *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

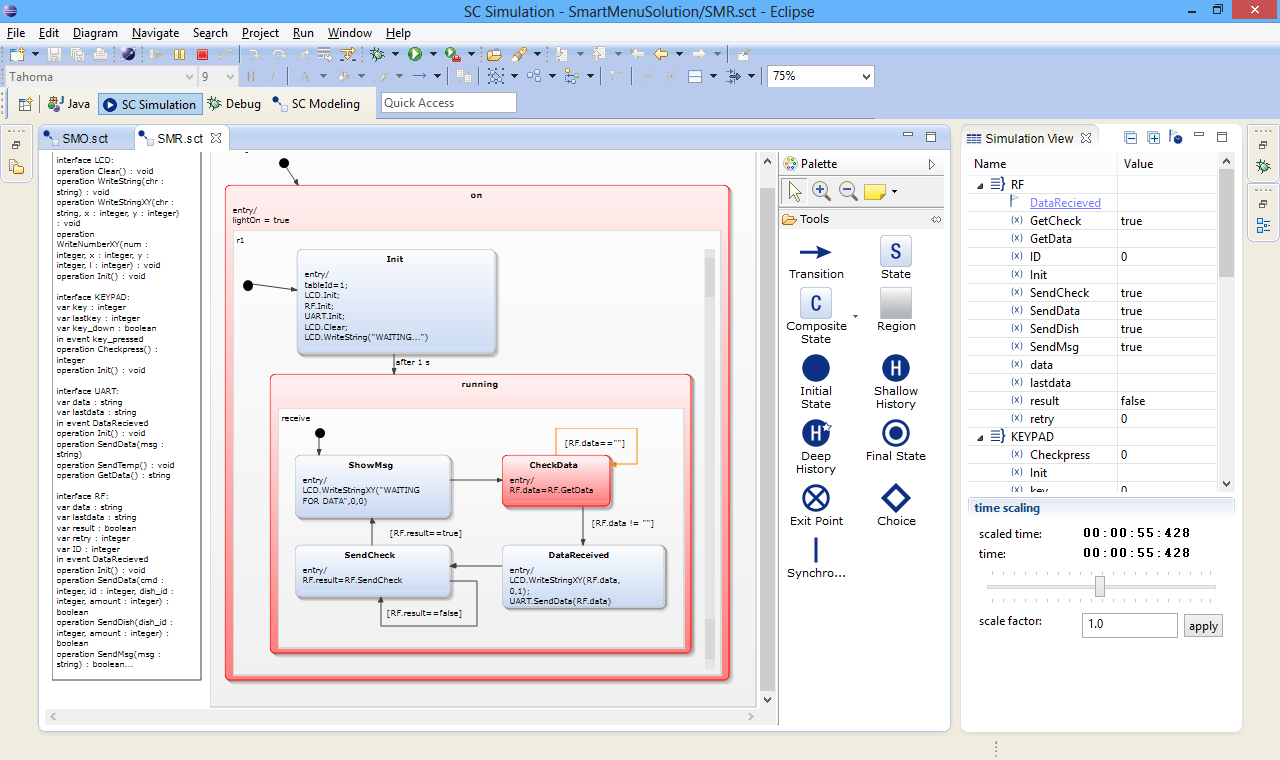
* a textual modeling field on the left

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.

# YAKINDU SCT 2 Reference

## State chart elements

In the following, the state chart elements of the YAKINDU SCT 2 editor are described. The meta model of the YAKINDU SCT 2 is the model of finite state machines. It is based on the view of a system that is defined by a finite number of states. The behavior of that system is based on the active states. These states are determined by the history of the state machine. Very important are the theoretical models for state machines by Mealy and Moore. Mealy state machines associate actions with transitions. Moore machines associate actions with states (entry, exit). In the YAKINDU SCT 2 both is possible.

The YAKINDU SCT 2 Meta model is designed similar to the UML state chart Meta model with the following differences

* they are self-contained with interfaces defined by events and variables
* core execution semantics are cycle driven, not event driven

this allows to process concurrent events

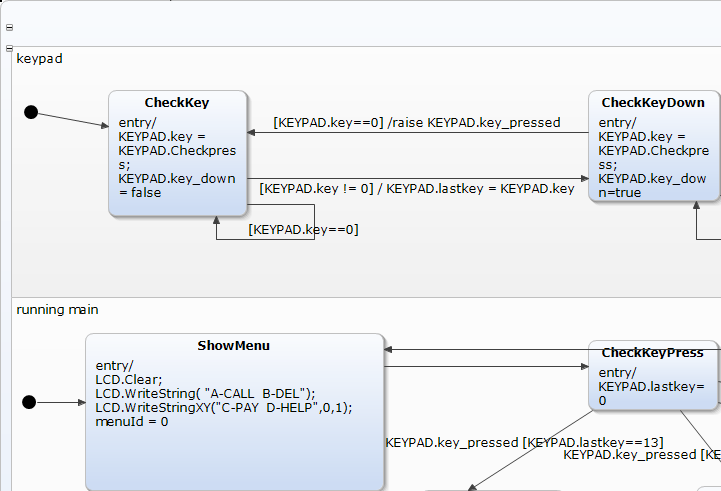
* + event driven behavior can be defined on top
* time is an abstract concept for state charts
* time control is delegated to the environment

The model interpreter and different flavors of generated code follow these same core semantics.

