YAKINDU C code generator

Web-Clip

1 ode generator

"."
cou
nte
r(h
2,
dec
ima
l)

This chapter describes the required steps for generating C++ code with YAKINDU Statechart Tools. Furthermore, all components of the generated code will be described in detail and each configurable generator feature will be explained.

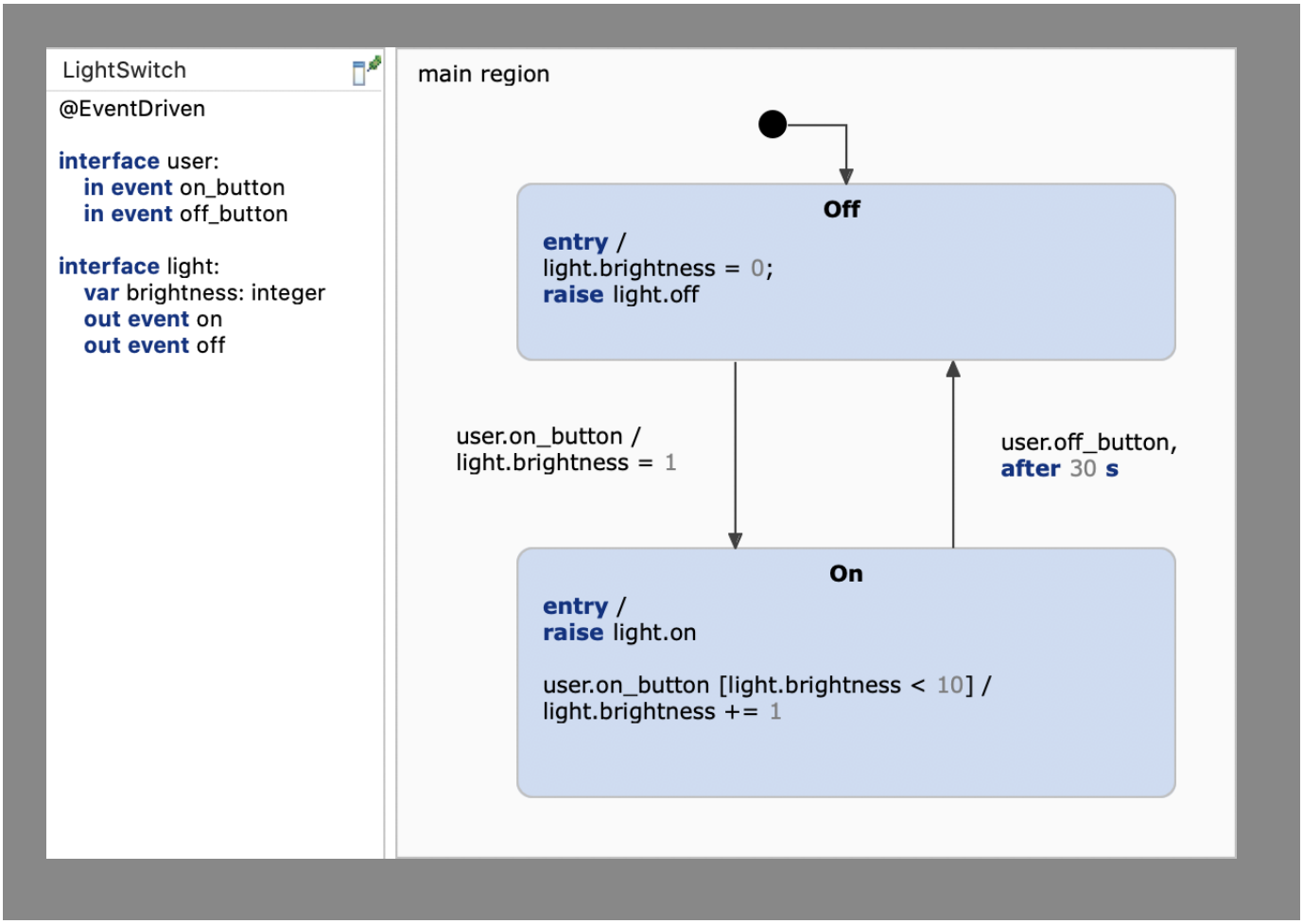
Table of content:

