Meta State Machine (MSM)

Christophe Henry

Meta State Machine (MSM)

Christophe Henry

Table of Contents

List of Tables

Preface

MSM is a library allowing you to easily and quickly define state machines of very high performance. From this point, two main questions usually quickly arise, so please allow me to try answering them upfront.

- When do I need a state machine?

  More often that you think. Very often, one defined a state machine informally without even noticing it. For example, one declares inside a class some boolean attribute, say to remember that a task has been completed. Later the boolean actually needs a third value, so it becomes an int. A few weeks, a second attribute is needed. Then a third. Soon, you find yourself writing:

  ```
  void incoming_data(data)

  {

  if (data == packet_3 && flag1 == work_done && flag2 > step3)...

  }
  ```

  This starts to look like event processing (contained inside data) if some stage of the object life has been achieved (but is ugly).

  This could be a protocol definition and it is a common use case for state machines. Another common one is a user interface. The stage of the user's interaction defines if some button is active, a functionality is available, etc.

  But there are many more use cases if you start looking. Actually, a whole model-driven development method, Executable UML (http://en.wikipedia.org/wiki/Executable_UML) specifies its complete dynamic behavior using state machines. Class diagram, state machine diagrams, and an action language are all you absolutely need in the Executable UML world.

- Another state machine library? What for?

  True, there are many state machine libraries. This should already be an indication that if you're not using any of them, you might be missing something. Why should you use this one? Unfortunately, when looking for a good state machine library, you usually pretty fast hit one or several of the following snags:

  - speed: "state machines are slow" is usually the first criticism you might hear. While it is often an excuse not to use any and instead resort to dirty, hand-written implementations (I mean, no, yours are not dirty of course, I'm talking about other developers). MSM removes this often feeble excuse because it is blazingly fast. Most hand-written implementations will be beaten by MSM.

  - ease of use: good argument. If you used another library, you are probably right. Many state machine definitions will look similar to:

    ```
    state s1 = new State; // a state

    state s2 = new State; // another state

    event e = new Event; // event

    s1->addTransition(e,s2); // transition s1 -> s2
    ```

    The more transitions you have, the less readable it is. A long time ago, there was not so much Java yet, and many electronic systems were built with a state machine defined by a simple transition table. You could easily see the whole structure and immediately see if you forgot some transitions.

Thanks to our new OO techniques, this ease of use was gone. MSM gives you back the transition table and reduces the noise to the minimum.

- expressiveness: MSM offers several front-ends and constantly tries to improve state machine definition techniques. For example, you can define a transition with eUML (one of MSM's front-ends) as:

  ```
  state1 == state2 + event [condition] / action
  ```

  This is not simply syntactic sugar. Such a formalized, readable structure allows easy communication with domain experts of a software to be constructed. Having domain experts understand your code will greatly reduce the number of bugs.

- model-driven-development: a common difficulty of a model-driven development is the complexity of making a round-trip (generating code from model and then model from code). This is due to the fact that if a state machine structure is hard for you to read, chances are that your parsing tool will also have a hard time. MSM's syntax will hopefully help tool writers.

- features: most developers use only 20% of the richly defined UML standard. Unfortunately, these are never the same 20% for all. And so, very likely, one will need something from the standard which is not implemented. MSM offers a very large part of the standard, with more on the way.

Let us not wait any longer, I hope you will enjoy MSM and have fun with it!

Part I. User' guide

Table of Contents

Chapter 1. Founding idea

Let's start with an example taken from the C++ Template Metaprogramming book:

```
class player : public state_machine<player>
{
  // The list of FSM states enum states { Empty, Open, Stopped, Playing, Paused

  // transition actions
  void start_playback(play const&) { std::cout << "player::start_playback\n"; }
  void open_drawer(open_close const&) { std::cout << "player::open_drawer\n"; }
  // more transition actions
  ...
  typedef player p; // makes transition table cleaner
  struct transition_table : mpl::vector11<
  //    Start     Event         Target      Action
  //   +---------+-------------+-----------+---------------------------+
    row< Stopped , play       ,  Playing  , &p::start_playback        >,
    row< Stopped , open_close ,  Open     , &::open_drawer            >,
  //   +---------+-------------+-----------+---------------------------+
    row< Open    , open_close ,  Empty    , &p::close_drawer          >,
  //   +---------+-------------+-----------+---------------------------+
    row< Empty   , open_close ,  Open     , &p::open_drawer           >,
    row< Empty   , cd_detected,  Stopped  , &p::store_cd_info         >,
  //   +---------+-------------+-----------+---------------------------+
    row< Playing , stop       ,  Stopped  , &p::stop_playback         >,
    row< Playing , pause      ,  Paused   , &p::pause_playback        >,
    row< Playing , open_close ,  Open     , &p::stop_and_open         >,
  //   +---------+-------------+-----------+---------------------------+
    row< Paused  , play       ,  Playing  , &p::resume_playback       >,
    row< Paused  , stop       ,  Stopped  , &p::stop_playback         >,
    row< Paused  , open_close ,  Open     , &p::stop_and_open         >
  //   +---------+-------------+-----------+---------------------------+
  > {};
  // Replaces the default no-transition response.
  template <class Event>
  int no_transition(int state, Event const& e)
  {
    std::cout << "no transition from state " << state << " on event " << typeid
    return state;
  }
};
```

This example is the foundation for the idea driving MSM: a descriptive and expressive language based on a transition table with as little syntactic noise as possible, all this while offering as many features from the UML 2.0 standard as possible. MSM also offers several expressive state machine definition syntaxes with different trade-offs.

Chapter 2. UML Short Guide

What are state machines?

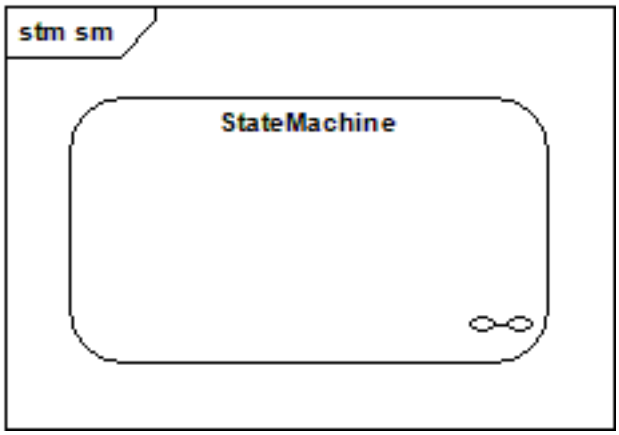
State machines are the description of a thing's lifeline. They describe the different stages of the lifeline, the events influencing it, and what it does when a particular event is detected at a particular stage. They offer the complete specification of the dynamic behavior of the thing.

Concepts

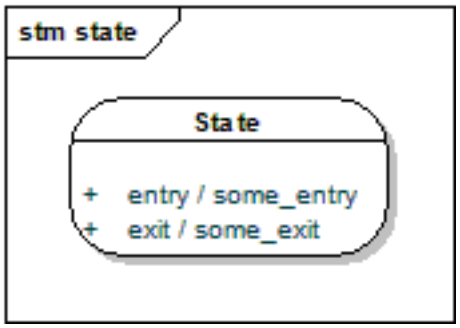
Thinking in terms of state machines is a bit surprising at first, so let us have a quick glance at the concepts.

State machine, state, transition, event

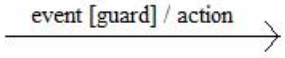
A state machine is a concrete model describing the behavior of a system. It is composed of a finite number of states and transitions.



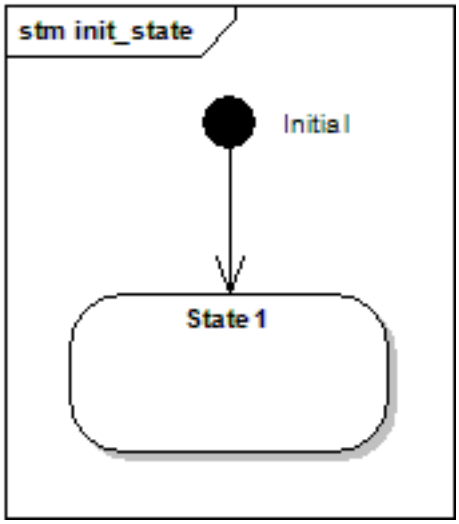
A simple state has no sub states. It can have data, entry and exit behaviors and deferred events. One can provide entry and exit behaviors (also called actions) to states (or state machines), which are executed whenever a state is entered or left, no matter how. A state can also have internal transitions which cause no entry or exit behavior to be called. A state can mark events as deferred. This means the event cannot be processed if this state is active, but it must be retained. Next time a state not deferring this event is active, the event will be processed, as if it had just been fired.



A transition is the switching between active states, triggered by an event. Actions and guard conditions can be attached to the transition. The action executes when the transition fires, the guard is a Boolean operation executed first and which can prevent the transition from firing by returning false.



An initial state marks the first active state of a state machine. It has no real existence and neither has the transition originating from it.

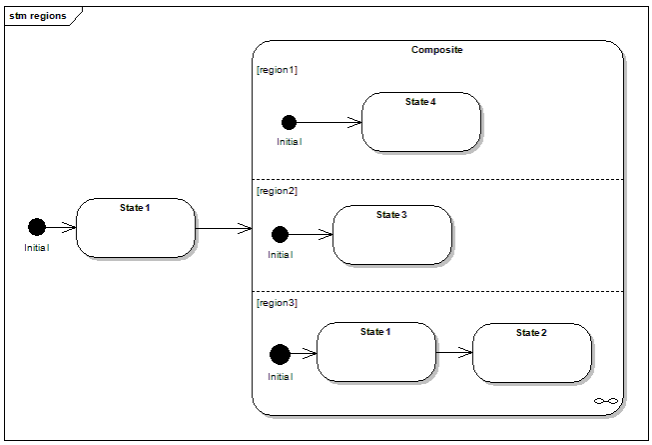


Submachines, orthogonal regions, pseudostates

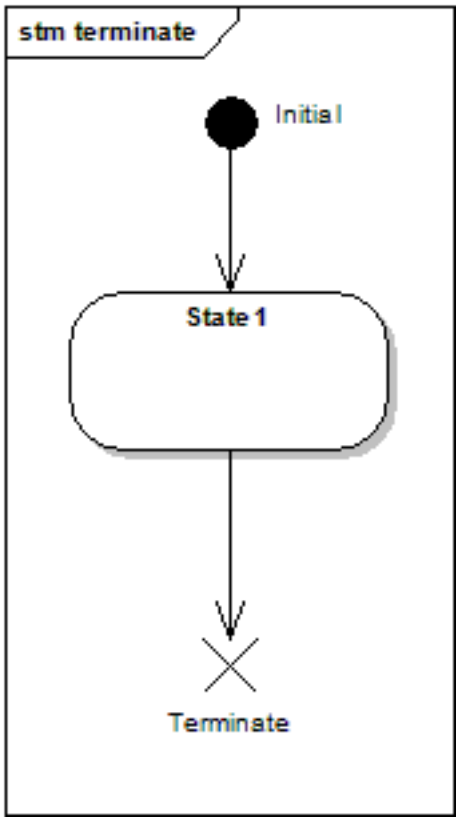
A composite state is a state containing a region or decomposed in two or more regions. A composite state contains its own set of states and regions.

A submachine is a state machine inserted as a state in another state machine. The same submachine can be inserted more than once.

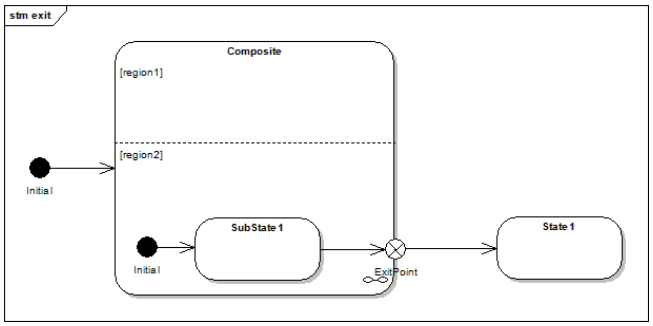
Orthogonal regions are parts of a composite state or submachine, each having its own set of mutually exclusive set of states and transitions.



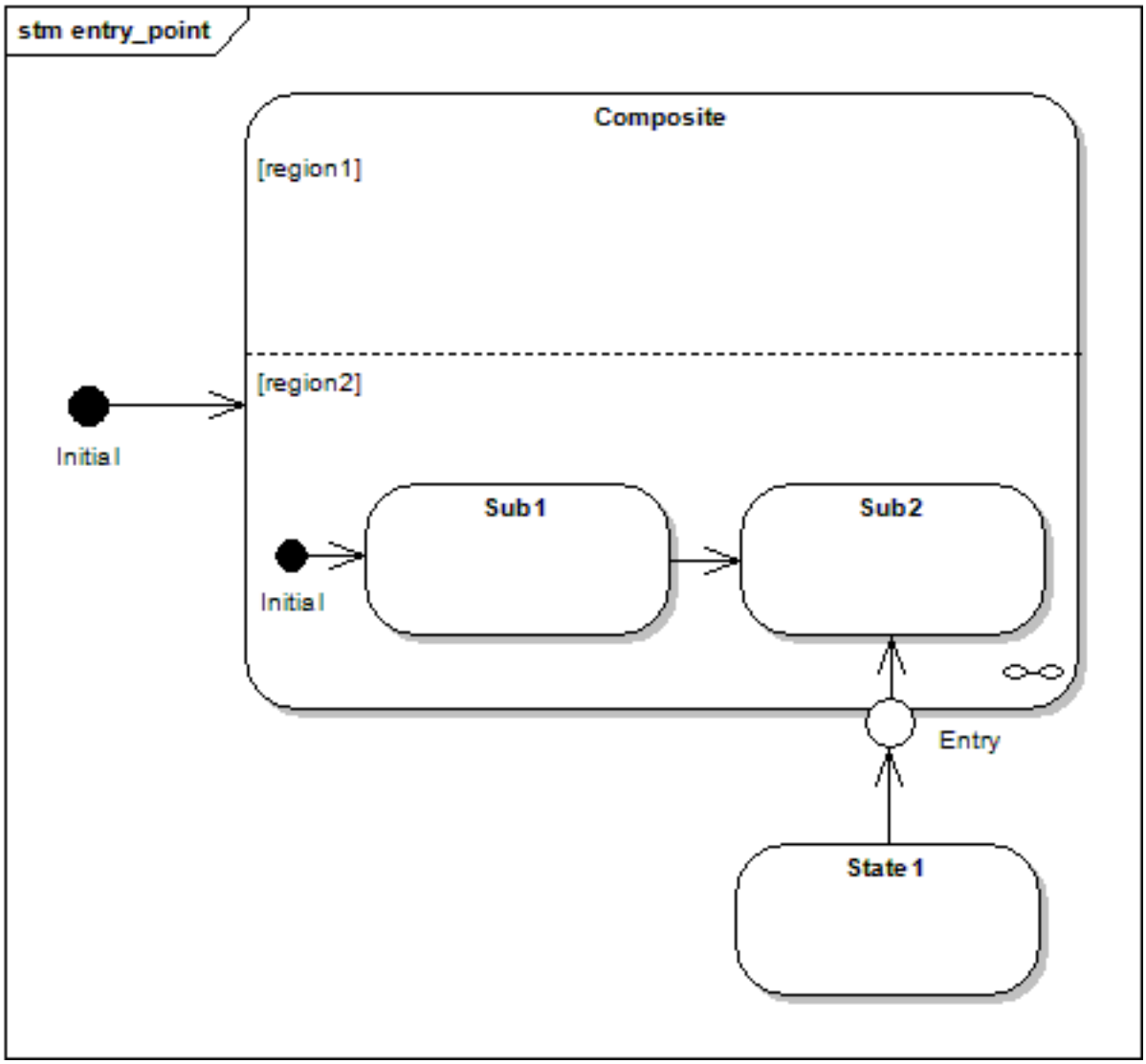
UML also defines a number of pseudo states, which are considered important concepts to model, but not enough to make them first-class citizens. The terminate pseudo state terminates the execution of a state machine (MSM handles this slightly differently. The state machine is not destroyed but no further event processing occurs.).



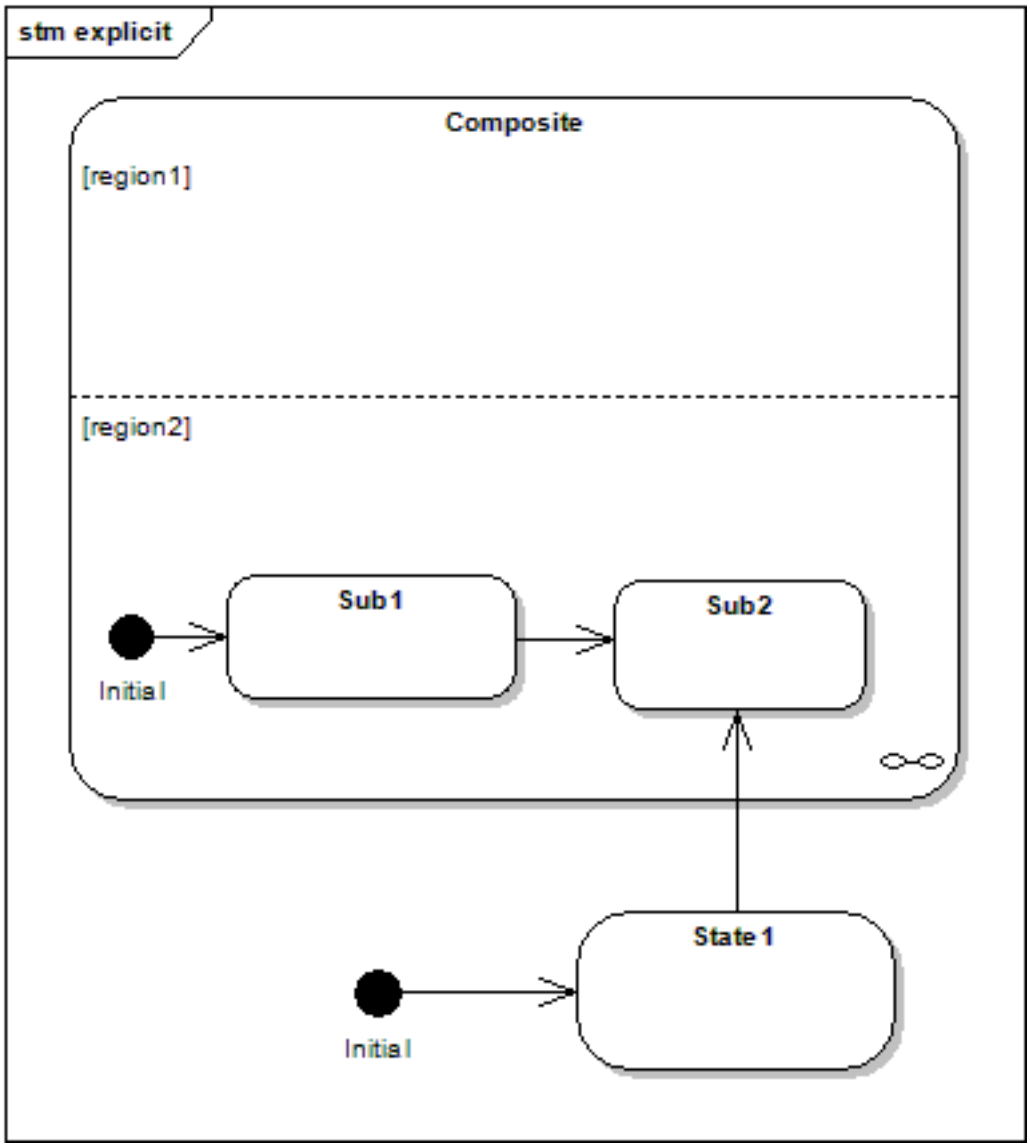
An exit point pseudo state exits a composite state or a submachine and forces termination of execution in all contained regions.



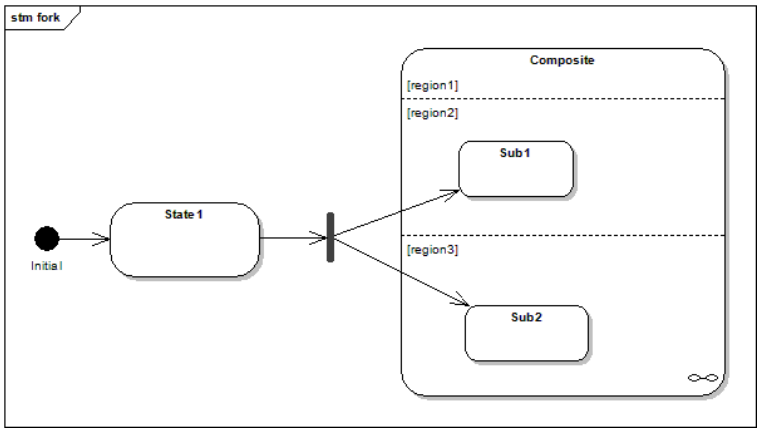
An entry point pseudo state allows a kind of controlled entry inside a composite. Precisely, it connects a transition outside the composite to a transition inside the composite. An important point is that this mechanism only allows a single region to be entered. In the above diagram, in region1, the initial state would become active.



There are also two more ways to enter a submachine (apart the obvious and more common case of a transition terminating on the submachine as shown in the region case). An explicit entry means that an inside state is the target of a transition. Unlike with direct entry, no tentative encapsulation is made, and only one transition is executed. An explicit exit is a transition from an inner state to a state outside the submachine (not supported by MSM). I would not recommend using explicit entry or exit.