Please refer to the description of the *UML Statecharts* for more details.

## Regions

As already mentioned the YAKINDU state charts are self-contained. They are organized in regions. Due to this, it is possible to organize multiple state machines in different regions and to run them concurrently.



## States

States are the central elements of a state machine. A state has to be placed inside a region and needs a unique name inside this region. During simulation, each state can be active or passive. An active state has actions that are accomplished. Either an action is carried out on entering a state, during active state or on exit.

## Transitions

A transition is the transfer of one state to another. Transitions are diagrammed as arrows and can carry events and actions but must not.

A textual description language (#Statechartdescriptionlanguage) defines the syntax of events and actions. Please refer to the documentation section *Events* for more details. For more details on *Actions*, refer to the chapter *Actions*.

If a state has more than one outgoing transition without event that transition is carried out first that was modeled first.

## Initial state and final state

Initial and final states are pseudo states, because the state chart does not rest on them. Pseudo states express characteristics that are impossible to express by simple states.

The initial state is always the first state that is active during interpretation or simulation of the state machine. An initial state can only have one outgoing transition and no incoming. This transition has no events or actions.

Inside a region, only one initial state is allowed, but every region can have an initial state.

## Choice

Choice is also a pseudo state. It can be used to model a conditional path. Choice nodes divide a transition into multiple parts.

Usually the first transition points towards the choice node. One of the choice outgoing transitions can carry a condition.

## Junction

A junction is a pseudo state do combine transitions. This is very comfortable if a state machine has many similar transitions. Junctions add clear arrangement to the state machine.

## Composite State

A composite state is a state that is composed of other state machines. These are also organized in regions. Besides the simple composite state YAKINDU knows two kinds of composite states: orthogonal state and submachine states.

Composite states contain other state machine branches.

## Statechart description language

The textual description language is used to declare and describe behaviors in the state machine. It is case sensitive.

### Type system

The language has an integrated small type system with the following simple types:

* integer
* real
* boolean
* string
* void

So events and variables can be declared with types:

var intVar : integer

var realVar : real

var boolVar : boolean

var stringVar : string

var voidVar : void

event addInt : integer

event checkValidity : boolean

**For example:**

**interface** RF:

**var** data : string

**var** lastdata : string

**var** result : boolean

**var** retry : integer

**var** ID : integer

**in** **event** DataRecieved

### Expressions

Expressions can be defined similar to other programming languages. The language offers operators to define logical expressions, bitwise arithmetic, and arithmetic expressions and bit shifting.

Logical expressions are similar to other programming languages. The return type is **boolean**. In the following, there are some examples of these:

|  |  |
| --- | --- |
| Logical AND | var1 && var2 |
| Logical OR | var1 || var2 |
| Logical NOT | !var1 |
| Conditional expression | var1 ? var2 : 23 |
| Bitwise XOR | var1 ^ var2 |
| Bitwise OR | var1 | var2 |
| Bitwise AND | var1 & var2 |
| Logical Relations and Shift Operators | |  |  | | --- | --- | | less than | < | | equal or less than | <= | | greater than | > | | equal or greater than | >= | | equal | == | | not equal | != | | shift left | << | | shift right | >> | |
| Binary arithmetic operators | |  |  | | --- | --- | | plus | + | | minus | - | | multiply | \* | | divide | / | | modulo | % | |
| Unary arithmetic operators | |  |  | | --- | --- | | positive | + | | negative | - | | circa | ~ | |

### Statements

A statement can either be an assignment, raising an event or call an operation. The language has the following assignment operators:

* simple assignment: =
* multiply and assign: \*=
* divide and assign: /=
* calculate modulo and assign: %=
* add and assign: +=
* subtract and assign: -=
* bitshift left and assign: <<=
* bitshift right and assign: >>=
* bitwise AND and assign: &=
* bitwise XOR and assign: ^=
* bitwise OR and assign: |=

An event is raised by the keyword raise followed by the event name and if it is an interface event the name of the interface.

An operation is called similar to other programming languages with the operation name and passing concrete parameters. The parameters can be expressions.

### Scopes

**Namespace**

The language allows defining unique namespaces, which can be used to qualify references to the statechart.

namespace trafficlights

**Interface scope**

Declarations in the interface scope are externally visible. They can be shared within the environment.

interface NamedInterface:

in event event1

out event event3

var variable1 : real

entrypoint entry1

exitpoint exit1

**Internal scope**

Declarations made in an internal scope are only visible for contained states.

internal:

var localVariable1: integer

event localEvent: integer

local event localEvent2: NamedInterface.event1 || localEvent

local event localEvent3: localEvent || localEvent2 : 25

operation localOperation (integer, integer): integer

localEvent3 / raise NamedInterface.event3 :

localOperation(valueOf(localEvent),NamedInterface.variable1);

### Declarations

Within scopes, there can be declarations of Events, Variables, Operations, LocalReactions, EntryPoints and ExitPoints.

