



# QUICK REFERENCE

This quick reference gives an overview of the building blocks a statechart consists of and their semantics. It also shows the textual syntax used to express behavior inside the graphical model. If you are new to **YAKINDU Statechart Tools**, this is a good starting point to learn the main concepts.



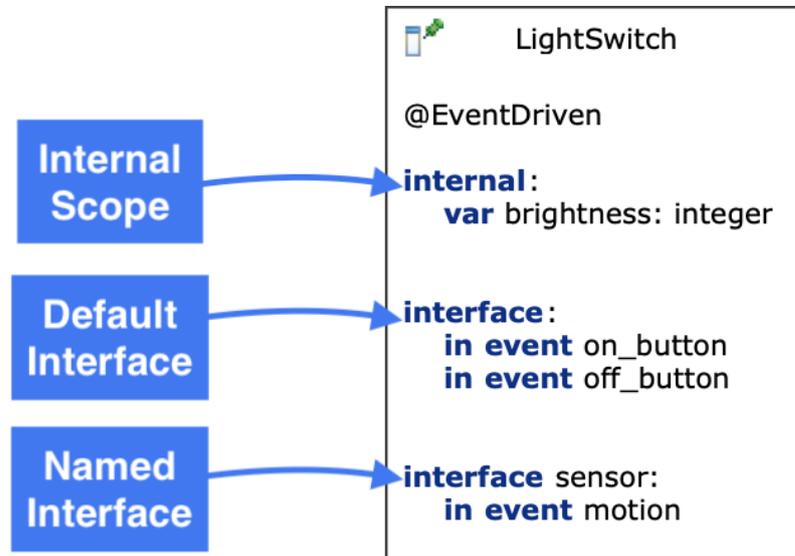
# STATECHART ELEMENTS OVERVIEW

A statechart consists of a number of different elements. The following list gives an overview of these elements in the order they are listed in the editor palette.

Statechart Element	Description
 <b>Transition</b> <a href="#">Section: States and transitions</a>	Transitions connect states with each other. Transition reactions define under which conditions a transition is taken.
 <b>State</b> <a href="#">Section: States and transitions</a>	A state is the most basic building block of a statechart. A state can define reactions for when it gets entered or left.
 <b>Composite State</b> <a href="#">Section: Composite State</a>	A composite state groups a number of substates. It can be used to express state hierarchies.
 <b>Orthogonal State</b> <a href="#">Section: Orthogonality</a>	An orthogonal state is used to express concurrency.
 <b>Region</b>	A region is a container for states and transitions. Regions can exist as top-level elements or inside of a composite or orthogonal state. Multiple regions that coexist on the same level express concurrency as in an orthogonal state.
 <b>Entry</b> <a href="#">Section: Composite State</a>	Entry points mark the initial state of a region. A region can have multiple named entry points to specify different execution flows.
 <b>Shallow History</b> <a href="#">Section: History Nodes</a>	A shallow history remembers the last active state inside a composite state.
 <b>Deep History</b> <a href="#">Section: History Nodes</a>	A deep history remembers all nested active state inside a composite state.
 <b>Final State</b> <a href="#">Chapter: Final State</a>	A final state denotes the end of the execution flow.
 <b>Exit Note</b> <a href="#">Section: Composite State</a>	Exit points are used to leave a composite state and are the counterparts of entry points.
 <b>Choice</b> <a href="#">Section: Choice</a>	A choice node is used to model a conditional path.
 <b>Synchronization</b> <a href="#">Section: Orthogonality</a>	Synchronization nodes are used to model forks and joins in combination with orthogonal states.

# STATECHART DEFINITION SECTION

The *definition section* of the statechart defines which entities of the statechart, like variables, events and operations, are accessible from the outside and which ones are only used internally. For this, a definition section can declare an internal scope and multiple interfaces.



# STATECHART INTERFACES

A statechart interface declares the entities that are externally visible. These are the elements by which the client code can interact with the statechart. A statechart can declare multiple interfaces with different names. The unnamed interface is also called the default interface.