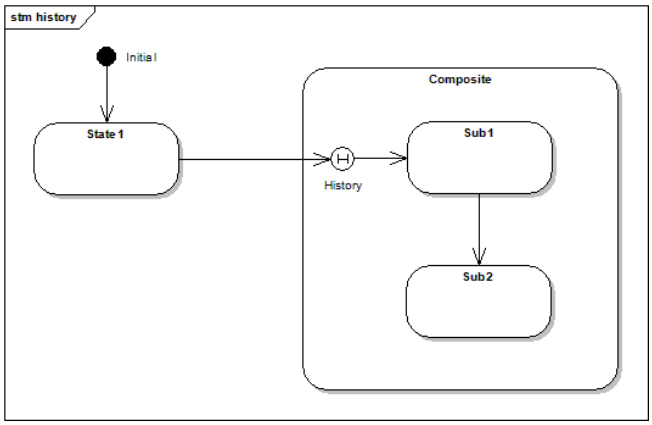
The last entry possibility is using fork. A fork is an explicit entry into one or more regions. Other regions are again activated using their initial state.



History

UML defines two kinds of history, shallow history and deep history. Shallow history is a pseudo state representing the most recent substate of a submachine. A submachine can have at most one shallow

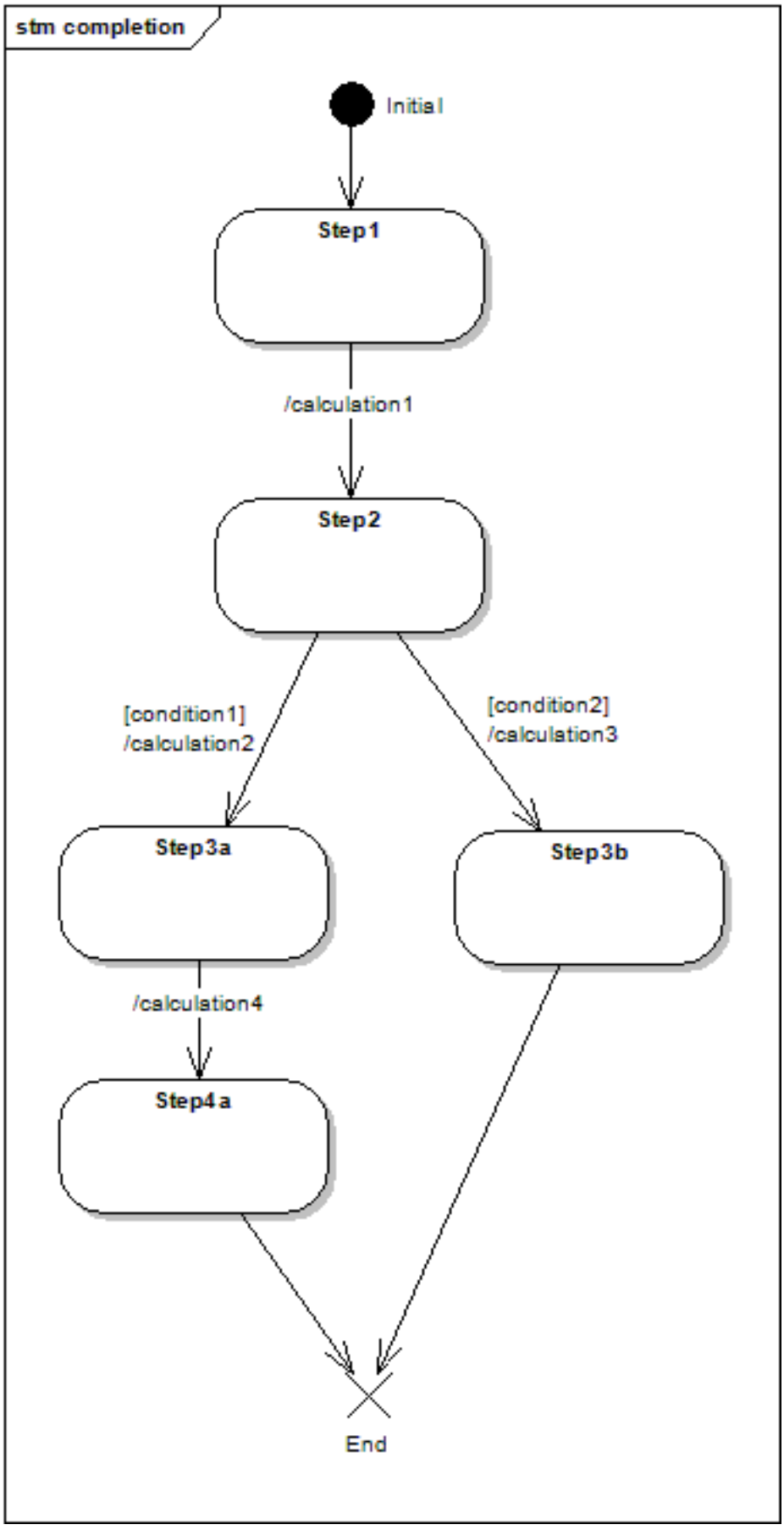
history. A transition with a history pseudo state as target is equivalent to a transition with the most recent substate as target. And very importantly, only one transition may originate from the history. Deep history is a shallow history recursively reactivating the substates of the most recent substate. It is represented like the shallow history with a star (H* inside a circle).



History is not a completely satisfying concept. First of all, there can be just one history pseudo state and only one transition may originate from it. So they do not mix well with orthogonal regions as only one region can be “remembered”. Deep history is even worse and looks like a last-minute addition. History has to be activated by a transition and only one transition originates from it, so how to model the transition originating from the deep history pseudo state and pointing to the most recent substate of the substate? As a bonus, it is also inflexible and does not accept new types of histories. Let's face it, history sounds great and is useful in theory, but the UML version is not quite making the cut. And therefore, MSM provides a different version of this useful concept.

Completion transitions / anonymous transitions

Completion events (or transitions), also called anonymous transitions, are defined as transitions having no defined event triggering them. This means that such transitions will immediately fire when a state being the source of an anonymous transition becomes active, provided that a guard allows it. They are useful in modeling algorithms as an activity diagram would normally do. In the real-time world, they have the advantage of making it easier to estimate how long a periodically executed action will last. For example, consider the following diagram.


stm completion
Initial
Step1
/calculation1
Step2
[condition1]
/calculation2
[condition2]
/calculation3
Step3a
Step3b
/calculation4
Step4a
End

The designer now knows at any time that he will need a maximum of 4 transitions. Being able to estimate how long a transition takes, he can estimate how much of a time frame he will need to require (real-time tasks are often executed at regular intervals). If he can also estimate the duration of actions, he can even use graph algorithms to better estimate his timing requirements.

Internal transitions

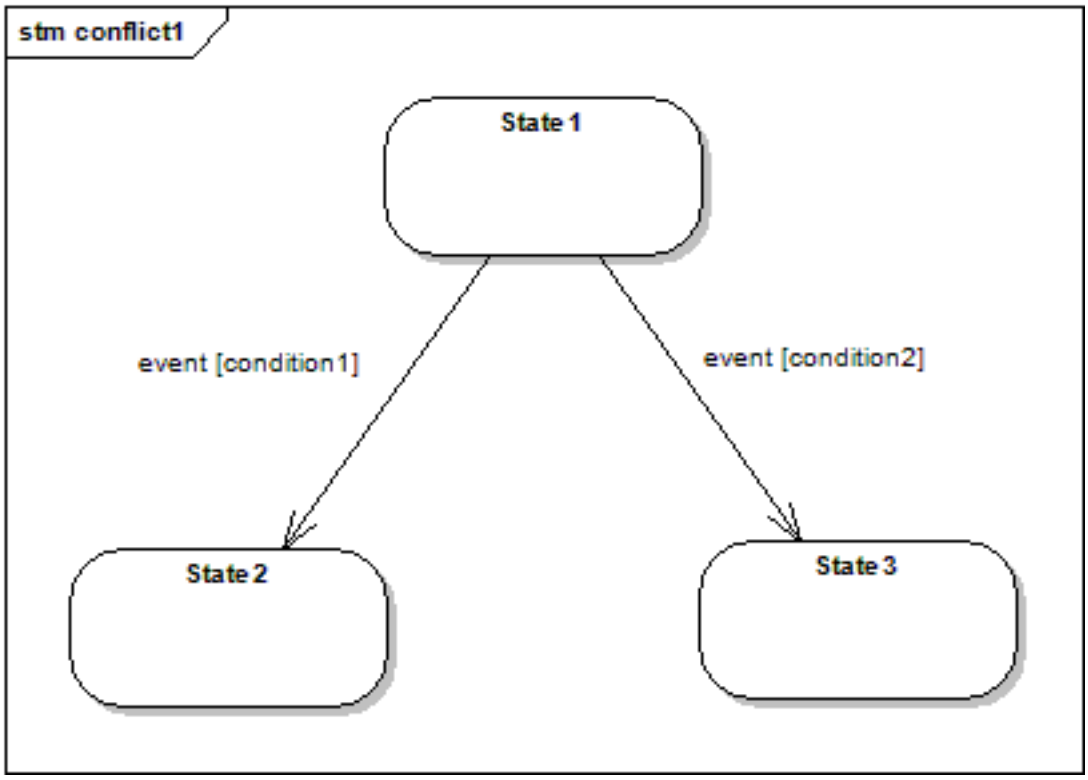
Internal transitions are transitions executing in the scope of the active state, being a simple state or a submachine. One can see them as a self-transition of this state, without an entry or exit action called.

Conflicting transitions

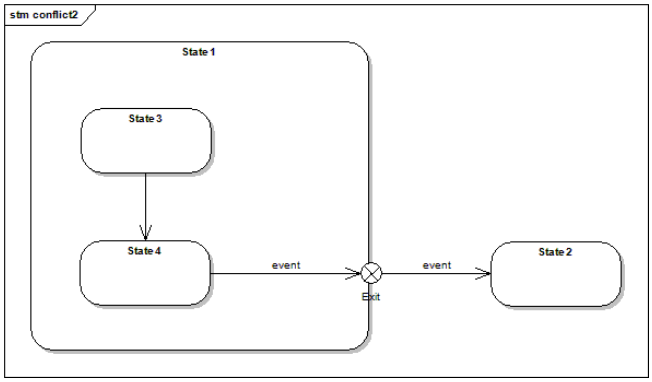
If, for a given event, several transitions are enabled, they are said to be in conflict. There are two kinds of conflicts:

- For a given source state, several transitions are defined, triggered by the same event. Normally, the guard condition in each transition defines which one is fired.
- The source state is a submachine or simple state and the conflict is between a transition internal to this state and a transition triggered by the same event and having as target another state.

The first one is simple; one only needs to define two or more rows in the transition table, with the same source and trigger, with a different guard condition. Beware, however, that the UML standard wants these conditions to be not overlapping. If they do, the standard says nothing except that this is incorrect, so the implementer is free to implement it the way he sees fit. In the case of MSM, the transition appearing last in the transition table gets selected first, if it returns false (meaning disabled), the library tries with the previous one, and so on.



In the second case, UML defines that the most inner transition gets selected first, which makes sense, otherwise no exit point pseudo state would be possible (the inner transition brings us to the exit point, from where the containing state machine can take over).



MSM handles both cases itself, so the designer needs only concentrate on its state machine and the UML subtleties (not overlapping conditions), not on implementing this behavior himself.

Added concepts

- Interrupt states: a terminate state which can be exited if a defined event is triggered.
- Kleene (any) event: a transition with a kleene event will accept any event as trigger. Unlike a completion transition, an event must be triggered and the original event is kept accessible in the kleene event.

State machine glossary

- state machine: the life cycle of a thing. It is made of states, regions, transitions and processes incoming events.
- state: a stage in the life cycle of a state machine. A state (like a submachine) can have an entry and exit behaviors.
- event: an incident provoking (or not) a reaction of the state machine
- transition: a specification of how a state machine reacts to an event. It specifies a source state, the event triggering the transition, the target state (which will become the newly active state if the transition is triggered), guard and actions.
- action: an operation executed during the triggering of the transition.
- guard: a boolean operation being able to prevent the triggering of a transition which would otherwise fire.
- transition table: representation of a state machine. A state machine diagram is a graphical, but incomplete representation of the same model. A transition table, on the other hand, is a complete representation.
- initial state: The state in which the state machine starts. Having several orthogonal regions means having as many initial states.
- submachine: A submachine is a state machine inserted as a state in another state machine and can be found several times in a same state machine.
- orthogonal regions: (logical) parallel flow of execution of a state machine. Every region of a state machine gets a chance to process an incoming event.
- terminate pseudo-state: when this state becomes active, it terminates the execution of the whole state machine. MSM does not destroy the state machine as required by the UML standard, however, which lets you keep all the state machine's data.

- entry/exit pseudo state: defined for submachines and are defined as a connection between a transition outside of the submachine and a transition inside the submachine. It is a way to enter or leave a submachine through a predefined point.
- fork: a fork allows explicit entry into several orthogonal regions of a submachine.
- history: a history is a way to remember the active state of a submachine so that the submachine can proceed in its last active state next time it becomes active.
- completion events (also called completion/anonymous transitions): when a transition has no named event triggering it, it automatically fires when the source state is active, unless a guard forbids it.
- transition conflict: a conflict is present if for a given source state and incoming event, several transitions are possible. UML specifies that guard conditions have to solve the conflict.
- internal transitions: transition from a state to itself without having exit and entry actions being called.

Chapter 3. Tutorial

Design

MSM is divided between front–ends and back-ends. At the moment, there is just one back-end. On the front-end side, you will find three of them which are as many state machine description languages, with many more possible. For potential language writers, this document contains a description of the interface between front-end and back-end.

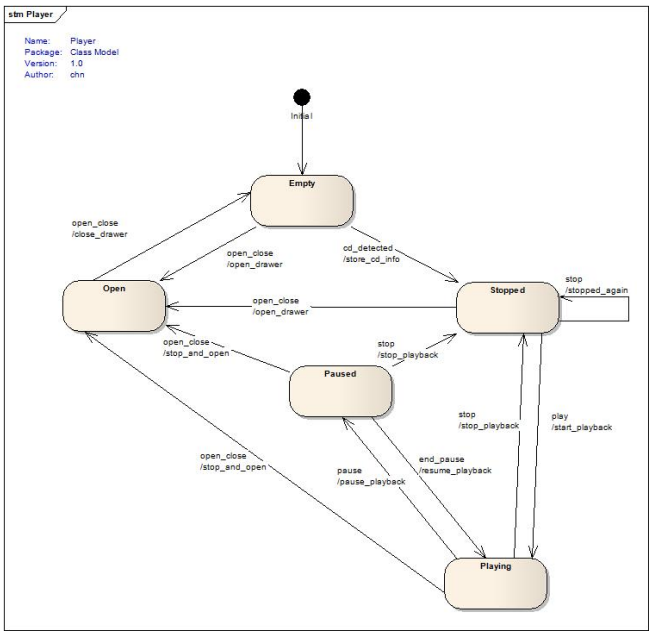
The first front-end is an adaptation of the example provided in the MPL book [http://boostpro.com/mplbook] with actions defined as pointers to state or state machine methods. The second one is based on functors. The third, eUML (embedded UML) is an experimental language based on Boost.Proto and Boost.Typeof and hiding most of the metaprogramming to increase readability. Both eUML and the functor front-end also offer a functional library (a bit like Boost.Phoenix) for use as action language (UML defining none).

Basic front-end

This is the historical front-end, inherited from the MPL book. It provides a transition table made of rows of different names and functionality. Actions and guards are defined as methods and referenced through a pointer in the transition. This front-end provides a simple interface making easy state machines easy to define, but more complex state machines a bit harder.

A simple example

Let us have a look at a state machine diagram of the founding example:



We are now going to build it with MSM's basic front-end. An implementation [examples/SimpleTutorial.cpp] is also provided.

Transition table

As previously stated, MSM is based on the transition table, so let us define one:

```
struct transition_table : mpl::vector<
//    Start     Event         Target      Action                         Guard
//   +---------+-------------+-----------+------------------------------+---------
a_row< Stopped , play       ,  Playing  , &player_::start_playback
a_row< Stopped , open_close ,  Open     , &player_::open_drawer
 _row< Stopped , stop       ,  Stopped
//   +---------+-------------+-----------+------------------------------+---------
a_row< Open    , open_close ,  Empty    , &player_::close_drawer
//   +---------+-------------+-----------+------------------------------+---------
a_row< Empty   , open_close ,  Open     , &player_::open_drawer
  row< Empty   , cd_detected,  Stopped  , &player_::store_cd_info   , &player_:
  row< Empty   , cd_detected,  Playing  , &player_::store_cd_info   , &player_:
//   +---------+-------------+-----------+------------------------------+---------
a_row< Playing , stop       ,  Stopped  , &player_::stop_playback
a_row< Playing , pause      ,  Paused   , &player_::pause_playback
a_row< Playing , open_close ,  Open     , &player_::stop_and_open
//   +---------+-------------+-----------+------------------------------+---------
a_row< Paused  , end_pause  ,  Playing  , &player_::resume_playback
a_row< Paused  , stop       ,  Stopped  , &player_::stop_playback
a_row< Paused  , open_close ,  Open     , &player_::stop_and_open
//   +---------+-------------+-----------+------------------------------+---------
> {};
```

You will notice that this is almost exactly our founding example. The only change in the transition table is the different types of transitions (rows). The founding example forces one to define an action method and offers no guards. You have 4 basic row types:

- `row` takes 5 arguments: start state, event, target state, action and guard.
- `a_row` (“a” for action) allows defining only the action and omit the guard condition.
- `g_row` (“g” for guard) allows omitting the action behavior and defining only the guard.
- `_row` allows omitting action and guard.

The signature for an action methods is void method_name (event const&), for example:

```
void stop_playback(stop const&)
```

Action methods return nothing and take the argument as const reference. Of course nothing forbids you from using the same action for several events:

```
template <class Event> void stop_playback(Eventconst&)
```

Guards have as only difference the return value, which is a boolean:

```
bool good_disk_format(cd_detected const& evt)
```

The transition table is actually a MPL vector (or list), which brings the limitation that the default maximum size of the table is 20. If you need more transitions, overriding this default behavior is necessary, so you need to add before any header:

```
#define BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS
#define BOOST_MPL_LIMIT_VECTOR_SIZE 30 //or whatever you need
#define BOOST_MPL_LIMIT_MAP_SIZE 30 //or whatever you need
```

The other limitation is that the MPL types are defined only up to 50 entries. For the moment, the only solution to achieve more is to add headers to the MPL (luckily, this is not very complicated).

Defining states with entry/exit actions

While states were enums in the MPL book, they now are classes, which allows them to hold data, provide entry, exit behaviors and be reusable (as they do not know anything about the containing state machine). To define a state, inherit from the desired state type. You will mainly use simple states:

struct Empty : public msm::front::state<> {};

They can optionally provide entry and exit behaviors:

```
struct Empty : public msm::front::state<>
{
    template <class Event, class Fsm>
    void on_entry(Event const&, Fsm& )
    {std::cout <<"entering: Empty" << std::endl;}
    template <class Event, class Fsm>
    void on_exit(Event const&, Fsm& )
    {std::cout <<"leaving: Empty" << std::endl;}
};
```

Notice how the entry and exit behaviors are templatized on the event and state machine. Being generic facilitates reuse. There are more state types (terminate, interrupt, pseudo states, etc.) corresponding to the UML standard state types. These will be described in details in the next sections.

What do you actually do inside actions / guards?

State machines define a structure and important parts of the complete behavior, but not all. For example if you need to send a rocket to Alpha Centauri, you can have a transition to a state "SendRocketToAlphaCentauri" but no code actually sending the rocket. This is where you need actions. So a simple action could be:

```
template <class Fire> void send_rocket(Fire const&)
{
  fire_rocket();
}
```

Ok, this was simple. Now, we might want to give a direction. Let us suppose this information is externally given when needed, it makes sense do use the event for this:

```
// Event
struct Fire {Direction direction;};
template <class Fire> void send_rocket(Fire const& evt)
{
  fire_rocket(evt.direction);
}
```

We might want to calculate the direction based not only on external data but also on data accumulated during previous work. In this case, you might want to have this data in the state machine itself. As transition actions are members of the front-end, you can directly access the data:

```
// Event
struct Fire {Direction direction;};
//front-end definition, see down
struct launcher_ : public msm::front::state_machine_def<launcher_>{
Data current_calculation;
```

```
template <class Fire> void send_rocket(Fire const& evt)
{
  fire_rocket(evt.direction, current_calculation);
}
...
};
```

Entry and exit actions represent a behavior common to a state, no matter through which transition it is entered or left. States being reusable, it might make sense to locate your data there instead of in the state machine, to maximize reuse and make code more readable. Entry and exit actions have access to the state data (being state members) but also to the event and state machine, like transition actions. This happens through the Event and Fsm template parameters:

```
struct Launching : public msm::front::state<>
{
    template <class Event, class Fsm>
    void on_entry(Event const& evt, Fsm& fsm)
    {
       fire_rocket(evt.direction, fsm.current_calculation);
    }
};
```

Exit actions are also ideal for clanup when the state becomes inactive.

Another possible use of the entry action is to pass data to substates / submachines. Launching is a substate containing a `data` attribute:

```
struct launcher_ : public msm::front::state_machine_def<launcher_>{
Data current_calculation;
// state machines also have entry/exit actions
template <class Event, class Fsm>
void on_entry(Event const& evt, Fsm& fsm)
{
   launcher_::Launching& s = fsm.get_state<launcher_::Launching&>();
   s.data = fsm.current_calculation;
}
...
};
```

The **set_states** back-end method allows you to replace a complete state.

The **functor** front-end and eUML offer more capabilities.

However, this basic front-end also has special capabilities using the row2 / irow2 transitions.**_row2, a_row2, row2, g_row2, a_irow2, irow2, g_irow2** let you call an action located in any state of the current fsm or in the front-end itself, thus letting you place useful data anywhere you see fit.