1 . counter(h2, decimal) . counter(h3, decimal) . Statechart example model

We will use the following example to explain the code generation, the generated code and the integration with client code. The example model describes a light switch with two states and the following behavior:

> The *on_button* event turns the light on and, upon repeated pushing, turns the brightness higher until the latter's maximum is reached.
> The *off_button* event turns the light off.
> When the light is on, it automatically turns off after 30 seconds.
> Whenever a state is entered, an outgoing event is raised.

The ‚C Code Generation' example can be found in the example wizard:
File -> New -> Example... -> YAKINDU Statechart Examples -> Getting Started – Code Generation -> C Code Generation



Statechart example model

2 . counter(h2, decimal) . counter(h3, decimal) . Generating C code

Generating C code from a statechart requires a generator file (.sgen). It must

at least specify the *yakindu::c* generator, reference a statechart and define the *targetProject* and *targetFolder* in the *Outlet* feature. By specifying these attributes, C state machine code can be generated.

Example:

```
GeneratorModel for yakindu::c {

	statechart LightSwitch {

		feature Outlet {
			targetProject = "org.yakindu.sct.examples.codegen.c"
			targetFolder = "src-gen"
		}
	}
}
```

You can create a generator model with the *YAKINDU Statechart generator model* wizard by selecting *File → New → Code generator model*.

The code generation is performed automatically whenever the statechart or the generator file is modified. See also chapter Running a generator for more information.

3 . counter(h2, decimal) . counter(h3, decimal) . Generated code files

Generated code files can be categorized into *base*, *api* and *library* files.

The base source file is the implementation of the state machine model. It is generated into the folder defined by the *targetFolder* parameter. The file name is derived from the statechart name, but can be overridden by the *moduleName* parameter.

> LightSwitch.c: Implementation of the state machine. It implements the API defined in *LightSwitch.h*.

API files are the header files that expose the state machine's API. They are generated into the *apiTargetFolder* or, if that one is not defined, into the *targetFolder* .

LightSwitch.h: Contains several data types as well as the API functions to run the state machine, raise events, access variables and so on.
LightSwitch_required.h: Contains declarations of functions the developer has to implement. It is generated only if at least one of the following conditions is fulfilled:

The statechart declares operations in its interface(s).
The statechart uses timed event triggers, like *every* or *after* clauses.
The code generator model activates the Tracing feature.

Library files are independent of the concrete state machine model. They are generated into the *libraryTargetFolder* , or if that one is not defined, into the *targetFolder* .

sc_types.h: Contains type definitions used by the statechart. Since the contents of this file is always the same for all statecharts, it will be generated only if it does not yet exist. And since it will never be overwritten, you can change or amend the definitions made there. For example, you might wish to adapt the types to better match your target platform.
sc_rxc.h: Contains declarations of data types and functions used for the observer mechanism.
sc_rxc.c: Contains the function implementions for *sc_rxc.h*.
sc_timer_service.h: Header of the Default timer service. It is only generated if enabled in the generator model.
sc_timer_service.c: Implementation of the default timer service declared in *sc_timer_service.h*. It is only generated if enabled in the generator model.

1 . counter(h2, decimal) . counter(h3, decimal) . counter(h4, decimal) . Fundamental statechart functions

The generated header file (here *LightSwitch.h*) contains fundamental functions to initialize, enter, and exit a state machine.

In the header file, the function names are made up of the state machine name followed by the name of the respective functionality. For the light switch

example, these functions are generated as follows:

```
/*! Initializes the LightSwitch state machine data structures. Must be called before first
usage.*/
extern void lightSwitch_init(LightSwitch* handle);

/*! Activates the state machine. */
extern void lightSwitch_enter(LightSwitch* handle);

/*! Deactivates the state machine. */
extern void lightSwitch_exit(LightSwitch* handle);
```

The *init()* function is used to initialize the internal objects of the state machine right after its instantiation. Variables are initialized to their respective default values. If the statechart defines any initialized variables, these initializations are also done in the *init()* function.
The *enter()* method must be called to enter the state machine. It brings the state machine to a well-defined state.
The *exit()* method is used to leave a state machine statefully. If for example a history state is used in one of the top regions, the last active state is stored and the state machine is left via exit(). Re-entering it via enter() continues to work with the saved state.