Interface Element Declaration	Meaning
<code>in event SwitchOn</code>	Incoming event, supposed to be raised by the client code and processed by the state machine to evaluate potential state transitions.
<code>in event Slider : integer</code>	Incoming event with payload of type <code>integer</code>
<code>out event Finish</code>	Outgoing event, supposed to be raised by the state machine and delivered to the outside.

<code>out event Error : string</code>	Outgoing event with payload of type <code>string</code> . Can be raised by a transition or state reaction with <code>raise Error : "Some error message"</code>
<code>var brightness : integer</code>	Variable, used to store some data. Can be changed by the state machine and by the client code.
<code>var brightness : integer = 3</code>	Variable with initial value.
<code>var readonly brightness : integer</code>	Variable marked as readonly to ensure it is not changed by the client code.
<code>const PI : real = 3.14</code>	Constant, used to store some immutable data that is not changeable by the client code or the state machine. Constants must have an initial value.
<code>operation average (a : real, b : real) : real</code>	Operation, connects a state machine to the outside world by making external behaviour accessible. Implementation needs to be provided by the client code.

Typed elements must have one of the following types: *integer*, *real*, *boolean* or *string*.

## INTERNAL SCOPE

The internal scope declares the entities that are only used internally by the statechart and hence are not visible to the outside.

Internal Element Declaration	Meaning
<code>event Process</code>	Internal event, can only be raised by the state machine.
<code>var brightness : integer</code>	Internal variable, only visible by the state machine, not by the client code.
<code>const PI : real = 3.14</code>	Internal constant, only visible by the state machine, not by the client code.

```
operation average
(a : real, b : real) : real
```

Operation, connects a state machine to the outside world by making external behaviour accessible. Implementation needs to be provided by client code.

```
every 500ms / raise Process
```

Statechart reaction, is evaluated on each run cycle. Used to specify reactions that are independent of the current state.

For more details on the definition section, consult the [language reference](#).

## STATES AND TRANSITIONS

States and transitions are the basic building blocks of a statechart. States and transitions are contained in [regions](#). At each point of a statechart's execution, there is at most one active state per region.

States are connected by transitions. Transitions are directed, therefore states have incoming and outgoing transitions. All states must have at least one incoming transition.

## TRANSITION REACTIONS

Transition reactions specify under which conditions a state transition is taken. Reactions have the following syntax:

```
trigger [guard] / effect
```

A state transition is taken when its trigger is raised and the guard condition is satisfied. When the transition is taken, its effect actions are executed. Guards and effects are optional.

# TRIGGERS

A transition reaction can specify the following **triggers**:

Trigger Syntax Examples	Meaning
<code>ev1</code>	Event trigger, triggers when the event <code>ev1</code> is raised. The used event needs to be declared in the definition section.
<code>ev1, ev2</code>	Multiple event triggers, triggers when one of the events <code>ev1</code> or <code>ev2</code> is raised. The used events need to be declared in the definition section.
<code>after 10s</code>	Time trigger, trigger after given amount of time.
<code>always</code>	Always trigger, triggers always. Can be omitted when used with a guard.
<code>oncycle</code>	Oncycle trigger, same as <code>always</code> trigger.
<code>else</code>	Else trigger, only valid on outgoing transitions of choice states to denote the default transition if no other outgoing transition can be taken.
<code>default</code>	Default trigger, same as <code>else</code> trigger.

# GUARDS

The reaction **guard** is optional. If specified, it needs to be a boolean expression. Here are some examples of valid guard conditions:

Guard Syntax Examples	Expression Kind
<code>[var1 &amp;&amp; !var2]</code>	Logical AND, OR, NOT
<code>[var1 &gt; 0 &amp;&amp; var1 &lt;= 10]</code>	Logical comparisons <, <=, >, >=
<code>[var1 == 10 &amp;&amp; var2 != 17]</code>	Logical equality or inequality

<code>[isOdd(var1)]</code>	Operation calls with boolean return type
<code>[var1]</code>	Boolean variables or constants

## EFFECTS

The reaction **effect** is optional. If specified, the effect is executed when the transition is taken. Multiple effects are separated by a semicolon. The last effect has no trailing semicolon.