It is sometimes desirable to generate new events for the state machine inside actions. Since the process_event method belongs to the back end, you first need to gain a reference to it. The back end derives from the front end, so one way of doing this is to use a cast:

```
struct launcher_ : public msm::front::state_machine_def<launcher_>{
template <class Fire> void send_rocket(Fire const& evt)
{
  fire_rocket();
  msm::back::state_machine<launcher_> &fsm = static_cast<msm::back::state_machi
  fsm.process_event(rocket_launched());
}
...
```

```
};
```

The same can be implemented inside entry/exit actions. Admittedly, this is a bit awkward. A more natural mechanism is available using the **functor** front-end.

Defining a simple state machine

Declaring a state machine is straightforward and is done with a high signal / noise ratio. In our player example, we declare the state machine as:

```
struct player_ : public msm::front::state_machine_def<player_>{
                              /* see below */}
```

This declares a state machine using the basic front-end. We now declare inside the state machine structure the initial state:

```
typedef Empty initial_state;
```

And that is about all of what is absolutely needed. In the example, the states are declared inside the state machine for readability but this is not a requirements, states can be declared wherever you like.

All what is left to do is to pick a back-end (which is quite simple as there is only one at the moment):

```
typedef msm::back::state_machine<player_> player;
```

You now have a ready-to-use state machine with entry/exit actions, guards, transition actions, a message queue so that processing an event can generate another event. The state machine also adapted itself to your need and removed almost all features we didn't use in this simple example. Note that this is not per default the fastest possible state machine. See the section "getting more speed" to know how to get the maximum speed. In a nutshell, MSM cannot know about your usage of some features so you will have to explicitly tell it.

State objects are built automatically with the state machine. They will exist until state machine destruction. MSM is using Boost.Fusion behind the hood. This unfortunately means that if you define more than 10 states, you will need to extend the default:

```
#define FUSION_MAX_VECTOR_SIZE 20 // or whatever you need
```

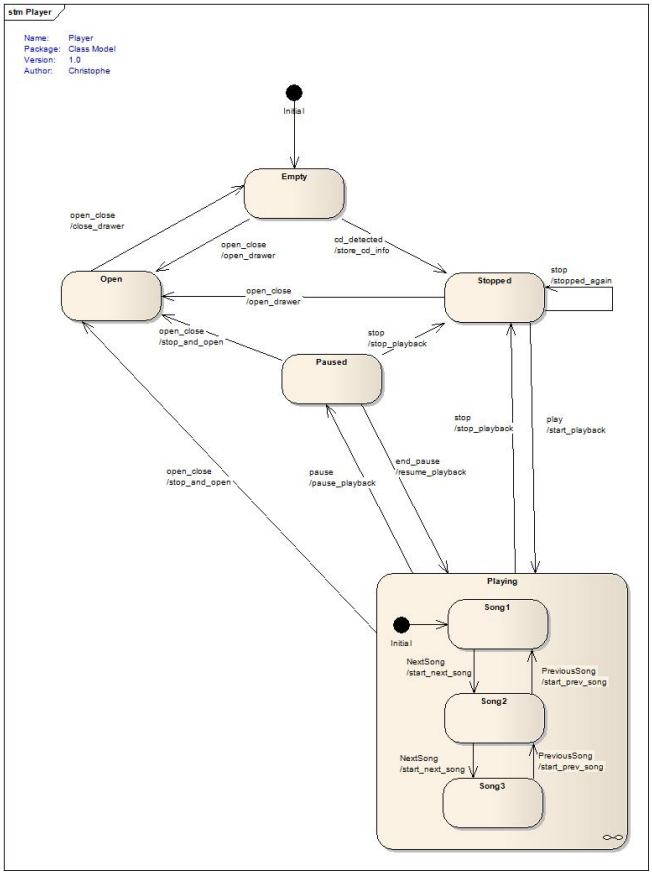
When an unexpected event is fired, the `no_transition(event, state machine, state id)` method of the state machine is called . By default, this method simply asserts when called. It is possible to overwrite the `no_transition` method to define a different handling:

```
template <class Fsm,class Event>
void no_transition(Event const& e, Fsm& ,int state){...}
```

Note: you might have noticed that the tutorial calls `start()` on the state machine just after creation. The start method will initiate the state machine, meaning it will activate the initial state, which means in turn that the initial state's entry behavior will be called. The reason why we need this will be explained in the back-end part. After a call to start, the state machine is ready to process events. The same way, calling `stop()` will cause the last exit actions to be called.

Defining a submachine

We now want to extend our last state machine by making the Playing state a state machine itself (a submachine).



Again, an example [examples/CompositeTutorial.cpp] is also provided.

A submachine really is a state machine itself, so we declare Playing as such, choosing a front-end and a back-end:

```
struct Playing_ : public msm::front::state_machine_def<Playing_>{...}
typedef msm::back::state_machine<Playing_> Playing;
```

Like for any state machine, one also needs a transition table and an initial state:

```
struct transition_table : mpl::vector<
//    Start    Event    Target    Action                         Guard
//   +--------+---------+--------+----------------------------+------+
a_row< Song1  , NextSong, Song2  , &Playing_::start_next_song        >,
a_row< Song2  , NextSong, Song1  , &Playing_::start_prev_song        >,
a_row< Song2  , NextSong, Song3  , &Playing_::start_next_song        >,
a_row< Song3  , NextSong, Song2  , &Playing_::start_prev_song        >
//   +--------+---------+--------+----------------------------+------+
> {};


typedef Song1 initial_state;
```

This is about all you need to do. MSM will now automatically recognize Playing as a submachine and all events handled by Playing (NextSong and PreviousSong) will now be automatically forwarded to Playing whenever this state is active. All other state machine features described later are also available. You can even decide to use a state machine sometimes as submachine or sometimes as an independent state machine.

There is, however, a limitation for submachines. If a submachine's substate has an entry action which requires a special event property (like a given method), the compiler will require all events

entering this submachine to support this property. As this is not practicable, we will need to use `boost::enable_if` / `boost::disable_if` to help, for example consider:

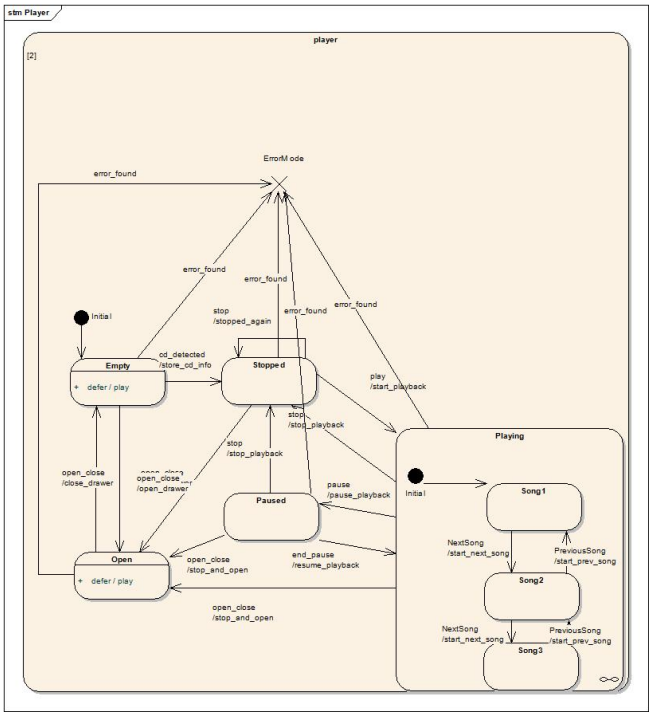
```
// define a property for use with enable_if
BOOST_MPL_HAS_XXX_TRAIT_DEF(some_event_property)

// this event supports some_event_property and a corresponding required method
struct event1
{
   // the property
   typedef int some_event_property;
   // the method required by this property
   void some_property(){...}
};
// this event does not supports some_event_property
struct event2
{
};
struct some_state : public msm::front::state<>
{
   template <class Event,class Fsm>
   // enable this version for events supporting some_event_property
   typename boost::enable_if<typename has_some_event_property<Event>::type,void
   on_entry(Event const& evt,Fsm& fsm)
   {
      evt.some_property();
   }
   // for events not supporting some_event_property
   template <class Event,class Fsm>
   typename boost::disable_if<typename has_some_event_property<Event>::type,voi
   on_entry(Event const& ,Fsm& )
   {    }
};
```

Now this state can be used in your submachine.

Orthogonal regions, terminate state, event deferring

It is a very common problem in many state machines to have to handle errors. It usually involves defining a transition from all the states to a special error state. Translation: not fun. It is also not practical to find from which state the error originated. The following diagram shows an example of what clearly becomes not very readable:

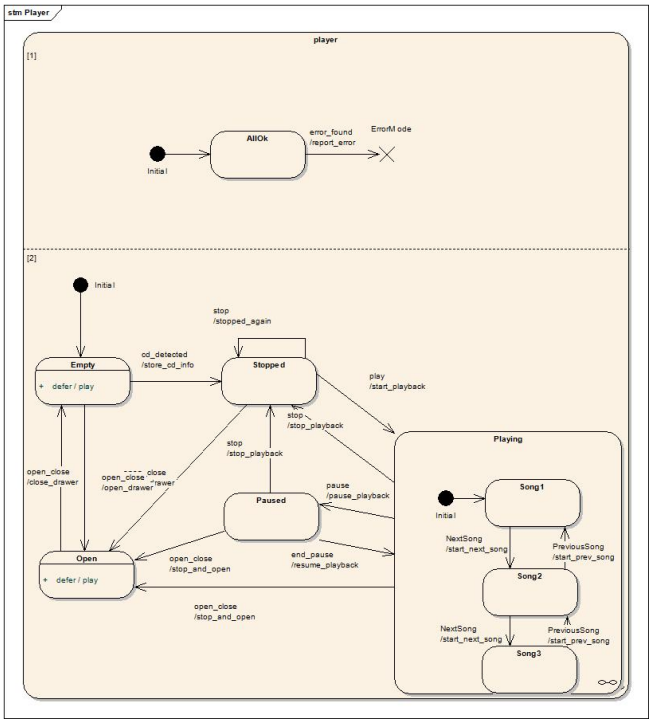


This is neither very readable nor beautiful. And we do not even have any action on the transitions yet to make it even less readable.

Luckily, UML provides a helpful concept, orthogonal regions. See them as lightweight state machines running at the same time inside a common state machine and having the capability to influence one another. The effect is that you have several active states at any time. We can therefore keep our state machine from the previous example and just define a new region made of two states, AllOk and ErrorMode. AllOk is most of the time active. But the error_found error event makes the second region move to the new active state ErrorMode. This event does not interest the main region so it will simply be ignored. "`no_transition`" will be called only if no region at all handles the event. Also, as UML mandates, every region gets a chance of handling the event, in the order as declared by the `initial_state` type.

Adding an orthogonal region is easy, one only needs to declare more states in the `initial_state` typedef. So, adding a new region with AllOk as the region's initial state is:

```
typedef mpl::vector<Empty,AllOk> initial_state;
```



Furthermore, when you detect an error, you usually do not want events to be further processed. To achieve this, we use another UML feature, terminate states. When any region moves to a terminate state, the state machine “terminates” (the state machine and all its states stay alive) and all events are ignored. This is of course not mandatory, one can use orthogonal regions without terminate states. MSM also provides a small extension to UML, interrupt states. If you declare ErrorMode (or a Boost.MPL sequence of events, like boost::mpl::vector<ErrorMode, AnotherEvent>) as interrupt state instead of terminate state, the state machine will not handle any event other than the one which ends the interrupt. So it's like a terminate state, with the difference that you are allowed to resume the state machine when a condition (like handling of the original error) is met.

Last but not least, this example also shows here the handling of event deferring. Let's say someone puts a disc and immediately presses play. The event cannot be handled, yet you'd want it to be handled at a later point and not force the user to press play again. The solution is to define it as deferred in the Empty and Open states and get it handled in the first state where the event is not to be deferred. It can then be handled or rejected. In this example, when Stopped becomes active, the event will be handled because only Empty and Open defer the event.

UML defines event deferring as a state property. To accommodate this, MSM lets you specify this in states by providing a `deferred_events` type:

```
struct Empty : public msm::front::state<>
{
   // if the play event is fired while in this state, defer it until a state
   // handles or rejects it
   typedef mpl::vector<play> deferred_events;
...
};
```

Please have a look at the complete example [examples/Orthogonal-deferred.cpp].

While this is wanted by UML and is simple, it is not always practical because one could wish to defer only in certain conditions. One could also want to make this be part of a transition action with the added bonus of a guard for more sophisticated behaviors. It would also be conform to the MSM philosophy to get as much as possible in the transition table, where you have the whole state machine structure. This is also possible but not practical with this front-end so we will need to pick a different

row from the functor front-end. For a complete description of the `Row` type, please have a look at the **functor front-end.**

First, as there is no state where MSM can automatically find out the usage of this feature, we need to require deferred events capability explicitly, by adding a type in the state machine definition:

```
struct player_ : public msm::front::state_machine_def<player_>
{
    typedef int activate_deferred_events;
...
};
```

We can now defer an event in any transition of the transition table by using as action the predefined `msm::front::Defer` functor, for example:

```
Row < Empty , play , none , Defer , none >
```

This is an internal transition row(see **internal transitions**) but you can ignore this for the moment. It just means that we are not leaving the Empty state. What matters is that we use Defer as action. This is roughly equivalent to the previous syntax but has the advantage of giving you all the information in the transition table with the added power of transition behavior.

The second difference is that as we now have a transition defined, this transition can play in the resolution of **transition conflicts**. For example, we could model an "if (condition2) move to Playing else if (condition1) defer play event":

```
Row   < Empty , play , none    , Defer , condition1   >,
g_row < Empty , play , Playing , &player_::condition2 >
```

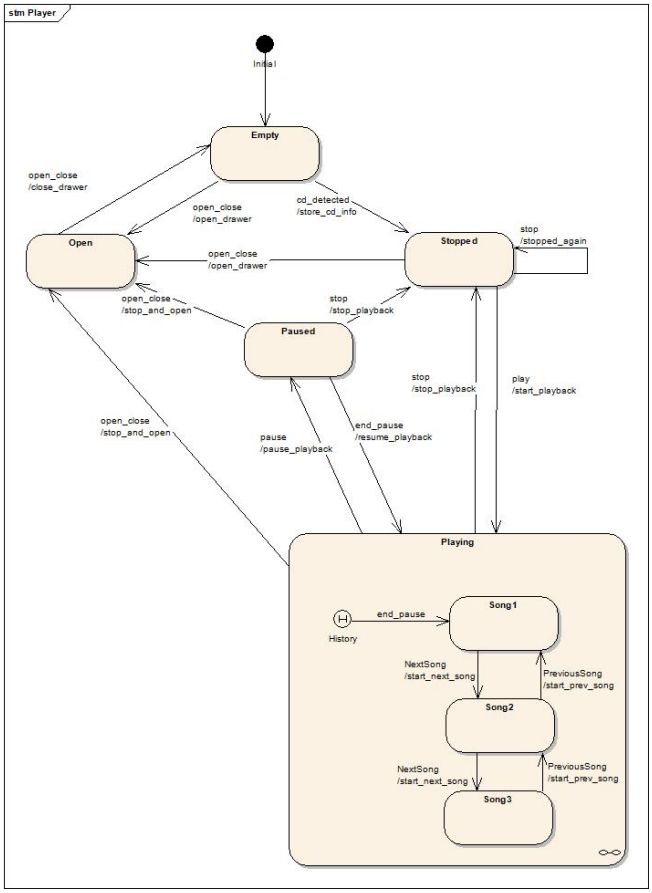
Please have a look at this possible implementation [examples/Orthogonal-deferred2.cpp].

History

UML defines two types of history, Shallow History and Deep History. In the previous examples, if the player was playing the second song and the user pressed pause, leaving Playing, at the next press on the play button, the Playing state would become active and the first song would play again. Soon would the first client complaints follow. They'd of course demand, that if the player was paused, then it should remember which song was playing. But it the player was stopped, then it should restart from the first song. How can it be done? Of course, you could add a bit of programming logic and generate extra events to make the second song start if coming from Pause. Something like:

```
if (Event == end_pause)
{
   for (int i=0;i< song number;++i) {player.process_event(NextSong()); }
}
```

Not much to like in this example, isn't it? To solve this problem, you define what is called a shallow or a deep history. A shallow history reactivates the last active substate of a submachine when this submachine becomes active again. The deep history does the same recursively, so if this last active substate of the submachine was itself a submachine, its last active substate would become active and this will continue recursively until an active state is a normal state. For example, let us have a look at the following UML diagram:



Notice that the main difference compared to previous diagrams is that the initial state is gone and replaced by a History symbol (the H inside a circle).

As explained in the **small UML tutorial**, History is a good concept with a not completely satisfying specification. MSM kept the concept but not the specification and goes another way by making this a policy and you can add your own history types (the reference explains what needs to be done). Furthermore, History is a backend policy. This allows you to reuse the same state machine definition with different history policies in different contexts.

Concretely, your frontend stays unchanged:

```
struct Playing_ : public msm::front::state_machine_def<Playing_>
```

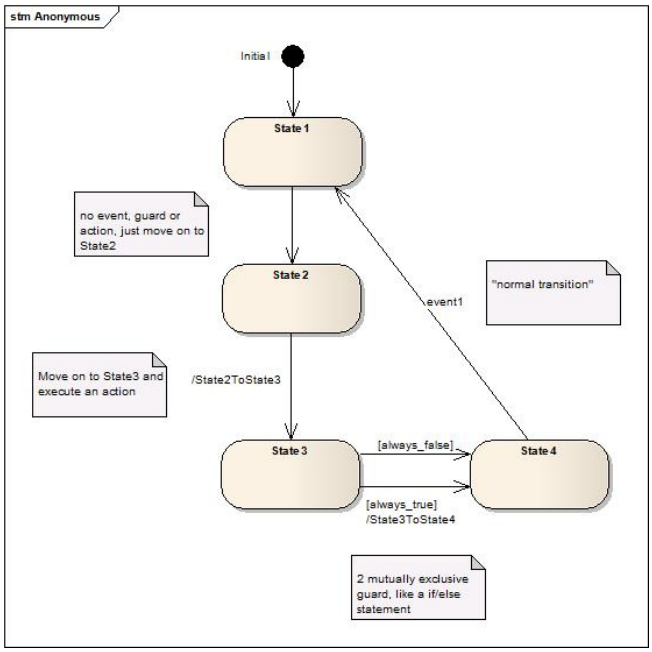
You then add the policy to the backend as second parameter:

```
typedef msm::back::state_machine<Playing_,
    msm::back::ShallowHistory<mpl::vector<end_pause> > > Playing;
```

This states that a shallow history must be activated if the Playing state machine gets activated by the end_pause event and only this one (or any other event added to the mpl::vector). If the state machine was in the Stopped state and the event play was generated, the history would not be activated and the normal initial state would become active. By default, history is disabled. For your convenience the library provides in addition to ShallowHistory a non-UML standard AlwaysHistory policy (likely to be your main choice) which always activates history, whatever event triggers the submachine activation. Deep history is not available as a policy (but could be added). The reason is that it would conflict with policies which submachines could define. Of course, if for example, Song1 were a state machine itself, it could use the ShallowHistory policy itself thus creating Deep History for itself. An example [examples/History.cpp] is also provided.

Completion (anonymous) transitions

The following diagram shows an example making use of this feature:



Anonymous transitions are transitions without a named event. This means that the transition automatically fires when the predecessor state is entered (to be exact, after the entry action). Otherwise it is a normal transition with actions and guards. Why would you need something like that? A possible case would be if a part of your state machine implements some algorithm, where states are steps of the algorithm implementation. Then, using several anonymous transitions with different guard conditions, you are actually implementing some if/else statement. Another possible use would be a real-time system called at regular intervals and always doing the same thing, meaning implementing the same algorithm. The advantage is that once you know how long a transition takes to execute on the system, by calculating the longest path (the number of transitions from start to end), you can pretty much know how long your algorithm will take in the worst case, which in turns tells you how much of a time frame you are to request from a scheduler.

If you are using Executable UML (a good book describing it is "Executable UML, a foundation for Model-Driven Architecture"), you will notice that it is common for a state machine to generate an event to itself only to force leaving a state. Anonymous transitions free you from this constraint.

If you do not use this feature in a concrete state machine, MSM will deactivate it and you will not pay for it. If you use it, there is however a small performance penalty as MSM will try to fire a compound event (the other UML name for anonymous transitions) after every taken transition. This will therefore double the event processing cost, which is not as bad as it sounds as MSM’s execution speed is very high anyway.

To define such a transition, use “none” as event in the transition table, for example:

```
row < State3 , none , State4 , &p::State3ToState4 , &p::always_true >
```

An implementation [examples/AnonymousTutorial.cpp] of the state machine diagram is also provided.

Internal transitions

Internal transitions are transitions executing in the scope of the active state, a simple state or a submachine. One can see them as a self-transition of this state, without an entry or exit action called. This is useful when all you want is to execute some code for a given event in a given state.

Internal transitions are specified as having a higher priority than normal transitions. While it makes sense for a submachine with exit points, it is surprising for a simple state. MSM lets you define the transition priority by setting the transition's position inside the transition table (see **internals**). The difference between "normal" and internal transitions is that internal transitions have no target state, therefore we need new row types. We had a_row, g_row, _row and row, we now add a_irow, g_irow, _irow and irow which are like normal transitions but define no target state. For, example an internal transition with a guard condition could be:

```
g_irow < Empty /*state*/,cd_detected/*event*/,&p::internal_guard/* guard */>
```

These new row types can be placed anywhere in the transition table so that you can still have your state machine structure grouped together. The only difference of behavior with the UML standard is the missing notion of higher priority for internal transitions. Please have a look at the example [examples/SimpleTutorialInternal.cpp].

