Chapter 68. Boost.MetaStateMachine

Boost.MetaStateMachine is used to define state machines. State machines describe objects through their states. They describe what states exist and what transitions between states are possible.

Boost.MetaStateMachine provides three different ways to define state machines. The code you need to write to create a state machine depends on the front-end.

If you go with the basic front-end or the function front-end, you define state machines in the conventional way: you create classes, derive them from other classes provided by Boost.MetaStateMachine, define required member variables, and write the required C++ code yourself. The fundamental difference between the basic front-end and the function front-end is that the basic front-end expects function pointers, while the function front-end lets you use function objects.

The third front-end is called eUML and is based on a domain-specific language. This front-end makes it possible to define state machines by reusing definitions of a UML state machine. Developers familiar with UML can copy definitions from a UML behavior diagram to C++ code. You don't need to translate UML definitions to C++ code.

eUML is based on a set of macros that you must use with this front-end. The advantage of the macros is that you don't need to work directly with many of the classes provided by Boost.MetaStateMachine. You just need to know which macros to use. This means you can't forget to derive your state machine from a class, which can happen with the basic front-end or the function front-end. This chapter introduces Boost.MetaStateMachine with eUML.

Example 68.1. Simple state machine with eUML

```
#include <boost/msm/front/euml/euml.hpp>
#include <boost/msm/front/euml/state_grammar.hpp>
#include <boost/msm/back/state_machine.hpp>
#include <iostream>

namespace msm = boost::msm;
using namespace boost::msm::front::euml;

BOOST_MSM_EUML_STATE((), Off)
BOOST_MSM_EUML_STATE((), On)

BOOST_MSM_EUML_EVENT(press)

BOOST_MSM_EUML_TRANSITION_TABLE((
  Off + press == On,
  On + press == Off
), light_transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
(light_transition_table, init_ << Off),
light_state_machine)

int main()
{
  msm::back::state_machine<light_state_machine> light;
  std::cout << *light.current_state() << '\n';
  light.process_event(press);
```

```
  std::cout << *light.current_state() << '\n';
  light.process_event(press);
  std::cout << *light.current_state() << '\n';
}
```

Example 68.1 uses the simplest state machine possible: A lamp has exactly two states. It is either on or off. If it is off, it can be switched on. If it is on, it can be switched off. It is possible to switch from every state to every other state.

Example 68.1 uses the eUML front-end to describe the state machine of a lamp. Boost.MetaStateMachine doesn't have a master header file. Therefore, the required header files have to be included one by one. `boost/msm/front/euml/euml.hpp` and `boost/msm/front/euml/state_grammar.hpp` provide access to eUML macros. `boost/msm/back/state_machine.hpp` is required to link a state machine from the front-end to a state-machine from the back-end. While front-ends provide various possibilities to define state machines, the actual implementation of a state machine is found in the back-end. Since Boost.MetaStateMachine contains only one back-end, you don't need to select an implementation.

All of the definitions from Boost.MetaStateMachine are in the namespace `boost::msm`. Unfortunately, many eUML macros don't refer explicitly to classes in this namespace. They use either the namespace `msm` or no namespace at all. That's why Example 68.1 creates an alias for the namespace `boost::msm` and makes the definitions in `boost::msm::front::euml` available with a `using` directive. Otherwise the eUML macros lead to compiler errors.

To use the state machine of a lamp, first define the states for off and on. States are defined with the macro `BOOST_MSM_EUML_STATE`, which expects the name of the state as its second parameter. The first parameter describes the state. You'll see later how these descriptions look like. The two states defined in Example 68.1 are called **`Off`** and **`On`**.

To switch between states, events are required. Events are defined with the macro `BOOST_MSM_EUML_EVENT`, which expects the name of the event as its sole parameter. Example 68.1 defines an event called **`press`**, which represents the action of pressing the light switch. Since the same event switches a light on and off, only one event is defined.

When the required states and events are defined, the macro `BOOST_MSM_EUML_TRANSITION_TABLE` is used to create a *transition table*. This tables defines valid transitions between states and which events trigger which state transitions.

`BOOST_MSM_EUML_TRANSITION_TABLE` expects two parameters. The first parameter defines the transition table, and the second is the name of the transition table. The syntax of the first parameter is designed to make it easy to recognize how states and events relate to each other. For example, `Off + press == On` means that the machine in the state **`Off`** switches to the state **`On`** with the event **`press`**. The intuitive and self-explanatory syntax of a transition table definition is one of the strengths of the eUML front-end.