Effect Syntax Examples	Meaning
<code>/ var1+=10; var2=var1</code>	Variable assignment
<code>/ calculate(var1, var2)</code>	Operation call
<code>/ raise ev1</code>	Event raising
<code>/ raise ev2 : 42</code>	Event raising with payload
<code>/ var1 &gt; 10 ? var1=0 : var1++</code>	Conditional expression
<code>/ var1 &lt;&lt; 8</code>	Bit shifting

## STATE REACTIONS

A state can also define reactions. The syntax is the same as for transitions. In addition to the above mentioned examples, a state can also define entry and exit reactions.

State Reaction Examples	Meaning
<code>entry / var1=10</code>	Entry reaction, is executed when the state is entered.
<code>exit / var1=0</code>	Exit reaction, is executed when the state is exited.
<code>ev1 / var1+=1</code>	Local reaction, is executed when no outgoing transition can be taken.

# BUILT-IN FUNCTIONS

The statechart language comes with two built-in functions that can be used inside a guard or effect expression:

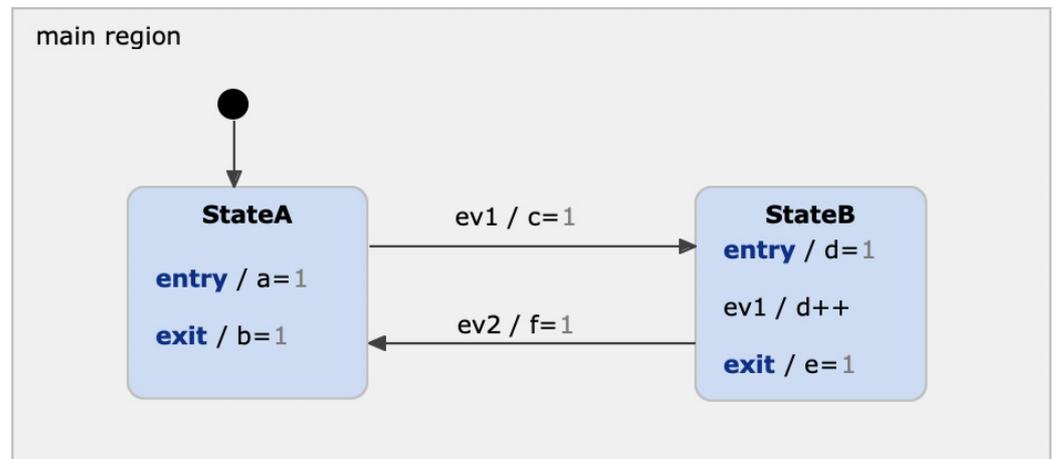
Built-in Function	Meaning
<code>valueOf(event)</code>	Returns the payload of an event. Note, that the event needs to be raised when <code>valueOf</code> is called.
<code>active(state)</code>	Returns <code>true</code> if the given state is active, otherwise <code>false</code> . This function is especially useful in combination with orthogonality.

# BASIC EXECUTION FLOW

When a state transition occurs, the specified reaction effects are executed in a defined **execution order**:

- 1 All source state's **exit** actions are executed
- 2 All transition actions are executed
- 3 All target state's **entry** actions are executed