It is also possible to do it the UML-conform way by declaring a transition table called `internal transition_table` inside the state itself and using internal row types. For example:

```
struct Empty : public msm::front::state<>
{
    struct internal_transition_table : mpl::vector<
           a_internal < cd_detected , Empty, &Empty::internal_action >
    > {};
};
```

This declares an internal transition table called internal_transition_table and reacting on the event cd_detected by calling internal_action on Empty. Let us note a few points:

- internal tables are NOT called transition_table but internal_transition_table
- they use different but similar row types: a_internal, g_internal, _internal and internal.
- These types take as first template argument the triggering event and then the action and guard method. Note that the only real difference to classical rows is the extra argument before the function pointer. This is the type on which the function will be called.
- This also allows you, if you wish, to use actions and guards from another state of the state machine or in the state machine itself.
- submachines can have an internal transition table and a classical transition table.

The following example [examples/TestInternal.cpp] makes use of an a_internal. It also uses functor-based internal transitions which will be explained in **the functor front-end**, please ignore them for the moment. Also note that the state-defined internal transitions, having the highest priority (as mandated by the UML standard), are tried before those defined inside the state machine transition table.

Which method should you use? It depends on what you need:

- the first version (using irow) is simpler and likely to compile faster. It also lets you choose the priority of your internal transition.
- the second version is more logical from a UML perspective and lets you make states more useful and reusable. It also allows you to call actions and guards on any state of the state machine.

Note: There is an added possibility coming from this feature. The `internal_transition_table` transitions being added directly inside the main state machine's transition table, it is possible, if it is more to your state, to distribute your state machine definition a bit like Boost.Statechart, leaving to the state machine itself the only task of declaring the states it wants to use using the `explicit_creation` type definition. While this is not the author's favorite way, it is still possible. A simplified example using only two states will show this possibility:

- state machine definition [examples/distributed_table/DistributedTable.cpp]
- Empty header [examples/distributed_table/Empty.hpp] and cpp [examples/distributed_table/Empty.cpp]
- Open header [examples/distributed_table/Open.hpp] and cpp [examples/distributed_table/Open.cpp]
- events definition [examples/distributed_table/Events.hpp]

There is an added bonus offered for submachines, which can have both the standard transition_table and an internal_transition_table (which has a higher priority). This makes it easier if you decide to make a full submachine from a state. It is also slightly faster than the standard alternative, adding orthogonal regions, because event dispatching will, if accepted by the internal table, not continue to the subregions. This gives you a O(1) dispatch instead of O(number of regions). While the example is with eUML, the same is also possible with any front-end.

more row types

It is also possible to write transitions using actions and guards not just from the state machine but also from its contained states. In this case, one must specify not just a method pointer but also the object on which to call it. This transition row is called, not very originally, `row2`. They come, like normal transitions in four flavors: `a_row2, g_row2, _row2 and row2`. For example, a transition calling an action from the state Empty could be:

```
a_row2<Stopped,open_close,Open,Empty
      /*action source*/,&Empty::open_drawer/*action*/>
```

The same capabilities are also available for internal transitions so that we have: `a_irow2, g_irow2, _irow2 and row2`. For transitions defined as part of the `internal_transition_table`, you can use the **a_internal, g_internal, _internal, internal** row types from the previous sections.

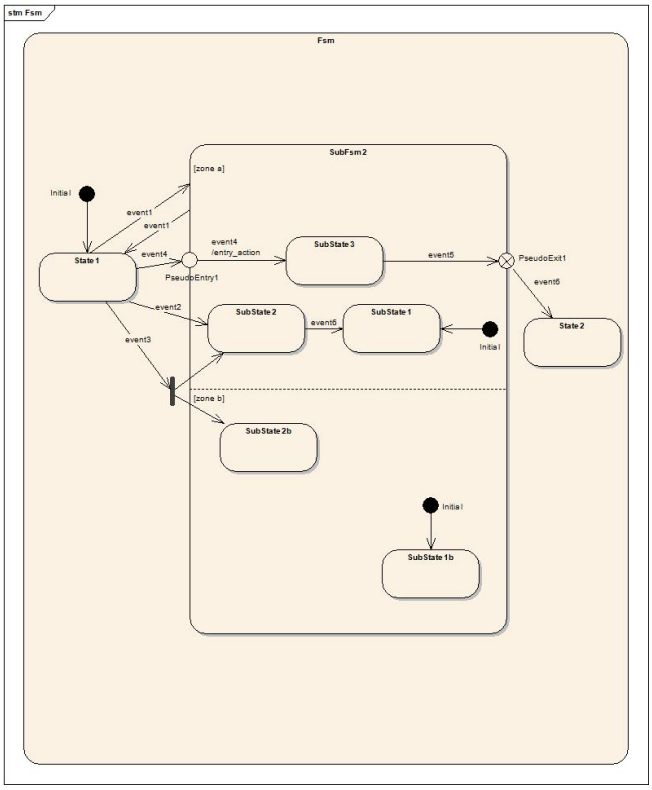
These row types allow us to distribute the state machine code among states, making them reusable and more useful. Using transition tables inside states also contributes to this possibility. An example [examples/SimpleTutorial2.cpp] of these new rows is also provided.

Explicit entry / entry and exit pseudo-state / fork

MSM (almost) fully supports these features, described in the **small UML tutorial**. Almost because there are currently two limitations:

- it is only possible to explicitly enter a sub- state of the target but not a sub-sub state.
- it is not possible to explicitly exit. Exit points must be used.

Let us see a concrete example:



We find in this diagram:

- A “normal” activation of SubFsm2, triggered by event1. In each region, the initial state is activated, i.e. SubState1 and SubState1b.
- An explicit entry into SubFsm2::SubState2 for region “1” with event2 as trigger, meaning that in region “2” the initial state, SubState1b, activated.
- A fork into regions “1” and “2” to the explicit entries SubState2 and SubState2b, triggered by event3. Both states become active so no region is default activated (if we had a third one, it would be).
- A connection of two transitions through an entry pseudo state, SubFsm2::PseudoEntry1, triggered by event4 and triggering also the second transition on the same event (both transitions must be triggered by the same event). Region “2” is default-activated and SubState1b becomes active.
- An exit from SubFsm2 using an exit pseudo-state, PseudoExit1, triggered by event5 and connecting two transitions using the same event. Again, the event is forwarded to the second transition and both regions are exited, as SubFsm2 becomes inactive. Note that if no transition is defined from PseudoExit1, an error (as defined in the UML standard) will be detected and no_transition called.

The example is also fully implemented [examples/DirectEntryTutorial.cpp].

This sounds complicated but the syntax is simple.

Explicit entry

First, to define that a state is an explicit entry, you have to make it a state and mark it as explicit, giving as template parameters the region id (the region id starts with 0 and corresponds to the first initial state of the initial_state type sequence).

```
struct SubFsm2_ : public msm::front::state_machine_def<SubFsm2_>
{
   struct SubState2 : public msm::front::state<> ,
```

```
                        public msm::front::explicit_entry<0>
   {...};
...
};
```

And define the submachine as:

```
typedef msm::back::state_machine<SubFsm2_> SubFsm2;
```

You can then use it as target in a transition with State1 as source:

```
_row < State1, Event2, SubFsm2::direct< SubFsm2_::SubState2> > //SubFsm2_::SubS
```

The syntax deserves some explanation. SubFsm2_ is a front end. SubState2 is a nested state, therefore the SubFsm2_::SubState2 syntax. The containing machine (containing State1 and SubFsm2) refers to the backend instance (SubFsm2). SubFsm2::direct states that an explicit entry is desired.

Thanks to the **mpl_graph** library you can also omit to provide the region index and let MSM find out for you. The are however two points to note:

- MSM can only find out the region index if the explicit entry state is somehow connected to an initial state through a transition, no matter the direction.
- There is a compile-time cost for this feature.

Note (also valid for forks): in order to make compile time more bearable for the more standard cases, and unlike initial states, explicit entry states which are also not found in the transition table of the entered submachine (a rare case) do NOT get automatically created. To explicitly create such states, you need to add in the state machine containing the explicit states a simple typedef giving a sequence of states to be explicitly created like:

```
typedef mpl::vector<SubState2,SubState2b> explicit_creation;
```

Note (also valid for forks): At the moment, it is not possible to use a submachine as the target of an explicit entry. Please use entry pseudo states for an almost identical effect.

Fork

Need a fork instead of an explicit entry? As a fork is an explicit entry into states of different regions, we do not change the state definition compared to the explicit entry and specify as target a list of explicit entry states:

```
_row < State1, Event3,
        mpl::vector<SubFsm2::direct<SubFsm2_::SubState2>,
        SubFsm2::direct <SubFsm2_::SubState2b>
     >
```

With SubState2 defined as before and SubState2b defined as being in the second region (Caution: MSM does not check that the region is correct):

```
struct SubState2b : public msm::front::state<> ,
                    public msm::front::explicit_entry<1>
```

Entry pseudo states

To define an entry pseudo state, you need derive from the corresponding class and give the region id:

```
struct PseudoEntry1 : public msm::front::entry_pseudo_state<0>
```

And add the corresponding transition in the top-level state machine's transition table:

```
_row < State1, Event4, SubFsm2::entry_pt<SubFsm2_::PseudoEntry1> >
```

And another in the SubFsm2_ submachine definition (remember that UML defines an entry point as a connection between two transitions), for example this time with an action method:

```
_row < PseudoEntry1, Event4, SubState3,&SubFsm2_::entry_action >
```

Exit pseudo states

And finally, exit pseudo states are to be used almost the same way, but defined differently: it takes as template argument the event to be forwarded (no region id is necessary):

```
struct PseudoExit1 : public exit_pseudo_state<event6>
```

And you need, like for entry pseudo states, two transitions, one in the submachine:

```
_row < SubState3, Event5, PseudoExit1 >
```

And one in the containing state machine:

```
_row < SubFsm2::exit_pt<SubFsm2_::PseudoExit1>, Event6,State2 >
```

Important note 1: UML defines transiting to an entry pseudo state and having either no second transition or one with a guard as an error but defines no error handling. MSM will tolerate this behavior; the entry pseudo state will simply be the newly active state.

Important note 2: UML defines transiting to an exit pseudo state and having no second transition as an error, and also defines no error handling. Therefore, it was decided to implement exit pseudo state as terminate states and the containing composite not properly exited will stay terminated as it was technically “exited”.

Important note 3: UML states that for the exit point, the same event must be used in both transitions. MSM relaxes this rule and only wants the event on the inside transition to be convertible to the one of the outside transition. In our case, event6 is convertible from event5. Notice that the forwarded event must be named in the exit point definition. For example, we could define event6 as simply as:

```
struct event
{
    event(){}
    template <class Event>
    event(Event const&){}
}; //convertible from any event
```

Note: There is a current limitation if you need not only convert but also get some data from the original event. Consider:

```
struct event1
{
    event1(int val_):val(val_) {}
    int val;
}; // forwarded from exit point
struct event2
{
    template <class Event>
    event2(Event const& e):val(e.val){} // compiler will complain about another
    int val;
}; // what the higher-level fsm wants to get
```

The solution is to provide two constructors:

```
struct event2
```

```
{
    template <class Event>
    event2(Event const& ):val(0){} // will not be used
    event2(event1 const& e)):val(e.val){} // the conversion constructor
    int val;
}; // what the higher-level fsm wants to get
```

Flags

This tutorial [examples/Flags.cpp] is devoted to a concept not defined in UML: flags. It has been added into MSM after proving itself useful on many occasions. Please, do not be frightened as we are not talking about ugly shortcuts made of an improbable collusion of Booleans.

If you look into the Boost.Statechart documentation you'll find this code:

```
if ( ( state_downcast< const NumLockOff * >() != 0 ) &&
     ( state_downcast< const CapsLockOff * >() != 0 ) &&
     ( state_downcast< const ScrollLockOff * >() != 0 ) )
```

While correct and found in many UML books, this can be error-prone and a potential time-bomb when your state machine grows and you add new states or orthogonal regions.

And most of all, it hides the real question, which would be “does my state machine's current state define a special property”? In this special case “are my keys in a lock state”? So let's apply the Fundamental Theorem of Software Engineering and move one level of abstraction higher.

In our player example, let's say we need to know if the player has a loaded CD. We could do the same:

```
if ( ( state_downcast< const Stopped * >() != 0 ) &&
     ( state_downcast< const Open * >() != 0 ) &&
     ( state_downcast< const Paused * >() != 0 ) &&
     ( state_downcast< const Playing * >() != 0 ))
```

Or flag these 4 states as CDLoaded-able. You add a flag_list type into each flagged state:

```
typedef mpl::vector1<CDLoaded> flag_list;
```

You can even define a list of flags, for example in Playing:

```
typedef mpl::vector2<PlayingPaused,CDLoaded> flag_list;
```

This means that Playing supports both properties. To check if your player has a loaded CD, check if your flag is active in the current state:

```
player p; if (p.is_flag_active<CDLoaded>()) ...
```

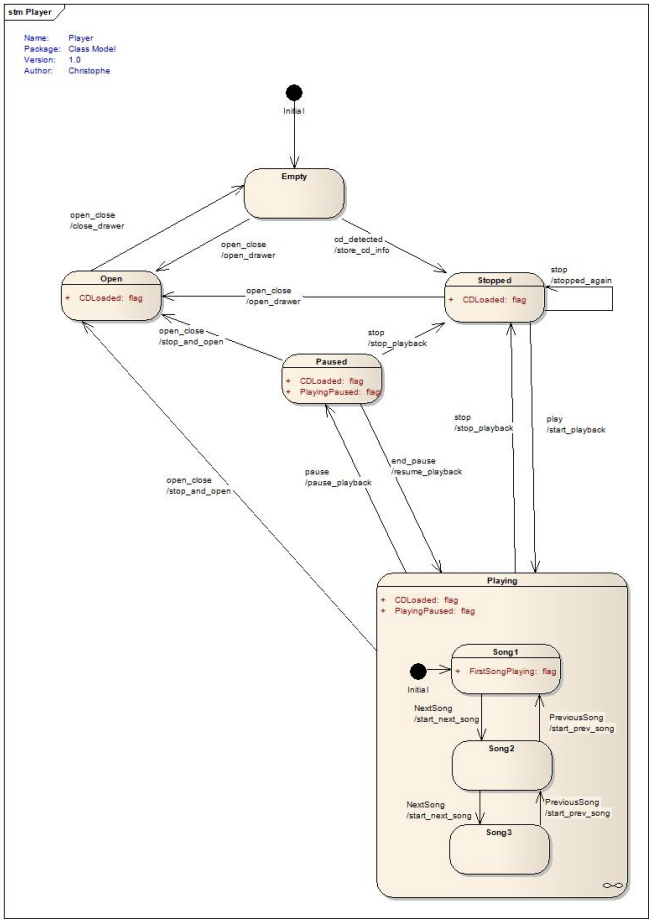
And what if you have orthogonal regions? How to decide if a state machine is in a flagged state? By default, you keep the same code and the current states will be OR'ed, meaning if one of the active states has the flag, then is_flag_active returns true. Of course, in some cases, you might want that all of the active states are flagged for the state to be active. You can also AND the active states:

```
if (p.is_flag_active<CDLoaded,player::Flag_AND>()) ...
```

Note. Due to arcane C++ rules, when called inside an action, the correct call is:

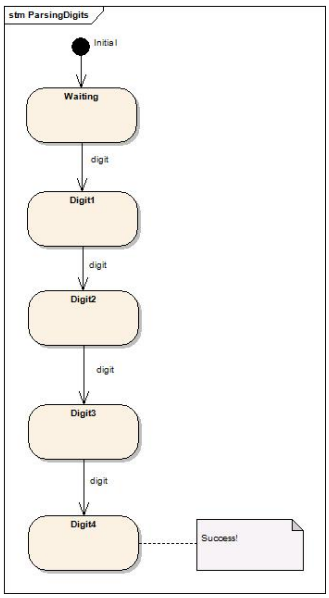
```
if (p.template is_flag_active<CDLoaded>()) ...
```

The following diagram displays the flag situation in the tutorial.



Event Hierarchy

There are cases where one needs transitions based on categories of events. An example is text parsing. Let's say you want to parse a string and use a state machine to manage your parsing state. You want to parse 4 digits and decide to use a state for every matched digit. Your state machine could look like:



But how to detect the digit event? We would like to avoid defining 10 transitions on char_0, char_1... between two states as it would force us to write 4 x 10 transitions and the compile-time would suffer.

To solve this problem, MSM supports the triggering of a transition on a subclass event. For example, if we define digits as:

```
struct digit {};
struct char_0 : public digit {};
```

And to the same for other digits, we can now fire char_0, char_1 events and this will cause a transition with "digit" as trigger to be taken.

An example [examples/ParsingDigits.cpp] with performance measurement, taken from the documentation of Boost.Xpressive illustrates this example. You might notice that the performance is actually very good (in this case even better).

Customizing a state machine / Getting more speed

MSM is offering many UML features at a high-speed, but sometimes, you just need more speed and are ready to give up some features in exchange. A process_event is handling several tasks:

- checking for terminate/interrupt states
- handling the message queue (for entry/exit/transition actions generating themselves events)
- handling deferred events
- catching exceptions (or not)
- handling the state switching and action calls

Of these tasks, only the last one is absolutely necessary to a state machine (its core job), the other ones are nice-to-haves which cost CPU time. In many cases, it is not so important, but in embedded systems, this can lead to ad-hoc state machine implementations. MSM detects by itself if a concrete state machine makes use of terminate/interrupt states and deferred events and deactivates them if not used. For the other two, if you do not need them, you need to help by indicating it in your implementation. This is done with two simple typedefs:

- `no_exception_thrown` indicates that behaviors will never throw and MSM does not need to catch anything
- `no_message_queue` indicates that no action will itself generate a new event and MSM can save us the message queue.

The third configuration possibility, explained here, is to manually activate deferred events, using `activate_deferred_events`. For example, the following state machine sets all three configuration types:

```
struct player_ : public msm::front::state_machine_def<player_>
{
   // no need for exception handling or message queue
   typedef int no_exception_thrown;
   typedef int no_message_queue;
   // also manually enable deferred events
   typedef int activate_deferred_events
   ...// rest of implementation
   };
```

Important note: As exit pseudo states are using the message queue to forward events out of a submachine, the `no_message_queue` option cannot be used with state machines containing an exit pseudo state.

Choosing the initial event

A state machine is started using the `start` method. This causes the initial state's entry behavior to be executed. Like every entry behavior, it becomes as parameter the event causing the state to be entered. But when the machine starts, there was no event triggered. In this case, MSM sends `msm::back::state_machine<...>::InitEvent`, which might not be the default you'd want. For this special case, MSM provides a configuration mechanism in the form of a typedef. If the state machine's front-end definition provides an initial_event typedef set to another event, this event will be used. For example:

```
struct my_initial_event{};
struct player_ : public msm::front::state_machine_def<player_>{
...
typedef my_initial_event initial_event;
};
```

Containing state machine (deprecated)

This feature is still supported in MSM for backward compatibility but made obsolete by the fact that every guard/action/entry action/exit action get the state machine passed as argument and might be removed at a later time.

All of the states defined in the state machine are created upon state machine construction. This has the huge advantage of a reduced syntactic noise. The cost is a small loss of control for the user on the state creation and access. But sometimes you needed a way for a state to get access to its containing state machine. Basically, a state needs to change its declaration to:

```
struct Stopped : public msm::front::state<sm_ptr>
```

And to provide a set_sm_ptr function: `void set_sm_ptr(player* pl)`

to get a pointer to the containing state machine. The same applies to terminate_state / interrupt_state and entry_pseudo_state / exit_pseudo_state.

Functor front-end

The functor front-end is the preferred front-end at the moment. It is more powerful than the standard front-end and has a more readable transition table. It also makes it easier to reuse parts of state machines. Like **eUML**, it also comes with a good deal of predefined actions. Actually, eUML generates a functor front-end through Boost.Typeof and Boost.Proto so both offer the same functionality.

The rows which MSM offered in the previous front-end come in different flavors. We saw the a_row, g_row, _row, row, not counting internal rows. This is already much to know, so why define new rows? These types have some disadvantages:

- They are more typing and information than we would wish. This means syntactic noise and more to learn.
- Function pointers are weird in C++.
- The action/guard signature is limited and does not allow for more variations of parameters (source state, target state, current state machine, etc.)
- It is not easy to reuse action code from a state machine to another.

Transition table

We can change the definition of the simple tutorial's transition table to:

```
struct transition_table : mpl::vector<
//    Start     Event         Target      Action                         Guard
//   +---------+------------+-----------+---------------------------+----------
Row  < Stopped , play       ,  Playing  , start_playback             , none
Row  < Stopped , open_close ,  Open     , open_drawer                , none
Row  < Stopped , stop       ,  Stopped  , none                       , none
//   +---------+------------+-----------+---------------------------+----------
Row  < Open    , open_close ,  Empty    , close_drawer               , none
//   +---------+------------+-----------+---------------------------+----------
Row  < Empty   , open_close ,  Open     , open_drawer                , none
Row  < Empty   , cd_detected,  Stopped  , store_cd_info              , good_disk
g_row< Empty   , cd_detected,  Playing  , &player_::store_cd_info    , &player_:
//   +---------+------------+-----------+---------------------------+----------
Row  < Playing , stop       ,  Stopped  , stop_playback              , none
Row  < Playing , pause      ,  Paused   , pause_playback             , none
Row  < Playing , open_close ,  Open     , stop_and_open              , none
//   +---------+------------+-----------+---------------------------+----------
Row  < Paused  , end_pause  ,  Playing  , resume_playback            , none
Row  < Paused  , stop       ,  Stopped  , stop_playback              , none
Row  < Paused  , open_close ,  Open     , stop_and_open              , none
//   +---------+------------+-----------+---------------------------+----------
> {};
```

Transitions are now of type "Row" with exactly 5 template arguments: source state, event, target state, action and guard. Wherever there is nothing (for example actions and guards), write "none". Actions and guards are no more methods but functors getting as arguments the detected event, the state machine, source and target state:

```
struct store_cd_info
{
    template <class Fsm,class Evt,class SourceState,class TargetState>
    void operator()(Evt const&, Fsm& fsm, SourceState&,TargetState& )
    {
        cout << "player::store_cd_info" << endl;
        fsm.process_event(play());
    }
};
```

The advantage of functors compared to functions are that functors are generic and reusable. They also allow passing more parameters than just events. The guard functors are the same but have an operator() returning a bool.