After the transition table has been created, the state machine is defined with the macro `BOOST_MSM_EUML_DECLARE_STATE_MACHINE`. The second parameter is again the simpler one: it

sets the name of the state machine. The state machine in Example 68.1 is named `light_state_machine`.

The first parameter of `BOOST_MSM_EUML_DECLARE_STATE_MACHINE` is a tuple. The first value is the name of the transition table. The second value is an expression using **`init_`**, which is an attribute provided by Boost.MetaStateMachine. You'll learn more about attributes later. The expression `init_ << Off` is required to set the initial state of the state machine to **`Off`**.

The state machine `light_state_machine`, defined with `BOOST_MSM_EUML_DECLARE_STATE_MACHINE`, is a class. You use this class to instantiate a state machine from the back-end. In Example 68.1 this is done by passing `light_state_machine` to the class template `boost::msm::back::state_machine` as a parameter. This creates a state machine called **`light`**.

State machines provide a member function `process_event()` to process events. If you pass an event to `process_event()`, the state machines changes its state depending on its transition table.

To make it easier to see what happens in Example 68.1 when `process_event()` is called multiple times, `current_state()` is called. This member function should only be used for debugging purposes. It returns a pointer to an `int`. Every state is an `int` value assigned in the order the states have been accessed in `BOOST_MSM_EUML_TRANSITION_TABLE`. In Example 68.1 **`Off`** is assigned the value 0 and **`On`** is assigned the value 1. The example writes `0`, `1`, and `0` to standard output. The light switch is pressed two times, which switches the light on and off.

Example 68.2. State machine with state, event, and action

```
#include <boost/msm/front/euml/euml.hpp>
#include <boost/msm/front/euml/state_grammar.hpp>
#include <boost/msm/back/state_machine.hpp>
#include <iostream>

namespace msm = boost::msm;
using namespace boost::msm::front::euml;

BOOST_MSM_EUML_STATE((), Off)
BOOST_MSM_EUML_STATE((), On)

BOOST_MSM_EUML_EVENT(press)

BOOST_MSM_EUML_ACTION(switch_light)
{
  template <class Event, class Fsm>
  void operator()(const Event &ev, Fsm &fsm,
    BOOST_MSM_EUML_STATE_NAME(Off) &sourceState,
    BOOST_MSM_EUML_STATE_NAME(On) &targetState) const
  {
    std::cout << "Switching on\n";
  }

  template <class Event, class Fsm>
  void operator()(const Event &ev, Fsm &fsm,
    decltype(On) &sourceState,
    decltype(Off) &targetState) const
  {
    std::cout << "Switching off\n";
  }
};
```

```
BOOST_MSM_EUML_TRANSITION_TABLE((
  Off + press / switch_light == On,
  On + press / switch_light == Off
), light_transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
(light_transition_table, init_ << Off),
light_state_machine)

int main()
{
  msm::back::state_machine<light_state_machine> light;
  light.process_event(press);
  light.process_event(press);
}
```

Example 68.2 extends the state machine for the lamp by an action. An action is executed by an event triggering a state transition. Because actions are optional, a state machine could be defined without them.

Actions are defined with `BOOST_MSM_EUML_ACTION`. Strictly speaking, a function object is defined. You must overload the operator `operator()`. The operator must accept four parameters. The parameters reference an event, a state machine and two states. You are free to define a template or use concrete types for all of the parameters. In Example 68.2, concrete types are only set for the last two parameters. Because these parameters describe the beginning and ending states, you can overload `operator()` so that different member functions are executed for different switches.

Please note that the states **`On`** and **`Off`** are objects. Boost.MetaStateMachine provides a macro `BOOST_MSM_EUML_STATE_NAME` to get the type of a state. If you use C++11, you can use the operator `decltype` instead of the macro.

The action **`switch_light`**, which has been defined with `BOOST_MSM_EUML_ACTION`, is executed when the light switch is pressed. The transition table has been changed accordingly. The first transition is now `Off + press / switch_light == On`. You pass actions after a slash after the event. This transition means that the operator `operator()` of **`switch_light`** is called if the current state is **`Off`** and the event **`press`** happens. After the action has been executed, the new state is **`On`**.