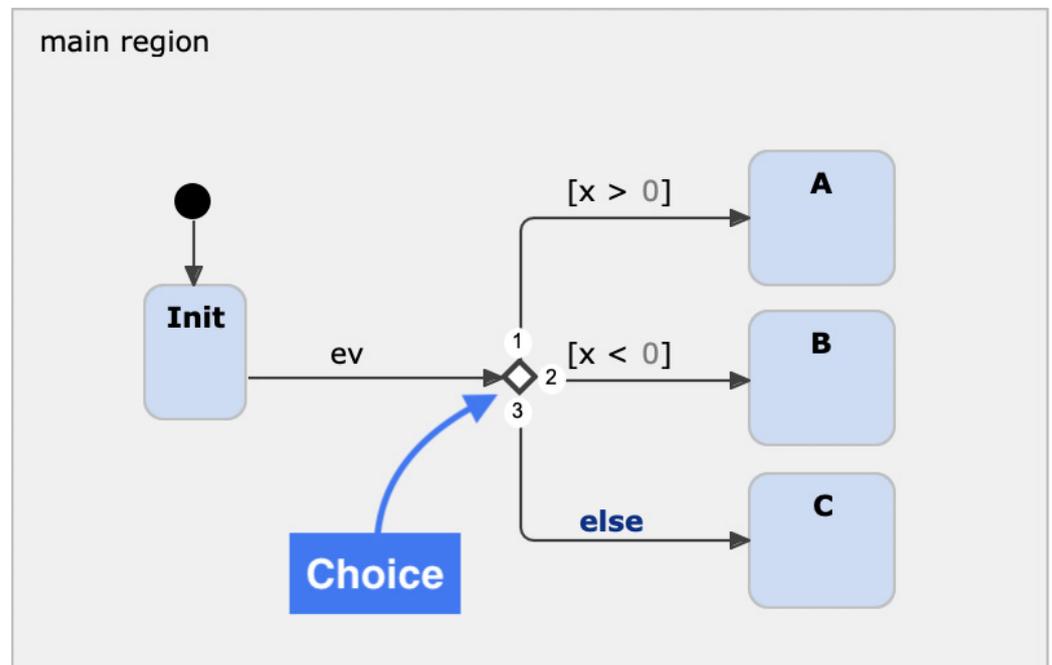
Consider the following simple example:



When **StateA** is entered, its entry reaction is executed first ( $a=1$ ). When event **ev1** is raised, the transition towards **StateB** is taken. As **StateA** is left, its exit reaction is executed ( $b=1$ ) before the transition reaction is executed ( $c=1$ ), following by entering **StateB** and executing its entry reaction ( $d=1$ ). While **StateB** is active, each time the event **ev1** is raised, the state's local reaction is executed ( $d++$ ). Note, that this is a different behavior compared to **StateB** having an outgoing transition pointing to itself, as taking such a self-transition would also invoke the state's exit and entry reactions. Finally, when event **ev2** is raised, **StateB** is left and its exit reaction is executed ( $e=1$ ), followed by the transition's reaction ( $f=1$ ), and **StateA**'s entry reaction ( $a=1$ ).

## CHOICES

A choice is a pseudo state. It is used to model a conditional path. If a choice's incoming transition is taken, its outgoing transitions are immediately evaluated to decide which path to take. To ensure there is always a valid path, a default transition can be defined with the trigger **else** or **default**.



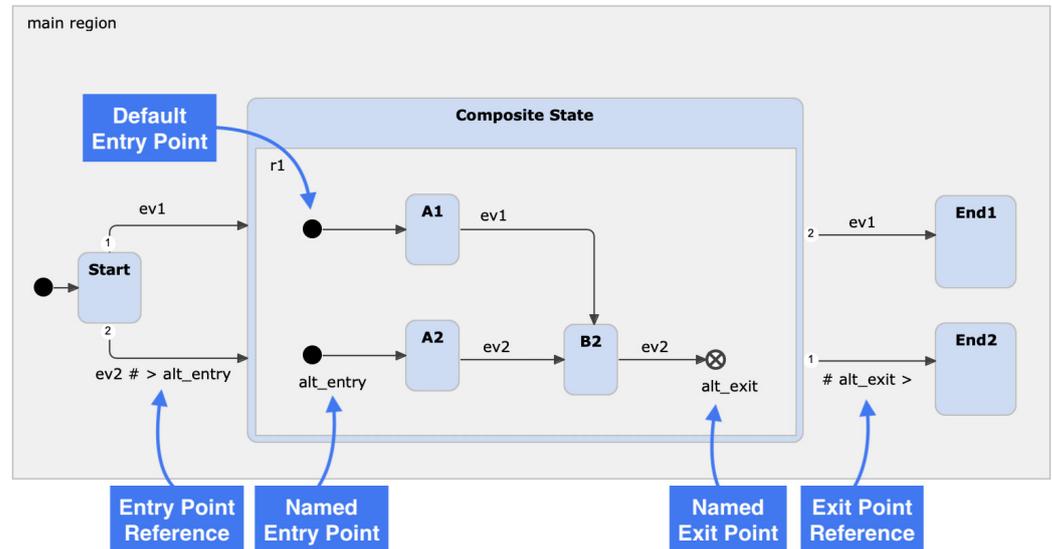
## COMPOSITE STATES

YAKINDU Statechart Tools allows to express state hierarchies by the means of composite states and subdiagrams.