It is also possible to mix rows from different front-ends. To show this, a g_row has been left in the transition table. Note: in case the action functor is used in the transition table of a state machine contained inside a top-level state machine, the “fsm” parameter refers to the lowest-level state machine (referencing this action), not the top-level one.

To illustrate the reusable point, MSM comes with a whole set of predefined functors. Please refer to eUML for the full list. For example, we are now going to replace the first action by an action sequence and the guard by a more complex functor.

We decide we now want to execute two actions in the first transition (Stopped -> Playing). We only need to change the action start_playback to

```
ActionSequence_< mpl::vector<some_action, start_playback> >
```

and now will execute some_action and start_playback every time the transition is taken. ActionSequence_ is a functor calling each action of the mpl::vector in sequence.

We also want to replace good_disk_format by a condition of the type: “good_disk_format && (some_condition || some_other_condition)”. We can achieve this using And_ and Or_ functors:

```
And_<good_disk_format,Or_< some_condition , some_other_condition> >
```

It even starts looking like functional programming. MSM ships with functors for operators, state machine usage, STL algorithms or container methods.

Defining states with entry/exit actions

You probably noticed that we just showed a different transition table and that we even mixed rows from different front-ends. This means that you can do this and leave the definitions for states unchanged. Most examples are doing this as it is the simplest solution. You still enjoy the simplicity of the first front-end with the extended power of the new transition types. This tutorial [examples/SimpleWithFunctors.cpp], adapted from the earlier example does just this.

Of course, it is also possible to define states where entry and exit actions are also provided as functors as these are generated by eUML and both front-ends are equivalent. For example, we can define a state as:

```
struct Empty_Entry
{
    template <class Event,class Fsm,class State>
    void operator()(Event const&,Fsm&,State&)
    {
        ...
    }
}; // same for Empty_Exit
struct Empty_tag {};
struct Empty : public msm::front::euml::func_state<Empty_tag,Empty_Entry,Empty_
```

This also means that you can, like in the transition table, write entry / exit actions made of more complicated action combinations. The previous example can therefore be rewritten [examples/SimpleWithFunctors2.cpp].

Usually, however, one will probably use the standard state definition as it provides the same capabilities as this front-end state definition, unless one needs some of the shipped predefined functors or is a fan of functional programming.

What do you actually do inside actions / guards (Part 2)?

Using the basic front-end, we saw how to pass data to actions through the event, that data common to all states could be stored in the state machine, state relevant data could be stored in the state and access as template parameter in the entry / exit actions. What was however missing was the capability to access relevant state data in the transition action. This is possible with this front-end. A transition's source and target state are also given as arguments. If the current calculation's state was to be found in the transition's source state (whatever it is), we could access it:

```
struct send_rocket
{
    template <class Fsm,class Evt,class SourceState,class TargetState>
    void operator()(Evt const&, Fsm& fsm, SourceState& src,TargetState& )
    {
        fire_rocket(evt.direction, src.current_calculation);
    }
};
```

It was a little awkward to generate new events inside actions with the basic front-end. With the functor front-end it is much cleaner:

```
struct send_rocket
{
    template <class Fsm,class Evt,class SourceState,class TargetState>
    void operator()(Evt const& evt, Fsm& fsm, SourceState& src,TargetState&)
    {
        fire_rocket(evt.direction, src.current_calculation);
        fsm.process_event(rocket_launched());
    }
};
```

Defining a simple state machine

Like states, state machines can be defined using the previous front-end, as the previous example showed, or with the functor front-end, which allows you to define a state machine entry and exit functions as functors, as in this example [examples/SimpleWithFunctors2.cpp].

Anonymous transitions

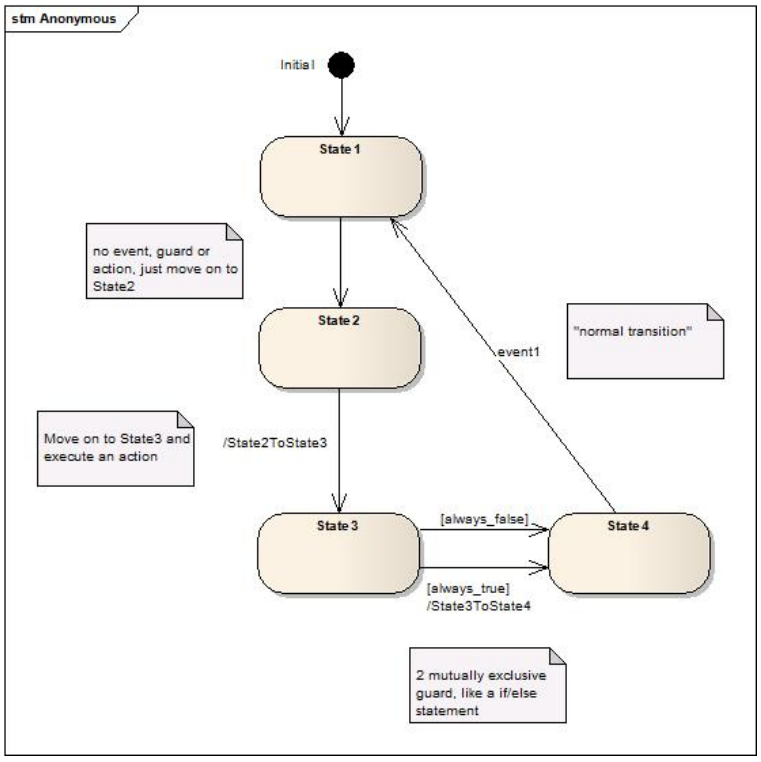
Anonymous (completion) transitions are transitions without a named event. We saw how this front-end uses `none` when no action or guard is required. We can also use `none` instead of an event to mark an anonymous transition. For example, the following transition makes an immediate transition from State1 to State2:

```
Row < State1 , none , State2 >
```

The following transition does the same but calling an action in the process:

```
Row < State1 , none , State2 , State1ToState2, none >
```

The following diagram shows an example and its implementation [examples/AnonymousTutorialWithFunctors.cpp]:



Internal transitions

The following example [examples/SimpleTutorialInternalFunctors.cpp] uses internal transitions with the functor front-end. As for the simple standard front-end, both methods of defining internal transitions are supported:

- providing a `Row` in the state machine's transition table with `none` as target state defines an internal transition.
- providing an `internal_transition_table` made of `Internal` rows inside a state or submachine defines UML-conform internal transitions with higher priority.
- transitions defined inside `internal_transition_table` require no source or target state as the source state is known (`Internal` really are `Row` without a source or target state) .

Like for the **standard front-end internal transitions**, internal transition tables are added into the main state machine's table, thus allowing you to distribute the transition table definition and reuse states.

There is an added bonus offered for submachines, which can have both the standard transition_table and an internal_transition_table (which has higher priority). This makes it easier if you decide to make a full submachine from a state later. It is also slightly faster than the standard alternative, adding orthogonal regions, because event dispatching will, if accepted by the internal table, not continue to the subregions. This gives you a O(1) dispatch instead of O(number of regions). While the example is with eUML, the same is also possible with this front-end.

Kleene (any) event

Normally, MSM requires an event to fire a transition. But there are cases, where any event, no matter which one would do:

- If you want to reduce the number of transitions: any event would do, possibly will guards decide what happens
- Pseudo entry states do not necessarily want to know the event which caused their activation, or they might want to know only a property of it.

MSM supports a boost::any as an acceptable event. This event will match any event, meaning that if a transition with boost::any as event originates from the current state, this transition would fire (provided no guards or transition with a higher priority fires first). This event is named Kleene, as reference top the Kleene star used in a regex.

For example, this transition on a state machine instance named fsm:

```
Row < State1, boost::any, State2>
```

will fire if State1 is active and an event is processed:

```
fsm.process_event(whatever_event());
```

At this point, you can use this *any* event in transition actions to get back to the original event by calling for example *boost::any::type()*.

It is also possible to support your own Kleene events by specializing boost::msm::is_kleene_event for a given event, for example:

```
namespace boost { namespace msm{
    template<>
    struct is_kleene_event< my_event >
    {
```

```
      typedef boost::mpl::true_ type;
    };
}}
```

The only requirement is that this event must have a copy constructor from the event originally processed on the state machine.

eUML

Important note: eUML requires a compiler supporting Boost.Typeof. Full eUML has experimental status (but not if only the transition table is written using eUML) because some compilers will start crashing when a state machine becomes too big (usually when you write huge actions).

The previous front-ends are simple to write but still force an amount of noise, mostly MPL types, so it would be nice to write code looking like C++ (with a C++ action language) directly inside the transition table, like UML designers like to do on their state machine diagrams. If it were functional programming, it would be even better. This is what eUML is for.

eUML is a Boost.Proto and Boost.Typeof-based compile-time domain specific embedded language. It provides grammars which allow the definition of actions/guards directly inside the transition table or entry/exit in the state definition. There are grammars for actions, guards, flags, attributes, deferred events, initial states.

It also relies on Boost.Typeof as a wrapper around the new decltype C++0x feature to provide a compile-time evaluation of all the grammars. Unfortunately, all the underlying Boost libraries are not Typeof-enabled, so for the moment, you will need a compiler where Typeof is supported (like VC9-10, g++ >= 4.3).

Examples will be provided in the next paragraphs. You need to include eUML basic features:

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

To add STL support (at possible cost of longer compilation times), include:

```
#include <msm/front/euml/stl.hpp>
```

eUML is defined in the namespace `msm::front::euml`.

Transition table

A transition can be defined using eUML as:

```
source + event [guard] / action == target
```

or as

```
target == source + event [guard] / action
```

The first version looks like a drawn transition in a diagram, the second one seems natural to a C++ developer.

The simple transition table written with the **functor front-end** can now be written as:

```
BOOST_MSM_EUML_TRANSITION_TABLE((
Stopped + play [some_guard] / (some_action , start_playback)  == Playing ,
Stopped + open_close/ open_drawer                             == Open    ,
Stopped + stop                                                == Stopped ,
Open    + open_close / close_drawer                           == Empty   ,
```

```
Empty   + open_close / open_drawer                              == Open    ,
Empty   + cd_detected [good_disk_format] / store_cd_info        == Stopped
),transition_table)
```

Or, using the alternative notation, it can be:

```
BOOST_MSM_EUML_TRANSITION_TABLE((
Playing  == Stopped + play [some_guard] / (some_action , start_playback) ,
Open     == Stopped + open_close/ open_drawer                            ,
Stopped  == Stopped + stop                                               ,
Empty    == Open    + open_close / close_drawer                          ,
Open     == Empty   + open_close / open_drawer                           ,
Stopped  == Empty   + cd_detected [good_disk_format] / store_cd_info
),transition_table)
```

The transition table now looks like a list of (readable) rules with little noise.

UML defines guards between “[ ]” and actions after a “/”, so the chosen syntax is already more readable for UML designers. UML also allows designers to define several actions sequentially (our previous ActionSequence_) separated by a comma. The first transition does just this: two actions separated by a comma and enclosed inside parenthesis to respect C++ operator precedence.

If this seems to you like it will cost you run-time performance, don't worry, eUML is based on typeof (or decltype) which only evaluates the parameters to BOOST_MSM_EUML_TRANSITION_TABLE and no run-time cost occurs. Actually, eUML is only a metaprogramming layer on top of "standard" MSM metaprogramming and this first layer generates the previously-introduced **functor front-end**.

UML also allows designers to define more complicated guards, like [good_disk_format && (some_condition || some_other_condition)]. This was possible with our previously defined functors, but using a complicated template syntax. This syntax is now possible exactly as written, which means without any syntactic noise at all.

A simple example: rewriting only our transition table

As an introduction to eUML, we will rewrite our tutorial's transition table using eUML. This will require two or three changes, depending on the compiler:

- events must inherit from msm::front::euml::euml_event< event_name >
- states must inherit from msm::front::euml::euml_state< state_name >
- with VC, states must be declared before the front-end

We now can write the transition table like just shown, using BOOST_MSM_EUML_DECLARE_TRANSITION_TABLE instead of BOOST_MSM_EUML_TRANSITION_TABLE. The implementation [examples/SimpleTutorialWithEumlTable.cpp] is pretty straightforward. The only required addition is the need to declare a variable for each state or add parenses (a default-constructor call) in the transition table.

The **composite** [examples/CompositeTutorialWithEumlTable.cpp] implementation is also natural:

```
// front-end like always
struct sub_front_end : public boost::msm::front::state_machine_def<sub_front_en
{
...
};
// back-end like always
typedef boost::msm::back::state_machine<sub_front_end> sub_back_end;
```

```
sub_back_end const sub; // sub can be used in a transition table.
```

Unfortunately, there is a bug with VC, which appears from time to time and causes in a stack overflow. If you get a warning that the program is recursive on all paths, revert to either standard eUML or another front-end as Microsoft doesn't seem to intend to fix it.

We now have a new, more readable transition table with few changes to our example. eUML can do much more so please follow the guide.

Defining events, actions and states with entry/exit actions

Events

Events must be proto-enabled. To achieve this, they must inherit from a proto terminal (euml_event<event-name>). eUML also provides a macro to make this easier:

```
BOOST_MSM_EUML_EVENT(play)
```

This declares an event type and an instance of this type called `play`, which is now ready to use in state or transition behaviors.

There is a second macro, BOOST_MSM_EUML_EVENT_WITH_ATTRIBUTES, which takes as second parameter the attributes an event will contain, using the **attribute syntax**.

Note: as we now have events defined as instances instead of just types, can we still process an event by creating one on the fly, like: `fsm.process_event(play());` or do we have to write: `fsm.process_event(play);`

The answer is you can do both. The second one is easier but unlike other front-ends, the second uses a defined operator(), which creates an event on the fly.

Actions

Actions (returning void) and guards (returning a bool) are defined like previous functors, with the difference that they also must be proto-enabled. This can be done by inheriting from euml_action< functor-name >. eUML also provides a macro:

```
BOOST_MSM_EUML_ACTION(some_condition)
{
    template <class Fsm,class Evt,class SourceState,class TargetState>
    bool operator()(Evt const& ,Fsm& ,SourceState&,TargetState& )
    { return true; }
};
```

Like for events, this macro declares a functor type and an instance for use in transition or state behaviors.

It is possible to use the same action grammar from the transition table to define state entry and exit behaviors. So `(action1,action2)` is a valid entry or exit behavior executing both actions in turn.

The state functors have a slightly different signature as there is no source and target state but only a current state (entry/exit actions are transition-independent), for example:

```
BOOST_MSM_EUML_ACTION(Empty_Entry)
{
    template <class Evt,class Fsm,class State>
    void operator()(Evt const& ,Fsm& ,State& ) { ... }
```

```
    };
```

It is also possible to reuse the functors from the functor front-end. The syntax is however slightly less comfortable as we need to pretend creating one on the fly for typeof. For example:

```
struct start_playback
{
        template <class Fsm,class Evt,class SourceState,class TargetState>
        void operator()(Evt const& ,Fsm&,SourceState& ,TargetState& )
        {
         ...
        }
};
BOOST_MSM_EUML_TRANSITION_TABLE((
Playing   == Stopped  + play        / start_playback() ,
...
),transition_table)
```

States

There is also a macro for states. This macro has 2 arguments, first the expression defining the state, then the state (instance) name:

```
BOOST_MSM_EUML_STATE((),Paused)
```

This defines a simple state without entry or exit action. You can provide in the expression parameter the state behaviors (entry and exit) using the action grammar, like in the transition table:

```
BOOST_MSM_EUML_STATE(((Empty_Entry,Dummy_Entry)/*2 entryactions*/,
                        Empty_Exit/*1 exit action*/ ),
                      Empty)
```

This means that Empty is defined as a state with an entry action made of two sub-actions, Empty_Entry and Dummy_Entry (enclosed inside parenthesis), and an exit action, Empty_Exit.

There are several possibilitites for the expression syntax:

- (): state without entry or exit action.
- (Expr1): state with entry but no exit action.
- (Expr1,Expr2): state with entry and exit action.
- (Expr1,Expr2,Attributes): state with entry and exit action, defining some attributes (read further on).
- (Expr1,Expr2,Attributes,Configure): state with entry and exit action, defining some attributes (read further on) and flags (standard MSM flags) or deferred events (standard MSM deferred events).
- (Expr1,Expr2,Attributes,Configure,Base): state with entry and exit action, defining some attributes (read further on), flags and deferred events (plain msm deferred events) and a non-default base state (as defined in standard MSM).

no_action is also defined, which does, well, nothing except being a placeholder (needed for example as entry action if we have no entry but an exit). Expr1 and Expr2 are a sequence of actions, obeying the same action grammar as in the transition table (following the “/” symbol).

The BOOST_MSM_EUML_STATE macro will allow you to define most common states, but sometimes you will need more, for example provide in your states some special behavior. In this case, you will have to do the macro's job by hand, which is not very complicated. The state will need to inherit from `msm::front::state<>`, like any state, and from `euml_state<state-name>` to be proto-enabled. You will then need to declare an instance for use in the transition table. For example:

```
struct Empty_impl : public msm::front::state<> , public euml_state<Empty_impl>
{
   void activate_empty() {std::cout << "switching to Empty " << std::endl;}
   template <class Event,class Fsm>
   void on_entry(Event const& evt,Fsm&fsm){...}
   template <class Event,class Fsm>
   void on_exit(Event const& evt,Fsm&fsm){...}
};
//instance for use in the transition table
Empty_impl const Empty;
```

Notice also that we defined a method named activate_empty. We would like to call it inside a behavior. This can be done using the BOOST_MSM_EUML_METHOD macro.

```
BOOST_MSM_EUML_METHOD(ActivateEmpty_,activate_empty,activate_empty_,void,void)
```

The first parameter is the name of the underlying functor, which you could use with the functor front-end, the second is the state method name, the third is the eUML-generated function, the fourth and fifth the return value when used inside a transition or a state behavior. You can now use this inside a transition:

```
Empty == Open + open_close / (close_drawer,activate_empty_(target_))
```

Wrapping up a simple state machine and first complete examples

You can reuse the state machine definition method from the standard front-end and simply replace the transition table by this new one. You can also use eUML to define a state machine "on the fly" (if, for example, you need to provide an on_entry/on_exit for this state machine as a functor). For this, there is also a macro, BOOST_MSM_EUML_DECLARE_STATE_MACHINE, which has 2 arguments, an expression describing the state machine and the state machine name. The expression has up to 8 arguments:

- (Stt, Init): simplest state machine where only the transition table and initial state(s) are defined.
- (Stt, Init, Expr1): state machine where the transition table, initial state and entry action are defined.
- (Stt, Init, Expr1, Expr2): state machine where the transition table, initial state, entry and exit actions are defined.
- (Stt, Init, Expr1, Expr2, Attributes): state machine where the transition table, initial state, entry and exit actions are defined. Furthermore, some attributes are added (read further on).
- (Stt, Init, Expr1, Expr2, Attributes, Configure): state machine where the transition table, initial state, entry and exit actions are defined. Furthermore, some attributes (read further on), flags, deferred events and configuration capabilities (no message queue / no exception catching) are added.
- (Stt, Init, Expr1, Expr2, Attributes, Flags, Deferred , Base): state machine where the transition table, initial state, entry and exit actions are defined. Furthermore, attributes (read further on), flags , deferred events and configuration capabilities (no message queue / no exception catching) are added and a non-default base state (see the back-end description) is defined.

For example, a minimum state machine could be defined as:

```
BOOST_MSM_EUML_TRANSITION_TABLE((
),transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE((transition_table,init_ << Empty ),
                                     player_)
```

Please have a look at the player tutorial written using eUML's first syntax [examples/SimpleTutorialEuml2.cpp] and second syntax [examples/SimpleTutorialEuml.cpp]. The BOOST_MSM_EUML_DECLARE_ATTRIBUTE macro, to which we will get back shortly, declares attributes given to an eUML type (state or event) using the **attribute syntax**.

Defining a submachine

Defining a submachine (see tutorial [examples/CompositeTutorialEuml.cpp]) with other front-ends simply means using a state which is a state machine in the transition table of another state machine. This is the same with eUML. One only needs define a second state machine and reference it in the transition table of the containing state machine.

Unlike the state or event definition macros, BOOST_MSM_EUML_DECLARE_STATE_MACHINE defines a type, not an instance because a type is what the back-end requires. This means that you will need to declare yourself an instance to reference your submachine into another state machine, for example:

```
BOOST_MSM_EUML_DECLARE_STATE_MACHINE(...,Playing_)
typedef msm::back::state_machine<Playing_> Playing_type;
Playing_type const Playing;
```

We can now use this instance inside the transition table of the containing state machine:

```
Paused == Playing + pause / pause_playback
```

Attributes / Function call

We now want to make our grammar more useful. Very often, one needs only very simple action methods, for example ++Counter or Counter > 5 where Counter is usually defined as some attribute of the class containing the state machine. It seems like a waste to write a functor for such a simple action. Furthermore, states within MSM are also classes so they can have attributes, and we would also like to provide them with attributes.