Example 68.2 writes `Switching on` and then `Switching off` to standard output.

Example 68.3. State machine with state, event, guard, and action

```
#include <boost/msm/front/euml/euml.hpp>
#include <boost/msm/front/euml/state_grammar.hpp>
#include <boost/msm/back/state_machine.hpp>
#include <iostream>

namespace msm = boost::msm;
using namespace boost::msm::front::euml;

BOOST_MSM_EUML_STATE((), Off)
BOOST_MSM_EUML_STATE((), On)

BOOST_MSM_EUML_EVENT(press)
```

```
BOOST_MSM_EUML_ACTION(is_broken)
{
  template <class Event, class Fsm, class Source, class Target>
  bool operator()(const Event &ev, Fsm &fsm, Source &src, Target &trg) const
  {
    return true;
  }
};

BOOST_MSM_EUML_ACTION(switch_light)
{
  template <class Event, class Fsm, class Source, class Target>
  void operator()(const Event &ev, Fsm &fsm, Source &src, Target &trg) const
  {
    std::cout << "Switching\n";
  }
};

BOOST_MSM_EUML_TRANSITION_TABLE((
  Off + press [!is_broken] / switch_light == On,
  On + press / switch_light == Off
), light_transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
(light_transition_table, init_ << Off),
light_state_machine)

int main()
{
  msm::back::state_machine<light_state_machine> light;
  light.process_event(press);
  light.process_event(press);
}
```

Example 68.3 uses a guard in the transition table. The definition of the first transition is `Off + press [!is_broken] / switch_light == On`. Passing **`is_broken`** in brackets means that the state machine checks before the action **`switch_light`** is called whether the transition may occur. This is called a guard. A guard must return a result of type `bool`.

A guard like **`is_broken`** is defined with `BOOST_MSM_EUML_ACTION` in the same way as actions. Thus, the operator `operator()` has to be overloaded for the same four parameters. `operator()` must have a return value of type `bool` to be used as a guard.

Please note that you can use logical operators like `operator!` on guards inside brackets.

If you run the example, you'll notice that nothing is written to standard output. The action **`switch_light`** is not executed – the light stays off. The guard **`is_broken`** returns `true`. However, because the operator `operator!` is used, the expression in brackets evaluates to `false`.

You can use guards to check whether a state transition can occur. Example 68.3 uses **`is_broken`** to check whether the lamp is broken. While a transition from off to on is usually possible and the transition table describes lamps correctly, in this example, the lamp cannot be switched on. Despite two calls to `process_event()`, the state of **`light`** is **`Off`**.

Example 68.4. State machine with state, event, entry action, and exit action

```
#include <boost/msm/front/euml/euml.hpp>
#include <boost/msm/front/euml/state_grammar.hpp>
```

```
#include <boost/msm/back/state_machine.hpp>
#include <iostream>

namespace msm = boost::msm;
using namespace boost::msm::front::euml;

BOOST_MSM_EUML_ACTION(state_entry)
{
  template <class Event, class Fsm, class State>
  void operator()(const Event &ev, Fsm &fsm, State &state) const
  {
    std::cout << "Entering\n";
  }
};

BOOST_MSM_EUML_ACTION(state_exit)
{
  template <class Event, class Fsm, class State>
  void operator()(const Event &ev, Fsm &fsm, State &state) const
  {
    std::cout << "Exiting\n";
  }
};

BOOST_MSM_EUML_STATE((state_entry, state_exit), Off)
BOOST_MSM_EUML_STATE((state_entry, state_exit), On)

BOOST_MSM_EUML_EVENT(press)

BOOST_MSM_EUML_TRANSITION_TABLE((
  Off + press == On,
  On + press == Off
), light_transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
(light_transition_table, init_ << Off),
light_state_machine)

int main()
{
  msm::back::state_machine<light_state_machine> light;
  light.process_event(press);
  light.process_event(press);
}
```

In Example 68.4, the first parameter passed to `BOOST_MSM_EUML_STATE` is a tuple consisting of `state_entry` and `state_exit`. `state_entry` is an entry action, and `state_exit` is an exit action. These actions are executed when a state is entered or exited.