### Events

Within interface scope, events have a direction. They can either be ingoing or outgoing:

interface NamedInterface:

in event event1

out event event2

Within local scope, events can carry variables:

internal:

event localEvent1 : integer

Local events can be derived from interface events or other local events and can have a value assignment:

internal:

event localEvent1: integer

local event localEvent2 = NamedInterface.event1 || localEvent1

local event localEvent3 = localEvent2 || 25

### Variables

Variables can have different visibilities. They can be visible for the environment:

var variable1: real

Variables can be **readonly** (constants):

var readonly pi: real = 3.1415

The environment can reference variables.

var external variable3: integer = 34

### Reaction Triggers

Actions are key constructs in state machines to model behavior. The YAKINDU SCT 2 knows the following kinds of actions.

**after**

The after trigger specifies one-shot time events.

After the specified time the reaction is triggered. An after trigger can be used in transitions of states as well in local reactions of states and statecharts. The specified time starts when the state or statechart is entered.

*after 20 s*

Structure:

*after time* *( unit* *)?*

The time unit can be:

* s – seconds
* ms – milliseconds
* us – microseconds
* ns – nanoseconds
* empty – implies seconds

**every**

The every trigger specifies periodic time events.

The reaction is triggered periodically after the specified time. An every trigger can be used in transitions of states as well in local reactions of states and statecharts. The specified time starts when the state or statechart is entered and repeats periodically.

*every 200 ms*

Structure:

*every time* *( unit* *)?*

The time unit can be:

* s – seconds
* ms – milliseconds
* us – microseconds
* ns – nanoseconds
* empty – implies seconds

**always**

This trigger is always true and enables a reaction to be executed in every run to completion step (RTS). It is equivalent to on cycle.

**default, else**

The default trigger is equivalent to the else trigger. It is intended for use for the outgoing transitions of choice pseudo states, to make sure that always an outgoing transition can be taken. It can only be be used in transitions and implies the lowest evaluation priority for that transition.

**entry**

An entry trigger marks actions that are carried out on entering a state or state machine.

**exit**

An exit trigger marks actions that are carried out on exiting a state or state machine.

**oncycle**

The oncycle trigger is always true and enables a reaction to be executed in every run to completion step (RTS). It is equivalent to always.

### Operations

Operations can have none, one or multiple parameters. The parameters are only declarated by their type. An operation can have one return type similar to Java.

operation localOperation (integer, integer):integer

localEvent3/ raise NamedInterface3.event1

### LocalReactions

Local reactions describe the internal behavior of a state. So they have internal scope. A local reaction is declared as follows:

LocalReaction: ReactionTrigger '/' ReactionEffect ('#' ReactionProperties)?

ReactionTrigger: (Event ("," Event )\* (=> '[' Expression ']')?) | '[' Expression ']'

ReactionEffect: Statement (';' Statement )\* (';')?

Statement: Assignment | EventRaising | OperationCall

ReactionProperties: (EntryPoint | ExitPoint)\*

Within a local reaction, an interface event can be raised:

internal:

localEvent1 / raise NamedInterface.event3 : localOperation (valueOf(localEvent), NamedInterface.variable1);

Local reactions can have priority values. These are defined by a following # and the integer number of priority:

localEvent2 / NamedInterface.variable2 += 3; #1

localEvent3 / NamedInterface.variable4 += 2.0; #2

### EntryPoints

Every state chart has an entry point. An entry point can be declared like the following:

entrypoint entry1

### ExitPoints

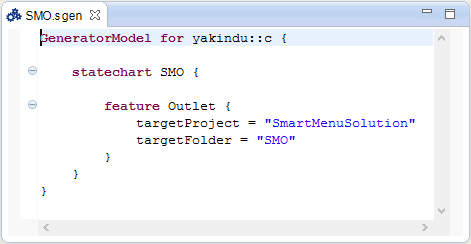
Every state chart has an exit point. This exit point can be declared like the following.

exitpoint exit1

### SGen

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.

The following screenshot shows an example configuration for the C code generator.

**Generator model for C**

* **Feature Outlet**

With the feature **Outlet** you define the folder the source files will be generated in:

feature Outlet {

targetProject = "org.terra.coffee.machine"

targetFolder = "src-gen"

}

* **Feature LicenseHeader**

With the feature License header, you can set a license text that is added to the headers of all generated files:

feature licenseHeader {

licenseText = "Copyright (c) 2012 itemis AG.

All rights reserved."

}

* **Feature Debug**

The feature debug controls the output of debug information. An important information source is the intermediate model sExec.

feature Debug {

dumpSexec = false

}

**Feature CCodeFeature**

feature CCodeFeature {

DebugType="DEBUG"

InterfaceEventListener=false

Singleton=true

}

**Feature FunctionInlining**