If you look back at our examples using the first [examples/SimpleTutorialEuml2.cpp] and second [examples/SimpleTutorialEuml.cpp] syntaxes, you will find a BOOST_MSM_EUML_DECLARE_ATTRIBUTE and a BOOST_MSM_EUML_ATTRIBUTES macro. The first one declares possible attributes:

```
BOOST_MSM_EUML_DECLARE_ATTRIBUTE(std::string,cd_name)
BOOST_MSM_EUML_DECLARE_ATTRIBUTE(DiskTypeEnum,cd_type)
```

This declares two attributes: cd_name of type std::string and cd_type of type DiskTypeEnum. These attributes are not part of any event or state in particular, we just declared a name and a type. Now, we can add attributes to our cd_detected event using the second one:

```
BOOST_MSM_EUML_ATTRIBUTES((attributes_ << cd_name << cd_type ),
                          cd_detected_attributes)
```

This declares an attribute list which is not linked to anything in particular yet. It can be attached to a state or an event. For example, if we want the event cd_detected to have these defined attributes we write:

```
BOOST_MSM_EUML_EVENT_WITH_ATTRIBUTES(cd_detected,cd_detected_attributes)
```

For states, we use the BOOST_MSM_EUML_STATE macro, which has an expression form where one can provide attributes. For example:

```
BOOST_MSM_EUML_STATE((no_action /*entry*/,no_action/*exit*/,
                      attributes_ << cd_detected_attributes),
```

```
some_state)
```

OK, great, we now have a way to add attributes to a class, which we could have done more easily, so what is the point? The point is that we can now reference these attributes directly, at compile-time, in the transition table. For example, in the example, you will find this transition:

```
Stopped==Empty+cd_detected[good_disk_format&&(event_(cd_type)==Int_<DISK_CD>())
```

Read event_(cd_type) as event_->cd_type with event_ a type generic for events, whatever the concrete event is (in this particular case, it happens to be a cd_detected as the transition shows).

The main advantage of this feature is that you do not need to define a new functor and you do not need to look inside the functor to know what it does, you have all at hand.

MSM provides more generic objects for state machine types:

- event_ : used inside any action, the event triggering the transition
- state_: used inside entry and exit actions, the entered / exited state
- source_: used inside a transition action, the source state
- target_: used inside a transition action, the target state
- fsm_: used inside any action, the (lowest-level) state machine processing the transition
- Int_<int value>: a functor representing an int
- Char_<value>: a functor representing a char
- Size_t_<value>: a functor representing a size_t
- String_<mpl::string> (boost >= 1.40): a functor representing a string.

These helpers can be used in two different ways:

- helper(attribute_name) returns the attribute with name attribute_name
- helper returns the state / event type itself.

The second form is helpful if you want to provide your states with their own methods, which you also want to use inside the transition table. In the above tutorial [examples/SimpleTutorialEuml.cpp], we provide Empty with an activate_empty method. We would like to create a eUML functor and call it from inside the transition table. This is done using the MSM_EUML_METHOD / MSM_EUML_FUNCTION macros. The first creates a functor to a method, the second to a free function. In the tutorial, we write:

```
MSM_EUML_METHOD(ActivateEmpty_,activate_empty,activate_empty_,void,void)
```

The first parameter is the functor name, for use with the functor front-end. The second is the name of the method to call. The third is the function name for use with eUML, the fourth is the return type of the function if used in the context of a transition action, the fifth is the result type if used in the context of a state entry / exit action (usually fourth and fifth are the same). We now have a new eUML function calling a method of "something", and this "something" is one of the five previously shown generic helpers. We can now use this in a transition, for example:

```
Empty == Open + open_close / (close_drawer,activate_empty_(target_))
```

The action is now defined as a sequence of two actions: close_drawer and activate_empty, which is called on the target itself. The target being Empty (the state defined left), this really will call Empty::activate_empty(). This method could also have an (or several) argument(s), for example the event, we could then call activate_empty_(target_ , event_).

More examples can be found in the terrible compiler stress test [examples/CompilerStressTestEuml.cpp], the timer example [examples/SimpleTimer.cpp] or in the iPodSearch with eUML [examples/iPodSearchEuml.cpp] (for String_ and more).

Orthogonal regions, flags, event deferring

Defining orthogonal regions really means providing more initial states. To add more initial states, “shift left” some, for example, if we had another initial state named AllOk :

```
BOOST_MSM_EUML_DECLARE_STATE_MACHINE((transition_table,
                                      init_ << Empty << AllOk ),
                                     player_)
```

You remember from the **BOOST_MSM_EUML_STATE** and **BOOST_MSM_EUML_DECLARE_STATE_MACHINE** signatures that just after attributes, we can define flags, like in the basic MSM front-end. To do this, we have another "shift-left" grammar, for example:

```
BOOST_MSM_EUML_STATE((no_action,no_action, attributes_ <<no_attributes_,
                       /* flags */ configure_<< PlayingPaused << CDLoaded),
                     Paused)
```

We now defined that Paused will get two flags, PlayingPaused and CDLoaded, defined, with another macro:

```
BOOST_MSM_EUML_FLAG(CDLoaded)
```

This corresponds to the following basic front-end definition of Paused:

```
struct Paused : public msm::front::state<>
{
   typedef mpl::vector2<PlayingPaused,CDLoaded> flag_list;
};
```

Under the hood, what you get really is a mpl::vector2.

Note: As we use the version of BOOST_MSM_EUML_STATE's expression with 4 arguments, we need to tell eUML that we need no attributes. Similarly to a `cout << endl`, we need a `attributes_ << no_attributes_` syntax.

You can use the flag with the is_flag_active method of a state machine. You can also use the provided helper function is_flag_ (returning a bool) for state and transition behaviors. For example, in the iPod implementation with eUML [examples/iPodEuml.cpp], you find the following transition:

```
ForwardPressed == NoForward + EastPressed[!is_flag_(NoFastFwd)]
```

The function also has an optional second parameter which is the state machine on which the function is called. By default, fsm_ is used (the current state machine) but you could provide a functor returning a reference to another state machine.

eUML also supports defining deferred events in the state (state machine) definition. To this aim, we can reuse the flag grammar. For example:

```
BOOST_MSM_EUML_STATE((Empty_Entry,Empty_Exit, attributes_ << no_attributes_,
                       /* deferred */ configure_<< play ),Empty)
```

The configure_ left shift is also responsible for deferring events. Shift inside configure_ a flag and the state will get a flag, shift an event and it will get a deferred event. This replaces the basic front-end definition:

```
typedef mpl::vector<play> deferred_events;
```

In this tutorial [examples/OrthogonalDeferredEuml.cpp], player is defining a second orthogonal region with AllOk as initial state. The `Empty` and `Open` states also defer the event `play`. `Open`, `Stopped` and `Pause` also support the flag `CDLoaded` using the same left shift into `configure_`.

In the functor front-end, we also had the possibility to defer an event inside a transition, which makes possible conditional deferring. This is also possible with eUML through the use of the defer_ order, as shown in this tutorial [examples/OrthogonalDeferredEuml.cpp]. You will find the following transition:

```
Open + play / defer_
```

This is an **internal transition**. Ignore it for the moment. Interesting is, that when the event `play` is fired and `Open` is active, the event will be deferred. Now add a guard and you can conditionally defer the event, for example:

```
Open + play [ some_condition ] / defer_
```

This is similar to what we did with the functor front-end. This means that we have the same constraints. Using defer_ instead of a state declaration, we need to tell MSM that we have deferred events in this state machine. We do this (again) using a configure_ declaration in the state machine definition in which we shift the deferred_events configuration flag:

```
BOOST_MSM_EUML_DECLARE_STATE_MACHINE((transition_table,
                                      init_ << Empty << AllOk,
                                      Entry_Action,
                                      Exit_Action,
                                      attributes_ << no_attributes_,
                                      configure_<< deferred_events ),
                                     player_)
```

A tutorial [examples/OrthogonalDeferredEuml2.cpp] illustrates this possibility.

Customizing a state machine / Getting more speed

We just saw how to use configure_ to define deferred events or flags. We can also use it to configure our state machine like we did with the other front-ends:

- `configure_ << no_exception`: disables exception handling
- `configure_ << no_msg_queue` deactivates the message queue
- `configure_ << deferred_events` manually enables event deferring

Deactivating the first two features and not activating the third if not needed greatly improves the event dispatching speed of your state machine. Our speed testing [examples/EumlSimple.cpp] example with eUML does this for the best performance.

Important note: As exit pseudo states are using the message queue to forward events out of a submachine, the `no_message_queue` option cannot be used with state machines containing an exit pseudo state.

Completion / Anonymous transitions

Anonymous transitions (See **UML tutorial**) are transitions without a named event, which are therefore triggered immediately when the source state becomes active, provided a guard allows it. As there is no event, to define such a transition, simply omit the "+" part of the transition (the event), for example:

```
State3 == State4 [always_true] / State3ToState4
State4 [always_true] / State3ToState4 == State3
```

Please have a look at this example [examples/AnonymousTutorialEuml.cpp], which implements the **previously defined** state machine with eUML.

Internal transitions

Like both other front-ends, eUML supports two ways of defining internal transitions:

- in the state machine's transition table. In this case, you need to specify a source state, event, actions and guards but no target state, which eUML will interpret as an internal transition, for example this defines a transition internal to Open, on the event open_close:

  ```
  Open + open_close [internal_guard1] / internal_action1
  ```

  A full example [examples/EumlInternal.cpp] is also provided.

- in a state's `internal_transition_table`. For example:

  ```
  BOOST_MSM_EUML_DECLARE_STATE((Open_Entry,Open_Exit),Open_def)
  struct Open_impl : public Open_def
  {
     BOOST_MSM_EUML_DECLARE_INTERNAL_TRANSITION_TABLE((
          open_close [internal_guard1] / internal_action1
     ))
  };
  ```

  Notice how we do not need to repeat that the transition originates from Open as we already are in Open's context.

  The implementation [examples/EumlInternalDistributed.cpp] also shows the added bonus offered for submachines, which can have both the standard transition_table and an internal_transition_table (which has higher priority). This makes it easier if you decide to make a full submachine from a state. It is also slightly faster than the standard alternative, adding orthogonal regions, because event dispatching will, if accepted by the internal table, not continue to the subregions. This gives you a O(1) dispatch instead of O(number of regions).

Kleene(any) event)

As for the functor front-end, eUML supports the concept of an ***any*** event, but boost::any is not an acceptable eUML terminal. If you need an *any* event, use msm::front::euml::kleene, which inherits boost::any. The same transition as with boost:any would be:

```
State1 + kleene == State2
```

Other state types

We saw the **build_state** function, which creates a simple state. Likewise, eUML provides other state-building macros for other types of states:

- BOOST_MSM_EUML_TERMINATE_STATE takes the same arguments as BOOST_MSM_EUML_STATE and defines, well, a terminate state.

- BOOST_MSM_EUML_INTERRUPT_STATE takes the same arguments as BOOST_MSM_EUML_STATE and defines an interrupt state. However, the expression argument must contain as first element the event ending the interruption, for example: `BOOST_MSM_EUML_INTERRUPT_STATE(( end_error /*end interrupt event*/,ErrorMode_Entry,ErrorMode_Exit ),ErrorMode)`

- BOOST_MSM_EUML_EXIT_STATE takes the same arguments as BOOST_MSM_EUML_STATE and defines an exit pseudo state. However, the

expression argument must contain as first element the event propagated from the exit point: `BOOST_MSM_EUML_EXIT_STATE(( event6 /*propagated event*/,PseudoExit1_Entry,PseudoExit1_Exit ),PseudoExit1)`

- BOOST_MSM_EUML_EXPLICIT_ENTRY_STATE defines an entry pseudo state. It takes 3 parameters: the region index to be entered is defined as an int argument, followed by the configuration expression like BOOST_MSM_EUML_STATE and the state name, so that `BOOST_MSM_EUML_EXPLICIT_ENTRY_STATE(0 /*region index*/, ( SubState2_Entry,SubState2_Exit ),SubState2)` defines an entry state into the first region of a submachine.
- BOOST_MSM_EUML_ENTRY_STATE defines an entry pseudo state. It takes 3 parameters: the region index to be entered is defined as an int argument, followed by the configuration expression like BOOST_MSM_EUML_STATE and the state name, so that `BOOST_MSM_EUML_ENTRY_STATE(0, ( PseudoEntry1_Entry,PseudoEntry1_Exit ),PseudoEntry1)` defines a pseudo entry state into the first region of a submachine.

To use these states in the transition table, eUML offers the functions `explicit_`, `exit_pt_` and `entry_pt_`. For example, a direct entry into the substate SubState2 from SubFsm2 could be:

```
explicit_(SubFsm2,SubState2) == State1 + event2
```

Forks being a list on direct entries, eUML supports a logical syntax (state1, state2, ...), for example:

```
(explicit_(SubFsm2,SubState2),
 explicit_(SubFsm2,SubState2b),
 explicit_(SubFsm2,SubState2c)) == State1 + event3
```

An entry point is entered using the same syntax as explicit entries:

```
entry_pt_(SubFsm2,PseudoEntry1) == State1 + event4
```

For exit points, it is again the same syntax except that exit points are used as source of the transition:

```
State2 == exit_pt_(SubFsm2,PseudoExit1) + event6
```

The entry tutorial [examples/DirectEntryEuml.cpp] is also available with eUML.

Helper functions

We saw a few helpers but there are more, so let us have a more complete description:

- event_ : used inside any action, the event triggering the transition
- state_: used inside entry and exit actions, the entered / exited state
- source_: used inside a transition action, the source state
- target_: used inside a transition action, the target state
- fsm_: used inside any action, the (deepest-level) state machine processing the transition
- These objects can also be used as a function and return an attribute, for example event_(cd_name)
- Int_<int value>: a functor representing an int
- Char_<value>: a functor representing a char
- Size_t_<value>: a functor representing a size_t
- True_ and False_ functors returning true and false respectively

- String_<mpl::string> (boost >= 1.40): a functor representing a string.
- if_then_else_(guard, action, action) where action can be an action sequence
- if_then_(guard, action) where action can be an action sequence
- while_(guard, action) where action can be an action sequence
- do_while_(guard, action) where action can be an action sequence
- for_(action, guard, action, action) where action can be an action sequence
- process_(some_event [, some state machine] [, some state machine] [, some state machine] [, some state machine]) will call process_event (some_event) on the current state machine or on the one(s) passed as 2nd , 3rd, 4th, 5th argument. This allow sending events to several external machines
- process_(event_): reprocesses the event which triggered the transition
- reprocess_(): same as above but shorter to write
- process2_(some_event,Value [, some state machine] [, some state machine] [, some state machine]) will call process_event (some_event(Value)) on the current state machine or on the one(s) passed as 3rd, 4th, 5th argument
- is_ flag_(some_flag[, some state machine]) will call is_flag_active on the current state machine or on the one passed as 2nd argument
- Predicate_<some predicate>: Used in STL algorithms. Wraps unary/binary functions to make them eUML-compatible so that they can be used in STL algorithms

This can be quite fun. For example,

```
/( if_then_else_(--fsm_(m_SongIndex) > Int_<0>(),/*if clause*/
                 show_playing_song, /*then clause*/
                 (fsm_(m_SongIndex)=Int_<1>(),process_(EndPlay))/*else clause*/
                 )
  )
```

means: if (fsm.SongIndex > 0, call show_playing_song else {fsm.SongIndex=1; process EndPlay on fsm;}

A few examples are using these features:

- the iPod example introduced at the BoostCon09 has been rewritten [examples/iPodEuml.cpp] with eUML (weak compilers please move on...)
- the iPodSearch example also introduced at the BoostCon09 has been rewritten [examples/iPodSearchEuml.cpp] with eUML. In this example, you will also find some examples of STL functor usage.
- A simpler timer [examples/SimpleTimer.cpp] example is a good starting point.

There is unfortunately a small catch. Defining a functor using MSM_EUML_METHOD or MSM_EUML_FUNCTION will create a correct functor. Your own eUML functors written as described at the beginning of this section will also work well, except, for the moment, with the while_, if_then_, if_then_else_ functions.

Phoenix-like STL support

eUML supports most C++ operators (except address-of). For example it is possible to write event_(some_attribute)++ or [source_(some_bool) && fsm_(some_other_bool)]. But a programmer needs more than operators in his daily programming. The STL is clearly a must have. Therefore, eUML

comes in with a lot of functors to further reduce the need for your own functors for the transition table. For almost every algorithm or container method of the STL, a corresponding eUML function is defined. Like Boost.Phoenix, "." And "->" of call on objects are replaced by a functional programming paradigm, for example:

- begin_(container), end_(container): return iterators of a container.
- empty_(container): returns container.empty()
- clear_(container): container.clear()
- transform_ : std::transform

In a nutshell, almost every STL method or algorithm is matched by a corresponding functor, which can then be used in the transition table or state actions. The reference lists all eUML functions and the underlying functor (so that this possibility is not reserved to eUML but also to the functor-based front-end). The file structure of this Phoenix-like library matches the one of Boost.Phoenix. All functors for STL algorithms are to be found in:

```
#include <msm/front/euml/algorithm.hpp>
```

The algorithms are also divided into sub-headers, matching the phoenix structure for simplicity:

```
#include < msm/front/euml/iteration.hpp>
#include < msm/front/euml/transformation.hpp>
#include < msm/front/euml/querying.hpp>
```

Container methods can be found in:

```
#include < msm/front/euml/container.hpp>
```

Or one can simply include the whole STL support (you will also need to include euml.hpp):

```
#include < msm/front/euml/stl.hpp>
```

A few examples (to be found in this tutorial [examples/iPodSearchEuml.cpp]):

- `push_back_(fsm_(m_tgt_container),event_(m_song))`: the state machine has an attribute m_tgt_container of type std::vector<OneSong> and the event has an attribute m_song of type OneSong. The line therefore pushes m_song at the end of m_tgt_container
- `if_then_( state_(m_src_it) != end_(fsm_(m_src_container)), process2_(OneSong(),*(state_(m_src_it)++)) )`: the current state has an attribute m_src_it (an iterator). If this iterator != fsm.m_src_container.end(), process OneSong on fsm, copy-constructed from state.m_src_it which we post-increment

Writing actions with Boost.Phoenix (in development)

It is also possible to write actions, guards, state entry and exit actions using a reduced set of Boost.Phoenix capabilities. This feature is still in development stage, so you might get here and there some surprise. Simple cases, however, should work well. What will not work will be mixing of eUML and Phoenix functors. Writing guards in one language and actions in another is ok though.

Phoenix also supports a larger syntax than what will ever be possible with eUML, so you can only use a reduced set of phoenix's grammar. This is due to the nature of eUML. The run-time transition table definition is translated to a type using Boost.Typeof. The result is a "normal" MSM transition table made of functor types. As C++ does not allow mixing run-time and compile-time constructs, there will be some limit (trying to instantiate a template class MyTemplateClass<i> where i is an int will give you an idea). This means following valid Phoenix constructs will not work:

- literals

- function pointers
- bind
- ->*

MSM also provides placeholders which make more sense in its context than arg1.. argn:

- _event: the event triggering the transition
- _fsm: the state machine processing the event
- _source: the source state of the transition
- _target: the target state of the transition
- _state: for state entry/exit actions, the entry/exit state

Future versions of MSM will support Phoenix better. You can contribute by finding out cases which do not work but should, so that they can be added.

Phoenix support is not activated by default. To activate it, add before any MSM header: #define BOOST_MSM_EUML_PHOENIX_SUPPORT.

A simple example [examples/SimplePhoenix.cpp] shows some basic capabilities.

eUML2

eUML2 is a C++11 only front-end trying to go even further than eUML on the way to a purely descriptive fsm. It is using the new Boost candidate library metaparse [http://abel.web.elte.hu/mpllibs/metaparse] from Abel Sinkovics.

eUML has the disadvantage that the transition table be valid C++ code, which means that whatever appears there (states, events, guards, actions) must be a valid C++ type. This forces us to write a lot of code. With eUML2, a transition is a string, allowing us to be completely free on the syntax and at the same type to only provide types we want to diverge from a default. On the other hand, we now need to be very careful because the compiler cannot protect us from typing mistakes.

Shortest state machine ever

Let us write the shortest state machine ever:

```
struct player_ : public msm::front::state_machine_def<player_>
{
    using initial_state = BOOST_MSM_EUML2_STATE("Empty",player_) ;
    EUML2_STT(
        player_,
        EUML2_STT_CFG(), // will be explained later on
        EUML2_ROW("Empty + open_close              / open_drawer   -> Open"),
        EUML2_ROW("Open  + open_close  [can_close] / close_drawer  -> Empty")
    )
};
typedef msm::back::state_machine<player_> my_fsm;
```

This compiles and even does something. Notice how close the syntax is from eUML.

We have:

- **source** + **event** -> **target**: a transition without guard or action
- **source** + **event** / **action** -> **target**: a transition without guard

- **source** + **event [guard]** -> **target**: a transition without action
- **source** + **event [guard] / action** -> **target**: a transition with guard and action
- **source** + **event**: an internal transition without guard or action
- **source** + **event / action**: an internal transition without guard
- **source** + **event [guard]**: an internal transition without action
- **source** + **event [guard] / action**: an internal transition with guard and action
- **EUML2_INTERNAL("event [guard] / action")** : used within a state, within a EUML2_STT_INTERNAL, an internal transition with guard and action
- **source** -> **target**: an anonymous transition without guard or action
- **source / action** -> **target**: an anonymous transition without guard. Other forms with / without guard/ action are of course provided
- **source** + ***** -> **target**: a transition with any (kleene) event. We now can use '*' for this.

Providing behaviour

The state machine previously defined compiles and does something but is not useful yet. Let us provide some behaviour to make it do something useful. Let us start with an action. Add the following after the previous code:

```
template<>
template <class Event,class Fsm,class SourceState,class TargetState>
void BOOST_MSM_EUML2_ACTION_IMPL("open_drawer",player_)::operator()(Event const&
{
    // do something useful
}
```

Notice how the signature is as always: we get as template parameters an event, fsm, source and target states. The syntax means: we have a default action names "open_drawer" for the front-end player_ (doing nothing) but we want to overwrite it using ours. The same way, we can provide a guard:

```
template<>
template <class Event,class Fsm,class SourceState,class TargetState>
bool BOOST_MSM_EUML2_GUARD_IMPL("can_close",player_)::operator()(Event const&,
{
    // do something useful
    return true;
}
```

Quite often using other front-end we are forced to declare states as empty structures so that we can use them in the transition table. This is no longer necessary so that we now have a function state machine within a few lines of code.