State machines that use the cycle-based execution scheme will additionally expose a *runCycle()* method:

```
/*! Performs a 'run to completion' step. */
extern void lightSwitch_run_cycle(LightSwitch* handle);
```

The *run_cycle()* method is used to trigger a run-to-completion step in which the state machine evaluates arising events and computes possible state changes. For event-driven statecharts, this method is called automatically when an event is received. For cycle-based statecharts, this methods needs to be called explicitly in the client code. See also chapter Execution schemes. Somewhat simplified, a run-to-completion cycle consists of the following steps:

Clear the list of outgoing events.
Check whether any events have occurred which are leading to a state change.
If a state change has to be done:

Make the present state inactive.
Execute exit actions of the present state.
Save history state, if necessary.
Execute transition actions, if any.
Execute entry actions of the new state.
Make the new state active.

Clear the list of incoming events.

Furthermore, the header file declares methods to check whether the state machine is active, final or whether a specific state is active:

```
/*!
 * Checks whether the state machine is active.
 * A state machine is active if it was entered. It is inactive if it has not been entered at all
or if it has been exited.
 */
extern sc_boolean lightSwitch_is_active(const LightSwitch* handle);

/*!
 * Checks if all active states are final.
 * If there are no active states then the state machine is considered being inactive. In this
case this method returns false.
 */
extern sc_boolean lightSwitch_is_final(const LightSwitch* handle);

/*! Checks if the specified state is active. */
extern sc_boolean lightSwitch_is_state_active(const LightSwitch* handle, LightSwitchStates
state);
```

The *is_active()* method checks whether the state machine is active, i.e. at least one state of the machine is active. A state machine is active if it has been entered. It is inactive if it has not been entered at all or if it has been exited.
The *is_final()* method checks whether the all active states are final. If there are no active states then the state machine is considered being inactive. In this case this method returns `false`.
The *is_state_active()* method checks whether the specified state is active.

2 . counter(h2, decimal) . counter(h3, decimal) . counter(h4, decimal) . Accessing variables and events

The client code can read and write state machine variables and raise state machine events. The getters and setters for each variable and event are also

machine events. The getters and setters for each variable and event are also contained in the header file. The function names are matching the following pattern:

[*statechart_name*] _ [*interface_name*] _ [`set` | `get` | `raise`] _ [*variable_name* | *event_name*]

A statechart can also define an unnamed interface. In this case, the interface name is ommitted from the function name.

For the example model above, the two statechart interfaces *user* and *light* are transformed into the following API functions:

```
/*! Gets the value of the variable 'brightness' that is defined in the interface scope 'light'.
*/
extern sc_integer lightSwitch_light_get_brightness(const LightSwitch* handle);
/*! Sets the value of the variable 'brightness' that is defined in the interface scope 'light'.
*/
extern void lightSwitch_light_set_brightness(LightSwitch* handle, sc_integer value);

/*! Raises the in event 'on_button' that is defined in the interface scope 'user'. */
extern void lightSwitch_user_raise_on_button(LightSwitch* handle);
/*! Raises the in event 'off_button' that is defined in the interface scope 'user'. */
extern void lightSwitch_user_raise_off_button(LightSwitch* handle);

/*! Returns the observable for the out event 'on' that is defined in the interface scope
'light'. */
extern sc_observable* lightSwitch_light_get_on(LightSwitch* handle);
/*! Returns the observable for the out event 'off' that is defined in the interface scope
'light'. */
extern sc_observable* lightSwitch_light_get_off(LightSwitch* handle);
```

In this example code you can see the following:

- Statechart variables are transformed into corresponding getters and setters (see *lightSwitch_light_get_brightness()* and *lightSwitch_light_set_brightness()*).
- Incoming events are transformed into methods to raise such an event (see lightSwitch_user_raise_on_button()_ and *lightSwitch_user_raise_off_button()*).
- Outgoing events are transformed into observable objects to which the client code can subscribe (see *lightSwitch_light_get_on()* and *lightSwitch_light_get_off()*).