Like actions, entry and exit actions are defined with `BOOST_MSM_EUML_ACTION`. However, the overloaded operator `operator()` expects only three parameters: references to an event, a state machine, and a state. Transitions between states don't matter for entry and exit actions, so only one state needs to be passed to `operator()`. For entry actions, this state is entered. For exit actions, this state is exited.

In Example 68.4, both states **`Off`** and **`On`** have entry and exit actions. Because the event **`press`** occurs twice, `Entering` and `Exiting` is displayed twice. Please note that `Exiting` is displayed first and `Entering` afterwards because the first action executed is an exit action.

The first event **`press`** triggers a transition from **`Off`** to **`On`**, and `Exiting` and `Entering` are each displayed once. The second event **`press`** switches the state to **`Off`**. Again `Exiting` and

`Entering` are each displayed once. Thus, state transitions execute the exit action first, then the entry action of the new state.

Example 68.5. Attributes in a state machine

```
#include <boost/msm/front/euml/euml.hpp>
#include <boost/msm/front/euml/state_grammar.hpp>
#include <boost/msm/back/state_machine.hpp>
#include <iostream>

namespace msm = boost::msm;
using namespace boost::msm::front::euml;

BOOST_MSM_EUML_DECLARE_ATTRIBUTE(int, switched_on)

BOOST_MSM_EUML_ACTION(state_entry)
{
  template <class Event, class Fsm, class State>
  void operator()(const Event &ev, Fsm &fsm, State &state) const
  {
    std::cout << "Switched on\n";
    ++fsm.get_attribute(switched_on);
  }
};
BOOST_MSM_EUML_ACTION(is_broken)
{
  template <class Event, class Fsm, class Source, class Target>
  bool operator()(const Event &ev, Fsm &fsm, Source &src, Target &trg) const
  {
    return fsm.get_attribute(switched_on) > 1;
  }
};

BOOST_MSM_EUML_STATE((), Off)
BOOST_MSM_EUML_STATE((state_entry), On)
BOOST_MSM_EUML_EVENT(press)

BOOST_MSM_EUML_TRANSITION_TABLE((
  Off + press [!is_broken] == On,
  On + press == Off
), light_transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
(light_transition_table, init_ << Off, no_action, no_action,
attributes_ << switched_on), light_state_machine)

int main()
{
  msm::back::state_machine<light_state_machine> light;
  light.process_event(press);
  light.process_event(press);
  light.process_event(press);
  light.process_event(press);
  light.process_event(press);
}
```

Example 68.5 uses the guard **`is_broken`** to check whether a state transition from **`Off`** to **`On`** is possible. This time the return value of **`is_broken`** depends on how often the light switch has been pressed. It is possible to switch the light on two times before the lamp is broken. To count how often the light has been switched on, an attribute is used.

Attributes are variables that can be attached to objects. They let you adapt the behavior of state machines at run time. Because data such as how often the light has been switched on has to be

stored somewhere, it makes sense to store it directly in the state machine, in a state, or in an event.

Before an attribute can be used, it has to be defined. This is done with the macro `BOOST_MSM_EUML_DECLARE_ATTRIBUTE`. The first parameter passed to `BOOST_MSM_EUML_DECLARE_ATTRIBUTE` is the type, and the second is the name of the attribute. Example 68.5 defines the attribute **`switched_on`** of type `int`.

After the attribute has been defined, it must be attached to an object. The example attaches the attribute **`switched_on`** to the state machine. This is done via the fifth value in the tuple, which is passed as the first parameter to `BOOST_MSM_EUML_DECLARE_STATE_MACHINE`. With **`attributes_`**, a keyword from Boost.MetaStateMachine is used to create a lambda function. To attach the attribute **`switched_on`** to the state machine, write **`switched_on`** to **`attributes_`** as though it were a stream, using `operator<<`.

The third and fourth values in the tuples are both set to **`no_action`**. The attribute is passed as the fifth value in the tuple. The third and fourth values can be used to define entry and exit actions for the state machine. If no entry and exit actions are defined, use **`no_action`**.

After the attribute has been attached to the state machine, it can be accessed with `get_attribute()`. In Example 68.5, this member function is called in the entry action `state_entry` to increment the value of the attribute. Because `state_entry` is only linked to the state **`On`**, **`switched_on`** is only incremented when the light is switched on.