A composite state is a state that contains one or more other substates. It can be used to group states into logical compounds and thus make the statechart more comprehensible.

When a composite state is entered, its entry node denotes the substate to be activated. A composite state can specify multiple entry nodes with unique names. Incoming transitions of the composite state can specify the desired entry node to take for entering the composite state.

When a composite state is left, all active substates are also left. A composite state can specify multiple exit nodes with unique names. Outgoing transitions of the composite state can specify the relevant exit node for them.



The syntax for referencing entry or exit points in a transition reaction is the following:

Transition Reaction Examples	Meaning
# > entry-point-1	Enters the target composite state by the entry point entry-point-1
# exit-point-1 >	Exit the composite state by this transition if exit point entry-point-1 is active
# exit-point-1 > exit-point-2 >	Exit the composite state by this transition if one of the exit points entry-point-1 or entry-point-2 is active

## PARENT-FIRST VS. CHILD-FIRST EXECUTION

The parent-first and child-first execution schemes define in which order a composite state and its substates are processed:

- In the parent-first execution scheme, the composite state (parent) is processed first, before its substates are processed.
- In the child-first execution scheme, the active substate (child) is processed first, before its parent composite state is processed.

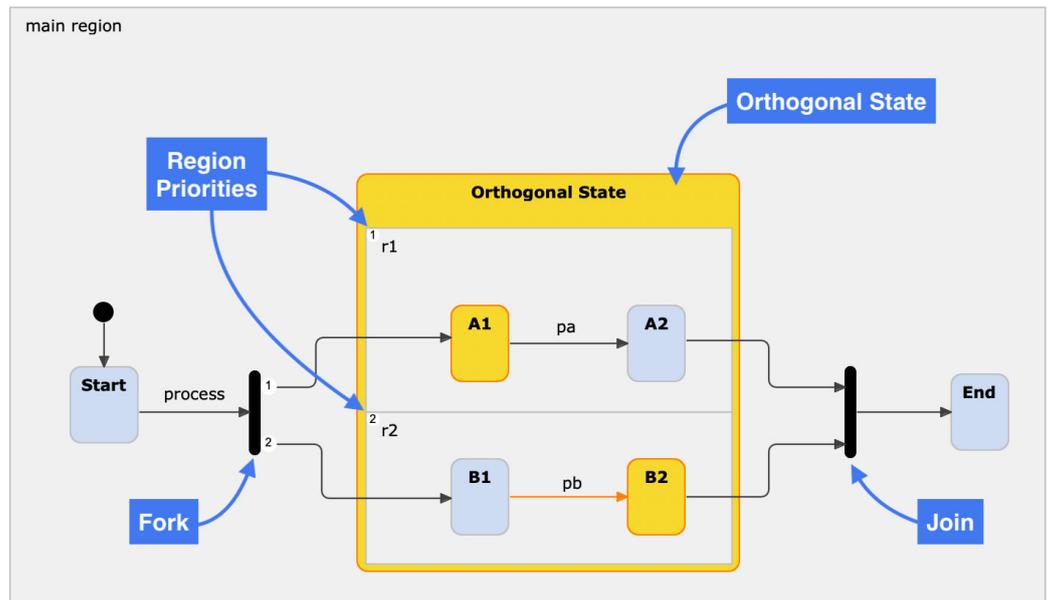
Consider the previous example model. When state **A1** is active and the event **ev1** is raised, it depends on the execution scheme whether the transition to state **B2** (child-first execution) or the one to state **End1** (parentfirst execution) is taken.

The execution scheme is specified by the `@ParentFirstExecution` resp. `@ChildFirstExecution` annotation in the definition section. For more details, take a look at the [language reference](#).