Bringing it all together, the state machine header file declares the following API functions for the light switch example:

```
extern void lightSwitch_init(LightSwitch* handle);
extern void lightSwitch_enter(LightSwitch* handle);
extern void lightSwitch_exit(LightSwitch* handle);

extern void lightSwitch_raise_time_event(LightSwitch* handle, sc_eventid evid);

extern void lightSwitch_user_raise_on_button(LightSwitch* handle);
extern void lightSwitch_user_raise_off_button(LightSwitch* handle);

extern sc_integer lightSwitch_light_get_brightness(const LightSwitch* handle);
extern void lightSwitch_light_set_brightness(LightSwitch* handle, sc_integer value);

extern sc_observable* lightSwitch_light_get_on(LightSwitch* handle);
extern sc_observable* lightSwitch_light_get_off(LightSwitch* handle);

extern sc_boolean lightSwitch_is_active(const LightSwitch* handle);
extern sc_boolean lightSwitch_is_final(const LightSwitch* handle);
extern sc_boolean lightSwitch_is_state_active(const LightSwitch* handle, LightSwitchStates
state);
```

The following code snippet demonstrates how to use the state machine API:

```
/*! Instantiates the state machine */
LightSwitch lightSwitch;

/*! Initializes the state machine, in particular all variables are set to a proper value */
lightSwitch_init(&lightSwitch);

/*! Enters the state machine; from this point on the state machine is ready to react on incoming
event */
lightSwitch_enter(&lightSwitch);

/*! Raises the On event in the state machine which causes the corresponding transition to be
taken */
lightSwitch_user_raise_on_button(&lightSwitch);

/*! Prints the value of the brightness variable */
printf("Brightness: %d.\n", lightSwitch_light_get_brightness(&lightSwitch));

/*! Exit the state machine */
lightSwitch_exit(&lightSwitch);
```

1 . counter(h2, decimal) . counter(h3, decimal) . counter(h4, decimal) . counter(h5, decimal) . Observing outgoing events

There are basically two ways to access outgoing events, getters and observables. The desired option can be enabled in the generator model, see *OutEventAPI* .

The getter mechanism is straight forward and simply allows to check if an outgoing event is raised by calling an *is_raised* function that returns a boolean:

```
/*! Checks if the out event 'on' that is defined in the interface scope 'light' has been raised.
*/
extern sc_boolean lightSwitch_light_is_raised_on(const LightSwitch* handle);

/*! Checks if the out event 'off' that is defined in the interface scope 'light' has been
raised. */
extern sc_boolean lightSwitch_light_is_raised_off(const LightSwitch* handle);
```

The observable mechanism is more complex to set up, but it allows to get notified whenever the event is raised. Thus, the client code does not need to check the event status explicitly. The client code basically passes on a pointer to a function that is called when the outgoing event is raised.

First of all, we need to specify the callback function:

```
/*! This function will be called by raising the out event light.on */
static void on_light_on(LightSwitch *o) {
        printf("Light is on.\n");
}
```

Then, we need to instantiate an observer, initialize it with the callback function, and subscribe it to the out event observable:

```
/*! Instantiate observer for the out events */
sc_single_subscription_observer lightOnObserver;

/*! Initialize the observer with the callback function
sc_single_subscription_observer_init(lightOnObserver, lightSwitch, (sc_observer_next_fp)
on_light_on)

/*! Subscribe the observer to the out event observable */
sc_single_subscription_observer_subscribe(lightOnObserver, &lightSwitch->ifaceLight.on);
```

With that code, our *on_light_on()* function will be called whenever the out event *on* in interface *light* is raised by the state machine.