`switched_on` is also accessed from the guard **`is_broken`**, which checks whether the value of the attribute is greater than 1. If it is, the guard returns `true`. Because attributes are initialized with the default constructor and **`switched_on`** is set to 0, **`is_broken`** returns `true` if the light has been switched on two times.

In Example 68.5, the event **`press`** occurs five times. The light is switched on and off two times and then switched on again. The first two times the light is switched on, `Switched on` is displayed. However, the third time the light is switched on there is no output. This happens because **`is_broken`** returns `true` after the light has been switched on two times, and therefore, there is no state transition from **`Off`** to **`On`**. This means the entry action for the state **`On`** is not executed, and the example does not write to standard output.

Example 68.6. Accessing attributes in transition tables

```
#include <boost/msm/front/euml/euml.hpp>
#include <boost/msm/front/euml/state_grammar.hpp>
#include <boost/msm/back/state_machine.hpp>
#include <iostream>

namespace msm = boost::msm;
using namespace boost::msm::front::euml;

BOOST_MSM_EUML_DECLARE_ATTRIBUTE(int, switched_on)

void write_message()
{
  std::cout << "Switched on\n";
}
```

```
BOOST_MSM_EUML_FUNCTION(WriteMessage_, write_message, write_message_,
  void, void)

BOOST_MSM_EUML_STATE((), Off)
BOOST_MSM_EUML_STATE((), On)

BOOST_MSM_EUML_EVENT(press)

BOOST_MSM_EUML_TRANSITION_TABLE((
  Off + press [fsm_(switched_on) < Int_<2>()] / (++fsm_(switched_on),
    write_message_()) == On,
  On + press == Off
), light_transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
(light_transition_table, init_ << Off, no_action, no_action,
attributes_ << switched_on), light_state_machine)

int main()
{
  msm::back::state_machine<light_state_machine> light;
  light.process_event(press);
  light.process_event(press);
  light.process_event(press);
  light.process_event(press);
  light.process_event(press);
}
```

Example 68.6 does the same thing as Example 68.5: after switching the light on two times, the light is broken and can't be switched on anymore. While the previous example accessed the attribute **`switched_on`** in actions, this example uses attributes in the transition table.

Boost.MetaStateMachine provides the function `fsm_()` to access an attribute in a state machine. That way a guard is defined that checks whether **`switched_on`** is smaller than 2. And an action is defined that increments **`switched_on`** every time the state switches from **`Off`** to **`On`**.

Please note that the smaller-than comparison in the guard is done with `Int_<2>()`. The number 2 must be passed as a template parameter to `Int_` to create an instance of this class. That creates a function object that has the type needed by Boost.MetaStateMachine.

Example 68.6 also uses the macro `BOOST_MSM_EUML_FUNCTION` to make a function an action. The first parameter passed to `BOOST_MSM_EUML_FUNCTION` is the name of the action that can be used in the function front-end. The second parameter is the name of the function. The third parameter is the name of the action as it is used in eUML. The fourth and fifth parameters are the return values for the function – one for the case where the action is used for a state transition, and the other for the case where the action describes an entry or exit action. After `write_message()` has been turned into an action this way, an object of type `write_message_` is created and used following `++fsm_(switched_on)` in the transition table. In a state transition from **`Off`** to **`On`**, the attribute **`switched_on`** is incremented and then `write_message()` is called.

Example 68.6 displays `Switched on` twice, as in Example 68.5.

Boost.MetaStateMachine provides additional functions, such as `state_()` and `event_()`, to access attributes attached to other objects. Other classes, such as `Char_` and `String_`, can also be used like `Int_`.

Tip As you can see in the examples, the front-end eUML requires you to use many macros. The header file `boost/msm/front/euml/common.hpp` contains definitions for all of the eUML macros, which makes it a useful reference.

Exercises

1. Create a state machine for a window that can be closed, opened or tilted. A closed window can be opened or tilted. An open window can't be tilted though without closing it first. Nor can a tilted window be opened without closing it first. Test your state machine by opening and tilting your window a couple of times. Use `current_state()` to write states to standard output.

2. Extend the state machine: The window should be part of a smart home. The state machine should now count how often the window was opened and tilted. To test your state machine open and tilt your window a couple of times. At the end of your program write to standard output how often the window was opened and how often it was tilted.