Also quite often, we are forced to declare states to provide a behaviour, always the same. We can now also avoid this tedious work, which used to make our code less readable. We only need provide a base state and tell the state machine to use it as default:

```
struct logging_base_state
{
    template <class Event,class FSM>
    void on_entry(Event const&,FSM& ) {std::cout << "Entering: " << name() << s
    template <class Event,class FSM>
    void on_entry(Event const&,FSM& ) {std::cout << "Exiting: " << name() << st
```

```
    virtual std::string name() const {return "";};
};
struct player_ : public msm::front::state_machine_def<player_,logging_base_sta
{
    using initial_state = BOOST_MSM_EUML2_STATE("Empty",player_) ;
    EUML2_STT(
        player_,
        EUML2_STT_CFG(), // will be explained later on
        EUML2_ROW("Empty + open_close              / open_drawer   -> Open"),
        EUML2_ROW("Open  + open_close  [can_close] / close_drawer  -> Empty")
    )
};
typedef msm::back::state_machine<player_> my_fsm;
```

By providing a base state in the front-end definition we ensure all states inherit from it and therefore will use the provided entry and exit behaviour. States also provide a **name()** member so that it is possible to use it for logging.

Note: to get the best compile time, we need to adjust to our longest string making a transition, for example:

```
#define BOOST_MPL_LIMIT_STRING_SIZE 65
#define MPLLIBS_LIMIT_STRING_SIZE BOOST_MPL_LIMIT_STRING_SIZE
// previous lines to be included before:
#include <boost/msm/front/euml2/euml2.hpp>
```

Please have a look at this minimum state machine [examples/eUML2Minimum.cpp].

More grammar capabilities

Like eUML, eUML2 supports UML-like action and guard syntax:

- Multiple actions separated by commas: action_1, action_n or (action_1, action_n)
- Logical operators for guards (following C++ rules): &&, ||, !, for example g1 && g2 || g3, g1 && ! (g2 || !g3)

When default behaviour is not enough

There are cases where these simple mechanisms to provide behaviour are not sufficient. When we have more complicated states (submachines, entry/exit pseudo states, etc.) and events we want to provide a type and use it. This is where our EUML2_STT_CFG comes of use. We can associate a type to a string and refer to this type within the transition table.

```
struct cd_detected
{
//...
};
struct Playing_ : public msm::front::state_machine_def<Playing_>
{
//...
};
typedef msm::back::state_machine<Playing_> Playing;
struct player_ : public msm::front::state_machine_def<player_>
{
    using initial_state = BOOST_MSM_EUML2_STATE("Empty",player_) ;
    EUML2_STT(
        player_,
        // use predefined cd_detected and Playing and associate them with the p
```

```
            EUML2_STT_CFG(EUML2_STT_USE("cd_detected",cd_detected),
                          EUML2_STT_USE("Playing",Playing)),
            EUML2_ROW("Empty   + cd_detected [good_disk && always_true] / store_cd
            EUML2_ROW("Playing + stop  / stop_pback -> Stopped"),
        )
};
typedef msm::back::state_machine<player_> my_fsm;
```

This allows us to use submachines, special states, or reuse previously defined types. This example [examples/eUML2Minimum.cpp] is using an already defined submachine.

Requirements

eUML2 is based on metaparse so it must be found within your boost directory. You also need a C++11 compiler. Tested were g++ 4.7 and higher and clang 3.3 and higher. Added compile-times compared to the functor front end are 0.5-1s per transition using clang and up to 2s using g++.

Back-end

There is, at the moment, one back-end. This back-end contains the library engine and defines the performance and functionality trade-offs. The currently available back-end implements most of the functionality defined by the UML 2.0 standard at very high runtime speed, in exchange for longer compile-time. The runtime speed is due to a constant-time double-dispatch and self-adapting capabilities allowing the framework to adapt itself to the features used by a given concrete state machine. All unneeded features either disable themselves or can be manually disabled. See section 5.1 for a complete description of the run-to-completion algorithm.

Creation

MSM being divided between front and back-end, one needs to first define a front-end. Then, to create a real state machine, the back-end must be declared:

```
typedef msm::back::state_machine<my_front_end> my_fsm;
```

We now have a fully functional state machine type. The next sections will describe what can be done with it.

Starting and stopping a state machine

The `start()` method starts the state machine, meaning it will activate the initial state, which means in turn that the initial state's entry behavior will be called. We need the start method because you do not always want the entry behavior of the initial state to be called immediately but only when your state machine is ready to process events. A good example of this is when you use a state machine to write an algorithm and each loop back to the initial state is an algorithm call. Each call to start will make the algorithm run once. The iPodSearch [examples/iPodSearch.cpp] example uses this possibility.

The `stop()` method works the same way. It will cause the exit actions of the currently active states(s) to be called.

Both methods are actually not an absolute need. Not calling them will simply cause your first entry or your last exit action not to be called.

Event dispatching

The main reason to exist for a state machine is to dispatch events. For MSM, events are objects of a given event type. The object itself can contain data, but the event type is what decides of the transition to be taken. For MSM, if some_event is a given type (a simple struct for example) and e1 and e2

concrete instances of some_event, e1 and e2 are equivalent, from a transition perspective. Of course, e1 and e2 can have different values and you can use them inside actions. Events are dispatched as const reference, so actions cannot modify events for obvious side-effect reasons. To dispatch an event of type some_event, you can simply create one on the fly or instantiate if before processing:

```
my_fsm fsm; fsm.process_event(some_event());
some_event e1; fsm.process_event(e1)
```

Creating an event on the fly will be optimized by the compiler so the performance will not degrade.

Active state(s)

The backend also offers a way to know which state is active, though you will normally only need this for debugging purposes. If what you need simply is doing something with the active state, **internal transitions** or **visitors** are a better alternative. If you need to know what state is active, const int* current_state() will return an array of state ids. Please refer to the **internals section** to know how state ids are generated.

Serialization

A common need is the ability to save a state machine and restore it at a different time. MSM supports this feature for the basic and functor front-ends, and in a more limited manner for eUML. MSM supports boost::serialization out of the box (by offering a `serialize` function). Actually, for basic serialization, you need not do much, a MSM state machine is serializable almost like any other type. Without any special work, you can make a state machine remember its state, for example:

```
MyFsm fsm;
// write to archive
std::ofstream ofs("fsm.txt");
// save fsm to archive
{
    boost::archive::text_oarchive oa(ofs);
    // write class instance to archive
    oa << fsm;
}
```

Loading back is very similar:

```
MyFsm fsm;
{
    // create and open an archive for input
    std::ifstream ifs("fsm.txt");
    boost::archive::text_iarchive ia(ifs);
    // read class state from archive
    ia >> fsm;
}
```

This will (de)serialize the state machine itself but not the concrete states' data. This can be done on a per-state basis to reduce the amount of typing necessary. To allow serialization of a concrete state, provide a do_serialize typedef and implement the serialize function:

```
struct Empty : public msm::front::state<>
{
    // we want Empty to be serialized. First provide the typedef
    typedef int do_serialize;
    // then implement serialize
    template<class Archive>
    void serialize(Archive & ar, const unsigned int /* version */)
    {
```

```
        ar & some_dummy_data;
    }
    Empty():some_dummy_data(0){}
    int some_dummy_data;
};
```

You can also serialize data contained in the front-end class. Again, you need to provide the typedef and implement serialize:

```
struct player_ : public msm::front::state_machine_def<player_>
{
    //we might want to serialize some data contained by the front-end
    int front_end_data;
    player_():front_end_data(0){}
    // to achieve this, provide the typedef
    typedef int do_serialize;
    // and implement serialize
    template<class Archive>
    void serialize(Archive & ar, const unsigned int )
    {
        ar & front_end_data;
    }
...
};
```

The saving of the back-end data (the current state(s)) is valid for all front-ends, so a front-end written using eUML can be serialized. However, to serialize a concrete state, the macros like `BOOST_MSM_EUML_STATE` cannot be used, so the state will have to be implemented by directly inheriting from `front::euml::euml_state`.

The only limitiation is that the event queues cannot be serialized so serializing must be done in a stable state, when no event is being processed. You can serialize during event processing only if using no queue (deferred or event queue).

This example [examples/Serialize.cpp] shows a state machine which we serialize after processing an event. The `Empty` state also has some data to serialize.

Base state type

Sometimes, one needs to customize states to avoid repetition and provide a common functionality, for example in the form of a virtual method. You might also want to make your states polymorphic so that you can call typeid on them for logging or debugging. It is also useful if you need a visitor, like the next section will show. You will notice that all front-ends offer the possibility of adding a base type. Note that all states and state machines must have the same base state, so this could reduce reuse. For example, using the basic front end, you need to:

- Add the non-default base state in your msm::front::state<> definition, as first template argument (except for interrupt_states for which it is the second argument, the first one being the event ending the interrupt), for example, my_base_state being your new base state for all states in a given state machine:

  ```
  struct Empty : public msm::front::state<my_base_state>
  ```

  Now, my_base_state is your new base state. If it has a virtual function, your states become polymorphic. MSM also provides a default polymorphic base type, `msm::front::polymorphic_state`

- Add the user-defined base state in the state machine frontend definition, as a second template argument, for example:

```
struct player_ : public msm::front::state_machine<player_,my_base_state>
```

You can also ask for a state with a given id (which you might have gotten from current_state()) using `const base_state* get_state_by_id(int id) const` where base_state is the one you just defined. You can now do something polymorphically.

Visitor

In some cases, having a pointer-to-base of the currently active states is not enough. You might want to call non-virtually a method of the currently active states. It will not be said that MSM forces the virtual keyword down your throat!

To achieve this goal, MSM provides its own variation of a visitor pattern using the previously described user-defined state technique. If you add to your user-defined base state an `accept_sig` typedef giving the return value (unused for the moment) and parameters and provide an accept method with this signature, calling visit_current_states will cause accept to be called on the currently active states. Typically, you will also want to provide an empty default accept in your base state in order in order not to force all your states to implement accept. For example your base state could be:

```
struct my_visitable_state
{
   // signature of the accept function
   typedef args<void> accept_sig;
   // we also want polymorphic states
   virtual ~my_visitable_state() {}
   // default implementation for states who do not need to be visited
   void accept() const {}
};
```

This makes your states polymorphic and visitable. In this case, accept is made const and takes no argument. It could also be:

```
struct SomeVisitor {…};
struct my_visitable_state
{
    // signature of the accept function
    typedef args<void,SomeVisitor&> accept_sig;
    // we also want polymorphic states
    virtual ~my_visitable_state() {}
    // default implementation for states who do not need to be visited
    void accept(SomeVisitor&) const {}
};
```

And now, `accept` will take one argument (it could also be non-const). By default, `accept` takes up to 2 arguments. To get more, set #define BOOST_MSM_VISITOR_ARG_SIZE to another value before including state_machine.hpp. For example:

```
#define BOOST_MSM_VISITOR_ARG_SIZE 3
#include <boost/msm/back/state_machine.hpp>
```

Note that accept will be called on ALL active states and also automatically on sub-states of a submachine.

Important warning: The method visit_current_states takes its parameter by value, so if the signature of the accept function is to contain a parameter passed by reference, pass this parameter with a boost:ref/cref to avoid undesired copies or slicing. So, for example, in the above case, call:

```
SomeVisitor vis; sm.visit_current_states(boost::ref(vis));
```

This example [examples/SM-2Arg.cpp] uses a visiting function with 2 arguments.

Flags

Flags is a MSM-only concept, supported by all front-ends, which base themselves on the functions:

```
template <class Flag> bool is_flag_active()
template <class Flag,class BinaryOp> bool is_flag_active()
```

These functions return true if the currently active state(s) support the Flag property. The first variant ORs the result if there are several orthogonal regions, the second one expects OR or AND, for example:

```
my_fsm.is_flag_active<MyFlag>()
my_fsm.is_flag_active<MyFlag,my_fsm_type::Flag_OR>()
```

Please refer to the front-ends sections for usage examples.

Getting a state

It is sometimes necessary to have the client code get access to the states' data. After all, the states are created once for good and hang around as long as the state machine does so why not use it? You simply just need sometimes to get information about any state, even inactive ones. An example is if you want to write a coverage tool and know how many times a state was visited. To get a state, use the get_state method giving the state name, for example:

```
player::Stopped* tempstate = p.get_state<player::Stopped*>();
```

or

```
player::Stopped& tempstate2 = p.get_state<player::Stopped&>();
```

depending on your personal taste.

State machine constructor with arguments

You might want to define a state machine with a non-default constructor. For example, you might want to write:

```
struct player_ : public msm::front::state_machine_def<player_>
{
    player_(int some_value){…}
};
```

This is possible, using the back-end as forwarding object:

```
typedef msm::back::state_machine<player_ > player; player p(3);
```

The back-end will call the corresponding front-end constructor upon creation.

You can pass arguments up to the value of the BOOST_MSM_CONSTRUCTOR_ARG_SIZE macro (currently 5) arguments. Change this value before including any header if you need to overwrite the default.

You can also pass arguments by reference (or const-reference) using boost::ref (or boost::cref):

```
struct player_ : public msm::front::state_machine_def<player_>
{
   player_(SomeType& t, int some_value){…}
};
```

```
typedef msm::back::state_machine<player_ > player;
SomeType data;
player p(boost::ref(data),3);
```

Normally, MSM default-constructs all its states or submachines. There are however cases where you might not want this. An example is when you use a state machine as submachine, and this submachine used the above defined constructors. You can add as first argument of the state machine constructor an expression where existing states are passed and copied:

```
player p( back::states_ << state_1 << ... << state_n , boost::ref(data),3);
```

Where state_1..n are instances of some or all of the states of the state machine. Submachines being state machines, this can recurse, for example, if Playing is a submachine containing a state Song1 having itself a constructor where some data is passed:

```
player p( back::states_ << Playing(back::states_ << Song1(some_Song1_data)) ,
          boost::ref(data),3);
```

It is also possible to replace a given state by a new instance at any time using `set_states()` and the same syntax, for example:

```
p.set_states( back::states_ << state_1 << ... << state_n );
```

An example [examples/Constructor.cpp] making intensive use of this capability is provided.

Trading run-time speed for better compile-time / multi-TU compilation

MSM is optimized for run-time speed at the cost of longer compile-time. This can become a problem with older compilers and big state machines, especially if you don't really care about run-time speed that much and would be satisfied by a performance roughly the same as most state machine libraries. MSM offers a back-end policy to help there. But before you try it, if you are using a VC compiler, deactivate the /Gm compiler option (default for debug builds). This option can cause builds to be 3 times longer... If the compile-time still is a problem, read further. MSM offers a policy which will speed up compiling in two main cases:

- many transition conflicts
- submachines

The back-end `msm::back::state_machine` has a policy argument (first is the front-end, then the history policy) defaulting to `favor_runtime_speed`. To switch to `favor_compile_time`, which is declared in `<msm/back/favor_compile_time.hpp>`, you need to:

- switch the policy to `favor_compile_time` for the main state machine (and possibly submachines)
- move the submachine declarations into their own header which includes `<msm/back/favor_compile_time.hpp>`
- add for each submachine a cpp file including your header and calling a macro, which generates helper code, for example:

  ```
  #include "mysubmachine.hpp"
  BOOST_MSM_BACK_GENERATE_PROCESS_EVENT(mysubmachine)
  ```

- configure your compiler for multi-core compilation

You will now compile your state machine on as many cores as you have submachines, which will greatly speed up the compilation if you factor your state machine into smaller submachines.

Independently, transition conflicts resolution will also be much faster.

This policy uses boost.any behind the hood, which means that we will lose a feature which MSM offers with the default policy, event hierarchy. The following example takes our iPod example and speeds up compile-time by using this technique. We have:

- our main state machine and main function [examples/iPod_distributed/iPod.cpp]
- PlayingMode moved to a separate header [examples/iPod_distributed/PlayingMode.hpp]
- a cpp for PlayingMode [examples/iPod_distributed/PlayingMode.cpp]
- MenuMode moved to a separate header [examples/iPod_distributed/MenuMode.hpp]
- a cpp for MenuMode [examples/iPod_distributed/MenuMode.cpp]
- events move to a separate header as all machines use it [examples/iPod_distributed/Events.hpp]

Compile-time state machine analysis

A MSM state machine being a metaprogram, it is only logical that cheking for the validity of a concrete state machine happens compile-time. To this aim, using the compile-time graph library mpl_graph [http://www.dynagraph.org/mpl_graph/] (delivered at the moment with MSM) from Gordon Woodhull, MSM provides several compile-time checks:

- Check that orthogonal regions ar truly orthogonal.
- Check that all states are either reachable from the initial states or are explicit entries / pseudo-entry states.

To make use of this feature, the back-end provides a policy (default is no analysis), `msm::back::mpl_graph_fsm_check`. For example:

```
typedef msm::back::state_machine< player_,msm::back::mpl_graph_fsm_check> play
```

As MSM is now using Boost.Parameter to declare policies, the policy choice can be made at any position after the front-end type (in this case `player_`).

In case an error is detected, a compile-time assertion is provoked.

This feature is not enabled by default because it has a non-neglectable compile-time cost. The algorithm is linear if no explicit or pseudo entry states are found in the state machine, unfortunately still O(number of states * number of entry states) otherwise. This will be improved in future versions of MSM.

The same algorithm is also used in case you want to omit providing the region index in the **explicit entry / pseudo entry state** declaration.

The author's advice is to enable the checks after any state machine structure change and disable it again after sucessful analysis.

The following example [examples/TestErrorOrthogonality.cpp] provokes an assertion if one of the first two lines of the transition table is used.

Enqueueing events for later processing

Calling `process_event(Event const&)` will immediately process the event with run-to-completion semantics. You can also enqueue the events and delay their processing by calling

`enqueue_event(Event const&)` instead. Calling `execute_queued_events()` will then process all enqueued events (in FIFO order). Calling `execute_single_queued_event()` will execute the oldest enqueued event.

You can query the queue size by calling `get_message_queue_size()`.

Customizing the message queues

MSM uses by default a std::deque for its queues (one message queue for events generated during run-to-completion or with `enqueue_event`, one for deferred events). Unfortunately, on some STL implementations, it is a very expensive container in size and copying time. Should this be a problem, MSM offers an alternative based on boost::circular_buffer. The policy is msm::back::queue_container_circular. To use it, you need to provide it to the back-end definition:

```
typedef msm::back::state_machine< player_,msm::back::queue_container_circular>
```

You can access the queues with get_message_queue and get_deferred_queue, both returning a reference or a const reference to the queues themselves. Boost::circular_buffer is outside of the scope of this documentation. What you will however need to define is the queue capacity (initially is 0) to what you think your queue will at most grow, for example (size 1 is common):

```
fsm.get_message_queue().set_capacity(1);
```

Policy definition with Boost.Parameter

MSM uses Boost.Parameter to allow easier definition of back::state_machine<> policy arguments (all except the front-end). This allows you to define policy arguments (history, compile-time / run-time, state machine analysis, container for the queues) at any position, in any number. For example:

```
typedef msm::back::state_machine< player_,msm::back::mpl_graph_fsm_check> play
typedef msm::back::state_machine< player_,msm::back::AlwaysHistory> player;
typedef msm::back::state_machine< player_,msm::back::mpl_graph_fsm_check,msm::
typedef msm::back::state_machine< player_,msm::back::AlwaysHistory,msm::back::
```

Choosing when to switch active states

The UML Standard is silent about a very important question: when a transition fires, at which exact point is the target state the new active state of a state machine? At the end of the transition? After the source state has been left? What if an exception is thrown? The Standard considers that run-to-completion means a transition completes in almost no time. But even this can be in some conditions a very very long time. Consider the following example. We have a state machine representing a network connection. We can be `Connected` and `Disconnected`. When we move from one state to another, we send a (Boost) Signal to another entity. By default, MSM makes the target state as the new state after the transition is completed. We want to send a signal based on a flag is_connected which is true when in state Connected.

We are in state `Disconnected` and receive an event `connect`. The transition action will ask the state machine `is_flag_active<is_connected>` and will get... false because we are still in `Disconnected`. Hmm, what to do? We could queue the action and execute it later, but it means an extra queue, more work and higher run-time.

MSM provides the possibility (in form of a policy) for a front-end to decide when the target state becomes active. It can be:

- before the transition fires, if the guard will allow the transition to fire: `active_state_switch_before_transition`
- after calling the exit action of the source state: `active_state_switch_after_exit`

- after the transition action is executed: `active_state_switch_after_transition_action`
- after the entry action of the target state is executed (default): `active_state_switch_after_entry`

The problem and the solution is shown for the functor-front-end [examples/ActiveStateSetBeforeTransition.cpp] and eUML [examples/ActivateStateBeforeTransitionEuml.cpp]. Removing `active_state_switch_before_transition` will show the default state.

Chapter 4. Performance / Compilers

Tests were made on different PCs running Windows XP and Vista and compiled with VC9 SP1 or Ubuntu and compiled with g++ 4.2 and 4.3. For these tests, the same player state machine was written using Boost.Statechart, as a state machine with only simple states [examples/SCSimple.cpp] and as a state machine with a composite state [examples/SCComposite.cpp]. The same simple and composite state machines are implemented with MSM with a standard frontend (simple) [examples/MsmSimple.cpp](composite) [examples/MsmComposite.cpp], the simple one also with functors [examples/MsmSimpleFunctors.cpp] and with eUML [examples/EumlSimple.cpp]. As these simple machines need no terminate/interrupt states, no message queue and have no-throw guarantee on their actions, the MSM state machines are defined with minimum functionality. Test machine is a Q6600 2.4GHz, Vista 64.