3 . counter(h2, decimal) . counter(h3, decimal) . counter(h4, decimal) . Serving operation callbacks

YAKINDU Statechart Tools support *operations* that are executed by a state machine as actions, but are implemented by client-side code.

As a simple example a function *myOp* can be defined in the definition section of the *LightSwitch* example:

```
interface:
operation myOp()
```

For state machines that define operations in their interface(s), the code generator adds corresponding functions to the *required.h* header file (in our example *LightSwitch_required.h*):

```
extern void lightSwitch_myOp( LightSwitch* handle);
```

This function has to be implemented and linked with the generated code, so that the state machine can use it.

4 . counter(h2, decimal) . counter(h3, decimal) . counter(h4, decimal) . Time-controlled state machines

If a statechart uses timing functionality or external operations, or tracing is activated in the generator model, an additional header file is generated. Its name matches the following pattern:

[*statechart_name*] `_required.h`

This header file declares functions the client code has to implement externally.

The light switch example model is such a time-controlled state machine as it uses an *after* clause at the transition from state *On* to state *Off*. Conseuquently, the code generator produces the *LightSwitch_required.h* header file which declares two function to set and unset a timer:

```
/*! This function has to set up timers for the time events that are required by the state
machine. */
/*!
        This function will be called for each time event that is relevant for a state when a
state will be entered.
        \param evid An unique identifier of the event.
        \time_ms The time in milliseconds
        \periodic Indicates the the time event must be raised periodically until the timer is
unset
*/
extern void lightSwitch_set_timer(LightSwitch* handle, const sc_eventid evid, const sc_integer
time_ms, const sc_boolean periodic);
```

```
/*! This function has to unset timers for the time events that are required by the state
machine. */
/*!
        This function will be called for each time event that is relevant for a state when a
state will be left.
        \param evid An unique identifier of the event.
*/
extern void lightSwitch_unset_timer(LightSwitch* handle, const sc_eventid evid);
```

Basically the proper time handling has to be implemented by the developer, because timer functions generally depend on the hardware target used. So for each hardware target the client code must provide a function to set a timer and another function to unset it. These functions have to be implemented externally and have to be linked to the generated code.

1 . counter(h2, decimal) . counter(h3, decimal) . counter(h4, decimal) . counter(h5, decimal) . Function set_timer

A state machine calls the *set_timer()* function to tell the timer service that it has to start a timer for the given time event identifier and raise it after the period of time specified by the *time_ms* parameter has expired. It is important to only start a timer thread or a hardware timer interrupt within the *set_timer()* function and avoid any time-consuming operations like extensive computations, sleeping or waiting. Never call the statechart API functions from within these functions! Otherwise the state machine execution might hang within the timer service or might not show the expected runtime behavior.

If the parameter *periodic* is *false*, the timer service is supposed to raise the time event only once. If *periodic* is *true*, the timer service is supposed to raise the time event every *time_ms* milliseconds.

2 . counter(h2, decimal) . counter(h3, decimal) . counter(h4, decimal) . counter(h5, decimal) . Function unset_timer

The state machine calls the function *unset_timer()* to notify the timer service to unset the timer for the given event ID.

3 . counter(h2, decimal) . counter(h3, decimal) . counter(h4, decimal) . counter(h5, decimal) . Function raise_time_vent

In order to notify the state machine about the occurence of a time event after a

period of time has expired, the *raise_time_event()* function – defined in the header file of the state machine – needs to be called on the state machine. In the case of the light switch example it is named *lightSwitch_raise_time_event(LightSwitch* handle, sc_eventid evid)* (in file *LightSwitch.h*).

The time event is recognized by the state machine and will be processed during the next run cycle.

You can conclude that in order to process the time events raised by the timing service without too much latency, the runtime environment has to call the state machine's *run_cycle()* function as frequently as needed. Consider for example a time event which is raised by the timer service after 500 ms. However, if the runtime environment calls the state machine's *run_cycle()* function only once per 1000 ms, the event will quite likely not be processed at the correct points in time.