## ORTHOOGONALITY

YAKINDU Statechart Tools allows to specify orthogonal regions that are executed virtually concurrently. Orthogonal regions can be modeled either on top level, or within a composite state (or subdiagram). They allow to describe that the modeled system can be in multiple states simultaneously.

Orthogonal regions are executed in a deterministic sequential order and not in parallel as one might expect. The execution order is defined by the regions' priorities. These are indicated in the top left corner of a region. The defined execution order plays a particular role when orthogonal regions raise and react to the same events. For more details, see also chapter Raising and processing an [event](#).



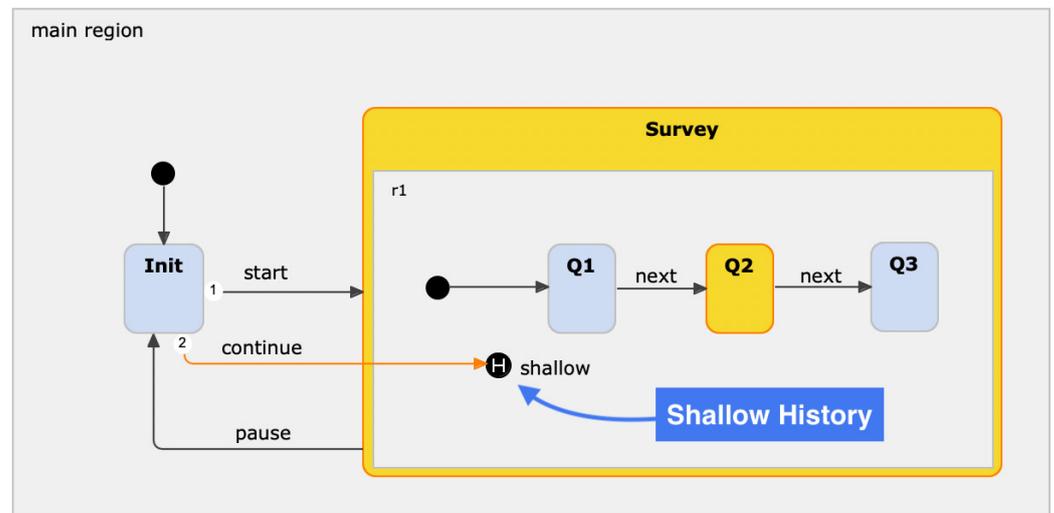
# SYNCHRONIZATIONS (FORKS AND JOINS)

Orthogonal regions can be defined on top level or within composite states. The semantics explained above are the same. The example model above uses a synchronization node to fork the execution flow into both orthogonal regions *r1* and *r2*. After both regions have executed their state transitions, the execution flow is joined again by a synchronization node. A joining synchronization is only executed when all incoming transitions can be taken within the same run-to-completion cycle. Read more about synchronization nodes in the [language reference](#).

# HISTORY NODES

A *shallow* history state is a pseudo state that is placed inside the region of a composite state. It is used to remember the last active state inside a composite state. This makes it possible to jump back to the remembered state instead of starting at the initial sub-state again.

A deep history state is similar to a shallow history state, but more complex. With a deep history state, the latest status of all nested states is remembered.



# FINAL STATE

A final state denotes the end of the execution flow of a state machine or region. It can have multiple incoming transitions but no outgoing ones. Each region may contain at most one final state. In case of orthogonal regions, the execution flow stops when all regions' final states have been reached.

# EVENT-DRIVEN VS. CYCLE-BASED EXECUTION

The state machine can define one of two different execution schemes:

- In the **cycle-based** execution scheme, a run-to-completion step is executed **periodically** in regular time intervals.
- In the **event-driven** execution scheme, a run-to-completion step is executed **each time an event is raised**.

The execution scheme is selected in the definition section by either using the `@CycleBased` or `@EventDriven` annotation. If nothing is specified, the cycle-based execution scheme with a time interval of 200 milliseconds is used for simulation. For a better understanding, see also this [example](#), or the more elaborate explanation in the [language reference](#).