4 . counter(h2, decimal) . counter(h3, decimal) . counter(h4, decimal) . counter(h5, decimal) . Default timer service implementation

The C code generator can create a default implementation of the timer service (software timer).

To generate the default timer service class, set the *timerService* feature in the generator model to *true*. Example:

```
GeneratorModel for yakindu::c {

   statechart LightSwitch {

    /* … */

    feature GeneralFeatures {
      timerService = true
    }
  }
}
```

The default timer service consists of a header file named *sc_timer_service.h* and a corresponding source file *sc_timer_service.c*. The header file defines the timer service API implemented by the source file:

```
/*! Initializes a timer service with a set of timers. */
extern void sc_timer_service_init(sc_timer_service_t *tservice, sc_timer_t *timers, sc_integer
count, sc_raise_time_event_fp raise_event);

/*! Updates all timers. */
extern void sc_timer_service_proceed(sc_timer_service_t *timer_service, const sc_integer
time_ms);

/*! Starts a timer with the specified parameters. */
extern void sc_timer_set(sc_timer_service_t *timer_service, void *handle, const sc_eventid evid,
const sc_integer time_ms, const sc_boolean periodic);

/*! Cancels a timer for the specified time event. */
extern void sc_timer_unset(sc_timer_service_t *timer_service, const sc_eventid evid);
```

The *init()* function intialized the timer service with a data structure that holds the individual timers, and a pointer to the state machine's *raise_time_event()* function. In order to start or cancel a timer, the functions *sc_timer_set()* and *sc_timer_unset()* can be invoked. However, the timer service does not know anything about the current time. Its internal clock needs to be pushed forward manually by calling the *sc_timer_service_proceed()* function. This function checks if an already started timer expired, and if so, raises the corresponding time event on the state machine.

The following code snippet demonstrates how the timer service can be used:

```
#include <sys/time.h>

#include "../src-gen/LightSwitch.h"
#include "../src-gen/sc_timer_service.h"

/* ! As we make use of time triggers (after & every)
 * we make use of a generic timer implementation
 * and need a defined number of timers. */
#define MAX_TIMERS 4

//! We allocate the desired array of timers.
static sc_timer_t timers[MAX_TIMERS];

//! The timers are managed by a timer service. */
static sc_timer_service_t timer_service;

unsigned long current_time = 0;
unsigned long last_time = 0;
unsigned long elapsed_time = 0;

unsigned long get_ms() {
        struct timeval tv;
        unsigned long ms;
        gettimeofday(&tv, 0);
```

```
        gettimeofday(&tv, 0);
        ms = tv.tv_sec * 1000 + (tv.tv_usec / 1000);
        return ms;
}

int main(int argc, char **argv) {
        /*! Instantiates the state machine */
        LightSwitch lightSwitch;

        /*! Initializes the timer service */
        sc_timer_service_init(&timer_service, timers, MAX_TIMERS,
                        (sc_raise_time_event_fp) &lightSwitch_raise_time_event);

        /*! Initializes the state machine, in particular all variables are set to a proper value
*/
        lightSwitch_init(&lightSwitch);

        /*! Enters the state machine; from this point on the state machine is ready to react on
incoming event */
        lightSwitch_enter(&lightSwitch);

        /*! Application loop */
        last_time = get_ms();
        while (1) {
                current_time = get_ms();
                elapsed_time = current_time - last_time;
                last_time = current_time;
                /*! push the timer service clock forward */
                sc_timer_service_proceed(&timer_service, current_time - last_time);

                // interact with the state machine, raise events etc..
        }
        return 0;
}

/*! This function will be called for each time event in LightSwitch when a state is entered. */
void lightSwitch_set_timer(LightSwitch *handle, const sc_eventid evid, const sc_integer time_ms,
const sc_boolean periodic) {
        sc_timer_set(&timer_service, handle, evid, time_ms, periodic);
}

/*! This function will be called for each time event in LightSwitch when a state will be left.
*/
void lightSwitch_unset_timer(LightSwitch *handle, const sc_eventid evid) {
        sc_timer_unset(&timer_service, evid);
}
```

4 . counter(h2, decimal) . counter(h3, decimal) . C code generator features

Beside the general code generator features, there are language specific generator features, which are listed in the following chapter.

1 . counter(h2, decimal) . counter(h3, decimal) . counter(h4, decimal) . Outlet feature

The Outlet feature specifies target project and target folder for the generated artifacts. It is a *required* feature and has the parameters as described in Outlet feature .

The C code generator extends this feature by the following parameter:

> *apiTargetFolder* (String, optional): The folder to write API code to, i.e. the statechart specific header files. If this parameter is not specified, these artifacts will be generated into the *target folder* (see Outlet feature).

Example:

```
apiTargetFolder = "api-gen"
```

2 . counter(h2, decimal) . counter(h3, decimal) . counter(h4, decimal) . IdentifierSettings feature

The IdentifierSettings feature allows the configuration of module names and identifier character length:

> *moduleName* (String, optional): Name for header and implementation. By default, the name of the statechart is used.
> *statemachinePrefix* (String, optional): Prefix that is prepended to function, state, and type names. By default, the name of the statechart is used.
> *separator* (String, optional): Character to replace whitespace and otherwise illegal characters in names.

Please note that the *maxIdentifierLength* option, which existed in older versions of YAKINDU Statechart Tools, has been removed in favor of a statechart annotation that is only available in the C/C++ domain bundled with YAKINDU Statechart Tools Professional Edition, see @ShortIdentifiers.

Example:

```
feature IdentifierSettings {
    moduleName = "MyStatechart"
    statemachinePrefix = "myStatechart"
    separator = "_"
}
```

3 . counter(h2, decimal) . counter(h3, decimal) . counter(h4, decimal) . GeneratorOptions feature

The GeneratorOptions feature allows the configuration of additional aspects concerning the behavior of the generated code:

userAllocatedQueue (Boolean, optional): If you want to allocate the buffer used by queues yourself, use this option. Defaults to false.

Example:

```
feature GeneratorOptions {
    userAllocatedQueue = true
}
```

4 . counter(h2, decimal) . counter(h3, decimal) . counter(h4, decimal) . Tracing feature

The Tracing feature enables the generation of tracing callback functions:

enterState (boolean, optional): Specifies whether to generate a callback function that is used to notify about state-entering events.
exitState (boolean, optional): Specifies whether to generate a callback that is used to notify about state-exiting events.
generic (boolean, optional): Specifies whether to generate callbacks that are used to notify about any changes. These can be implemented manually or by the YET Tracer.

Example:

```
feature Tracing {
    enterState = true
    exitState  = true
    generic = true
}
```

5 . counter(h2, decimal) . counter(h3, decimal) . counter(h4, decimal) . YET Tracer

The YET Tracer is a C generator extension, which allows the use of the YET Tracing run option. The generated C code implements the callback funtions, which fit into the generated architecture of the generic tracing feature. To use the YET Tracer, please also supply a C generator with generic Tracing feature enabled.

```
GeneratorModel for yakindu::c::yet {
        ...
}
```

6 . counter(h2, decimal) . counter(h3, decimal) . counter(h4, decimal) . Includes feature

The Includes feature allows to change how include statements are generated:

> *useRelativePaths* (Boolean, optional): If this parameter is set to *true*, relative paths are calculated for include statements, otherwise simple includes are used. Default: *true*.

Example:

```
feature Includes {
    useRelativePaths = false
}
```

7 . counter(h2, decimal) . counter(h3, decimal) . counter(h4, decimal) . GeneralFeatures feature

The GeneralFeatures feature allows to configure additional services to be generated along with the state machine. Per default, all parameters are *false*, meaning to disable the corresponding features, respectively.

GeneralFeatures is an *optional* feature and has the following parameters:

> *timerService* (Boolean, optional): Enables/disables the generation of a software timer service implementation.

Example:

```
feature GeneralFeatures {
    timerService = true
}
```

← Configuring a generator

→ C++ code generator