Speed

VC9:

- The simple test completes 90 times faster with MSM than with Boost.Statechart
- The composite test completes 25 times faster with MSM

gcc 4.2.3 (Ubuntu 8.04 in VMWare, same PC):

- The simple test completes 46 times faster with MSM
- The composite test completes 19 times faster with Msm

Executable size

There are some worries that MSM generates huge code. Is it true? The 2 compilers I tested disagree with this claim. On VC9, the test state machines used in the performance section produce executables of 14kB (for simple and eUML) and 21kB (for the composite). This includes the test code and iostreams. By comparison, an empty executable with iostreams generated by VC9 has a size of 7kB. Boost.Statechart generates executables of 43kB and 54kB. As a bonus, eUML comes for “free” in terms of executable size. You even get a speed gain. With g++ 4.3, it strongly depends on the compiler options (much more than VC). A good size state machine with –O3 can generate an executable of 600kB, and with eUML you can get to 1.5MB. Trying with –Os –s I come down to 18kB and 30kB for the test state machines, while eUML will go down to 1MB (which is still big), so in this case eUML does not come for free.

Supported compilers

For a current status, have a look at the regression tests [http://www.boost.org/development/tests/trunk/developer/msm.html].

MSM was successfully tested with:

- VC8 (partly), VC9, VC10
- g++ 4.0.1 and higher
- Intel 10.1 and higher
- Clang 2.9
- Green Hills Software MULTI for ARM v5.0.5 patch 4416 (Simple and Composite tutorials)

- Partial support for IBM compiler

VC8 and to some lesser extent VC9 suffer from a bug. Enabling the option "Enable Minimal Rebuild" (/Gm) will cause much higher compile-time (up to three times with VC8!). This option being activated per default in Debug mode, this can be a big problem.

Limitations

- Compilation times of state machines with > 80 transitions that are going to make you storm the CFO's office and make sure you get a shiny octocore with 12GB RAM by next week, unless he's interested in paying you watch the compiler agonize for hours... (Make sure you ask for dual 24" as well, it doesn't hurt).
- eUML allows very long constructs but will also quickly increase your compile time on some compilers (VC9, VC10) with buggy decltype support (I suspect some at least quadratic algorithms there). Even g++ 4.4 shows some regression compared to 4.3 and will crash if the constructs become too big.
- Need to overwrite the mpl::vector/list default-size-limit of 20 and fusion default vector size of 10 if more than 10 states found in a state machine
- **Limitation for submachines** and entry actions requiring an event property.

Compilers corner

Compilers are sometimes full of surprises and such strange errors happened in the course of the development that I wanted to list the most fun for readers' entertainment.

VC8:

```
template <class StateType>
typename ::boost::enable_if<
       typename ::boost::mpl::and_<
                typename ::boost::mpl::not_<
                    typename has_exit_pseudo_states<StateType>::type
                >::type,
                typename ::boost::mpl::not_<
                    typename is_pseudo_exit<StateType>::type
                >::type
       >::type,
       BaseState*>::type
```

I get the following error:

error C2770: invalid explicit template argument(s) for '`global namespace'::boost::enable_if<...>::...'

If I now remove the first "::" in ::boost::mpl , the compiler shuts up. So in this case, it is not possible to follow Boost's guidelines.

VC9:

- This one is my all times' favorite. Do you know why the exit pseudo states are referenced in the transition table with a "submachine::exit_pt" ? Because "exit" will crash the compiler. "Exit" is not possible either because it will crash the compiler on one machine, but not on another (the compiler was installed from the same disk).
- Sometimes, removing a policy crashes the compiler, so some versions are defining a dummy policy called WorkaroundVC9.

- Typeof: While g++ and VC9 compile “standard” state machines in comparable times, Typeof (while in both ways natively supported) seems to behave in a quadratic complexity with VC9 and VC10.
- eUML: in case of a compiler crash, changing the order of state definitions (first states without entry or exit) sometimes solves the problem.

g++ 4.x: Boring compiler, almost all is working almost as expected. Being not a language lawyer I am unsure about the following “Typeof problem”. VC9 and g++ disagree on the question if you can derive from the BOOST_TYPEOF generated type without first defining a typedef. I will be thankful for an answer on this. I only found two ways to break the compiler:

- Add more eUML constructs until something explodes (especially with g++-4.4)
- The build_terminate function uses 2 mpl::push_back instead of mpl::insert_range because g++ would not accept insert_range.

You can test your compiler’s decltype implementation with the following stress test [examples/CompilerStressTestEuml.cpp] and reactivate the commented-out code until the compiler crashes.

Chapter 5. Questions & Answers, tips

Where should I define a state machine?: The tutorials are implemented in a simple cpp source file for simplicity. I want to model dynamic behavior of a class as a state machine, how should I define the state machine?

Answer: Usually you'll want to implement the state machine as an attribute of the class. Unfortunately, a concrete state machine is a typedef, which cannot be forward-declared. This leaves you with two possibilities:

- Provide the state machine definition inside the header class and contain an instance as attribute. Simple, but with several drawbacks: using namespace directives are not advised, and compile-time cost for all modules including the header.
- Keep the state machine as (shared) pointer to void inside the class definition [examples/FsmAsPtr.hpp], and implement the state machine in the cpp file [examples/FsmAsPtr.cpp]. Minimum compile-time, using directives are okay, but the state machine is now located inside the heap.

Question: on_entry gets as argument, the sent event. What event do I get when the state becomes default-activated (because it is an initial state)?

Answer: To allow you to know that the state was default-activated, MSM generates a boost::msm::InitEvent default event.

Question: Why do I see no call to no_transition in my submachine?

Answer: Because of the priority rule defined by UML. It says that in case of transition conflict, the most inner state has a higher priority. So after asking the inner state, the containing composite has to be also asked to handle the transition and could find a possible transition.

Question: Why do I get a compile error saying the compiler cannot convert to a function ...Fsm::*(some_event)?

Answer: You probably defined a transition triggered by the event some_event, but used a guard/action method taking another event.

Question: Why do I get a compile error saying something like “too few” or “too many” template arguments?

Answer: You probably defined a transition in form of a a_row or g_row where you wanted just a _row or the other way around. With Row, it could mean that you forgot a "none".

Question: Why do I get a very long compile error when I define more than 20 rows in the transition table?

Answer: MSM uses Boost.MPL under the hood and this is the default maximum size. Please define the following 3 macros before including any MSM headers:

```
#define BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS
#define BOOST_MPL_LIMIT_VECTOR_SIZE 30 // or whatever you need
#define BOOST_MPL_LIMIT_MAP_SIZE 30 // or whatever you need
```

Question: Why do I get this error: ”error C2977: 'boost::mpl::vector' : too many template arguments”?

Answer: The first possibility is that you defined a transition table as, say, vector17 and have 18 entries. The second is that you have 17 entries and have a composite state. Under the hood, MSM adds a row for every event in the composite transition table. The third one is that you used a mpl::vector without the number of entries but are close to the MPL default of 50 and have a composite, thus pushing you above 50. Then you need mpl/vector60/70….hpp and a mpl/map60/70….hpp

Question: Why do I get a very long compile error when I define more than 10 states in a state machine?

Answer: MSM uses Boost.Fusion under the hood and this is the default maximum size. Please define the following macro before including any MSM headers:

```
#define FUSION_MAX_VECTOR_SIZE 20 // or whatever you need
```

Chapter 6. Internals

This chapter describes the internal machinery of the back-end, which can be useful for UML experts but can be safely ignored for most users. For implementers, the interface between front- and back-end is also described in detail.

Backend: Run To Completion

The back-end implements the following run-to completion algorithm:

- Check if one region of the concrete state machine is in a terminate or interrupt state. If yes, event processing is disabled while the condition lasts (forever for a terminate pseudo-state, while active for an interrupt pseudo-state).
- If the message queue feature is enabled and if the state machine is already processing an event, push the currently processed event into the queue and end processing. Otherwise, remember that the state machine is now processing an event and continue.
- If the state machine detected that no deferred event is used, skip this step. Otherwise, mark the first deferred event from the deferred queue as active.
- Now start the core of event dispatching. If exception handling is activated, this will happen inside a try/catch block and the front-end `exception_caught` is called if an exception occurs.
- The event is now dispatched in turn to every region, in the order defined by the initial state front-end definition. This will, for every region, call the corresponding front-end transition definition (the "row" or "Row" of the transition table).
- Without transition conflict, if for a given region a transition is possible, the guard condition is checked. If it returns `true`, the transition processing continues and the current state's exit action is called, followed by the transition action behavior and the new active state's entry behavior.
- With transition conflicts (several possible transitions, disambiguated by mutually exclusive guard conditions), the guard conditions are tried in reverse order of their transition definition in the transition table. The first one returning `true` selects its transition. Note that this is not defined by the UML standard, which simply specifies that if the guard conditions are not mutually exclusive, the state machine is ill-formed and the behaviour undefined. Relying on this implementation-specific behaviour will make it harder for the developer to support another state machine framework.
- If at least one region processes the event, this event is seen as having been accepted. If not, the library calls `no_transition` on the state machine for every contained region.
- If the currently active state is a submachine, the behaviour is slightly different. The UML standard specifies that internal transitions have to be tried first, so the event is first dispatched to the submachine. Only if the submachine does not accept the event are other (non internal) transitions tried.
- This back-end supports simple states' and submachines' internal transitions. These are provided in the state's `internal_transition_table` type. Transitions defined in this table are added at the end of the main state machine's transition table, but with a lesser priority than the submachine's transitions (defined in `transition_table`). This means, for simple states, that these transitions have higher priority than non-internal transitions, conform to the UML standard which gives higher priority to deeper-level transitions. For submachines, this is a non-standard addition which can help make event processing faster by giving a chance to bypass subregion processing. With standard UML, one would need to add a subregion only to process these internal transitions, which would be slower.
- After the dispatching itself, the deferred event marked in step 3 (if any) now gets a chance of processing.

- Then, events queued in the message queue also get a dispatching chance
- Finally, completion / anonymous transitions, if to be found in the transition table, also get their dispatching chance.

This algorithm illustrates how the back-end configures itself at compile-time as much as possible. Every feature not found in a given state machine definition is deactivated and has therefore no runtime cost. Completion events, deferred events, terminate states, dispatching to several regions, internal transitions are all deactivated if not used. User configuration is only for exception handling and message queue necessary.

Frontend / Backend interface

The design of MSM tries to make front-ends and back-ends (later) to be as interchangeable as possible. Of course, no back-end will ever implement every feature defined by any possible front-end and inversely, but the goal is to make it as easy as possible to extend the current state of the library.

To achieve this, MSM divides the functionality between both sides: the front-end is a sort of user interface and is descriptive, the back-end implements the state machine engine.

MSM being based on a transition table, a concrete state machine (or a given front-end) must provide a transition_table. This transition table must be made of rows. And each row must tell what kind of transition it is and implement the calls to the actions and guards. A state machine must also define its regions (marked by initial states) And that is about the only constraints for front-ends. How the rows are described is implementer's choice.

Every row must provide:

- A `Source` typedef indicating, well, the type of the source state.
- A `Target` typedef indicating, well, the type of the target state.
- A `Evt` typedef indicating the type of the event triggering the transition.
- A `row_type_tag` typedef indicating the type of the transition.
- Rows having a type requiring transition actions must provide a static function `action_call` with the following signature: `template <class Fsm,class SourceState,class TargetState,class AllStates>`

  ```
  static void action_call (Fsm& fsm, Event const& evt, SourceState&, TargetState&, AllStates&)
  ```

  The function gets as parameters the (back-end) state machine, the event, source and target states and a container (in the current back-end, a fusion::set) of all the states defined in the state machine. For example, as the back-end has the front-end as basic class, `action_call` is simply defined as `(fsm.*action)(evt)`.
- Rows having a type requiring a guard must provide a static function `guard_call` with the following signature:

  ```
  template <class Fsm,class SourceState,class TargetState,class AllStates>

  static bool guard_call (Fsm&, Event const&, SourceState&, TargetState&, AllStates&)
  ```
- The possible transition (row) types are:
  - a_row_tag: a transition with actions and no guard

- g_row_type: a transition with a guard and no actions
- _row_tag: a transition without actions or guard
- row_tag: a transition with guard and actions
- a_irow_tag: an internal transition (defined inside the `transition_table`) with actions
- g_irow_tag: an internal transition (defined inside the `transition_table`) with guard
- irow_tag: an internal transition (defined inside the `transition_table`) with actions and guards
- _irow_tag: an internal transition (defined inside the `transition_table`) without action or guard. Due to higher priority for internal transitions, this is equivalent to a "ignore event"
- sm_a_i_row_tag: an internal transition (defined inside the `internal_transition_table`) with actions
- sm_g_i_row_tag: an internal transition (defined inside the `internal_transition_table`) with guard
- sm_i_row_tag: an internal transition (defined inside the `internal_transition_table`) with actions and guards
- sm__i_row_tag: an internal transition (defined inside the `internal_transition_table`) without action or guard. Due to higher priority for internal transitions, this is quivalent to a "ignore event"

Furthermore, a front-end must provide the definition of states and state machines. State machine definitions must provide (the implementer is free to provide it or let it be done by every concrete state machine. Different MSM front-ends took one or the other approach):

- `initial_state`: This typedef can be a single state or a mpl container and provides the initial states defining one or several orthogonal regions.
- `transition_table`: This typedef is a MPL sequence of transition rows.
- `configuration`: this typedef is a MPL sequence of known types triggering special behavior in the back-end, for example if a concrete fsm requires a message queue or exception catching.

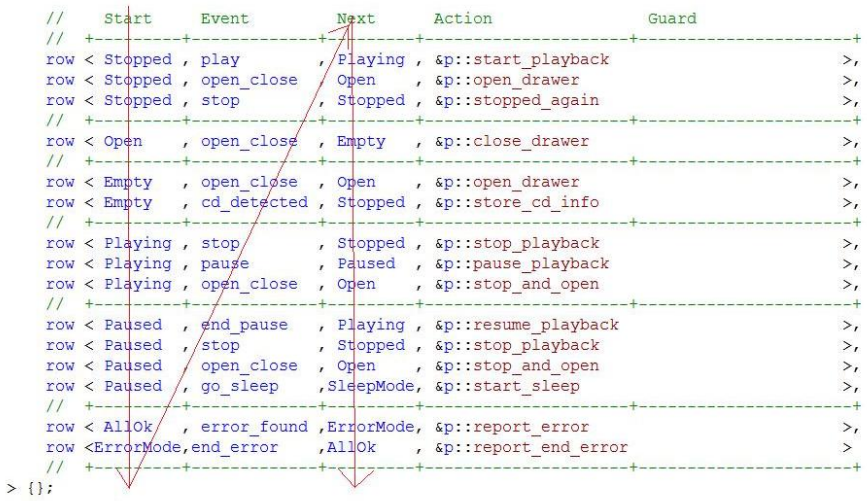
States and state machines must both provide a (possibly empty) definition of:

- `flag_list`: the flags being active when this state or state machine become the current state of the fsm.
- `deferred_events`: events being automatically deferred when the state is the current state of the fsm.
- `internal_transition_table`: the internal transitions of this state.
- `on_entry` and `on_exit` methods.

Generated state ids

Normally, one does not need to know the ids are generated for all the states of a state machine, unless for debugging purposes, like the pstate function does in the tutorials in order to display the name of the current state. This section will show how to automatically display typeid-generated names, but these are not very readable on all platforms, so it can help to know how the ids are generated. The ids

are generated using the transition table, from the “Start” column up to down, then from the “Next” column, up to down, as shown in the next image:



Stopped will get id 0, Open id 1, ErrorMode id 6 and SleepMode (seen only in the “Next” column) id 7. If you have some implicitly created states, like transition-less initial states or states created using the explicit_creation typedef, these will be added as a source at the end of the transition table. If you have submachine states, a row will be added for them at the end of the table, after the automatically or explicitly created states, which can change their id. The next help you will need for debugging would be to call the current_state method of the state_machine class, then the display_type helper to generate a readable name from the id. If you do not want to go through the transition table to fill an array of names, the library provides another helper, fill_state_names, which, given an array of sufficient size (please see next section to know how many states are defined in the state machine), will fill it with typeid-generated names.

Metaprogramming tools

We can find for the transition table more uses than what we have seen so far. Let's suppose you need to write a coverage tool. A state machine would be perfect for such a job, if only it could provide some information about its structure. Thanks to the transition table and Boost.MPL, it does.

What is needed for a coverage tool? You need to know how many states are defined in the state machine, and how many events can be fired. This way you can log the fired events and the states visited in the life of a concrete machine and be able to perform some coverage analysis, like “fired 65% of all possible events and visited 80% of the states defined in the state machine”. To achieve this, MSM provides a few useful tools:

- generate_state_set<transition table>: returns a mpl::set of all the states defined in the table.
- generate_event_set<transition table>: returns a mpl::set of all the events defined in the table.
- using mpl::size<>::value you can get the number of elements in the set.
- display_type defines an operator() sending typeid(Type).name() to cout.
- fill_state_names fills an array of char const* with names of all states (found by typeid)
- using mpl::for_each on the result of generate_state_set and generate_event_set passing display_type as argument will display all the states of the state machine.

- let's suppose you need to recursively find the states and events defined in the composite states and thus also having a transition table. Calling recursive_get_transition_table<Composite> will return you the transition table of the composite state, recursively adding the transition tables of all sub-state machines and sub-sub...-sub-state machines. Then call generate_state_set or generate_event_set on the result to get the full list of states and events.

An example [examples/BoostCon09Full.cpp] shows the tools in action.

Chapter 7. Acknowledgements

I am in debt to the following people who helped MSM along the way.

MSM v2

- Thanks to Dave Abrahams for managing the review
- Thanks to Eric Niebler for his patience correcting my grammar errors
- Special thanks to Joel de Guzman who gave me very good ideas at the BoostCon09. These ideas were the starting point of the redesign. Any time again, Joel #
- Thanks to Richard O'Hara for making Green Hills bring a patch in less than 1 week, thus adding one more compiler to the supported list.
- Big thanks to those who took the time to write a review: Franz Alt, David Bergman, Michael Caisse, Barend Gehrels, Darryl Greene, Juraj Ivancic, Erik Nelson, Kenny Riddile.
- Thanks to Matt Calabrese, Juraj Ivancic, Adam Merz and Joseph Wu for reporting bugs.
- Thanks to Thomas Mistretta for providing an addition to the section "What do you actually do inside actions / guards".

MSM v1

- The original version of this framework is based on the brilliant work of David Abrahams and Aleksey Gurtovoy who laid down the base and the principles of the framework in their excellent book, "C++ template Metaprogramming". The implementation also makes heavy use of the boost::mpl.
- Thanks to Jeff Flinn for his idea of the user-defined base state and his review which allowed MSM to be presented at the BoostCon09.
- Thanks to my MSM v1 beta testers, Christoph Woskowski and Franz Alt for using the framework with little documentation and to my private reviewer, Edouard Alligand

Chapter 8. Version history

From V2.27 to V2.28 (Boost 1.57)

- Fixed BOOST_MSM_EUML_EVENT_WITH_ATTRIBUTES (broken in 1.56).
- Fixed execute_queued_events, added execute_single_queued_event
- Fixed warnings for unused variables

From V2.26 to V2.27 (Boost 1.56)

- Bugfix: no_transition in case of an exception.
- Bugfix: Trac 9280
- Bugfix: incomplete namespace names in eUML

From V2.25 to V2.26 (Boost 1.55)

- New feature: interrupt states now support a sequence of events to end the interruption
- Bugfix: Trac 8686.

From V2.24 to V2.25 (Boost 1.54)

- Bugfix: Exit points broken for the favor_compile_time policy.
- Bugfix: copy breaks exit points of subsubmachines.
- Bugfix: Trac 8046.

From V2.23 to V2.24 (Boost 1.51)

- Support for **boost::any** or **kleene** as an acceptable event.
- Bugfix: compiler error with fsm internal table and `none`(compound) event.
- Bugfix: `euml::defer_` leading to stack overflow.

From V2.22 to V2.23 (Boost 1.50)

- **eUML** : better syntax for front-ends defined with eUML as transititon table only. Caution: Breaking Change!
- Bugfix: graph building was only working if `initial_state` defined as a sequence
- Bugfix: flags defined for a Terminate or Interrupt state do not break the blocking function of these states any more.
- Bugfix: multiple deferred events from several regions were not working in every case.
- Bugfix: visitor was passed by value to submachines.
- Bugfix: `no_transition` was not called for submachines who send an event to themselves.

- Fixed warnings with gcc

From V2.21 to V2.22 (Boost 1.48)

- eUML: added easier event reprocessing: `process(event_)` and `reprocess()`
- Rewrite of internal transition tables. There were a few bugs (failing recursivity in internal transition tables of sub-sub machines) and a missing feature (unused internal transition table of the main state machine).
- Bugfixes
  - Reverted favor_compile_time policy to Boost 1.46 state
  - `none` event now is convertible from any other event
  - eUML and pseudo exit states
  - Fixed not working Flag_AND
  - Fixed rare bugs causing multiple processing of the same event in a submachine whose transition table contains this event and a base event of it.
  - gcc warnings about unused variables
- Breaking change: the new internal transition table feature causes a minor breaking change. In a submachine, the "Fsm" template parameter for guards / actions of an internal table declared using `internal_transition_table` now is the submachine, not the higher-level state machine. Internal transitions declared using internal rows in the higher-level state machine keep their behavior (the "Fsm" parameter is the higher-level state machine). To sum up, the internal transition "Fsm" parameter is the closest state machine containing this transition.

From V2.20 to V2.21 (Boost 1.47)

- Added a **stop()** method in the back-end.
- **Added partial support for Boost.Phoenix functors in eUML**
- Added the possibility to choose when **state switching** occurs.
- Bugfixes
  - Trac 5117, 5253, 5533, 5573
  - gcc warnings about unused variables
  - better implemenation of favor_compile_time back-end policy
  - bug with eUML and state construction
  - incorrect eUML event and state macros
  - incorrect event type passed to a direct entry state's on_entry action
  - more examples

From V2.12 to V2.20 (Boost 1.46)

- Compile-time state machine analysis using mpl_graph:

    - **checking of region orthogonality**.
    - **search for unreachable states**.
    - **automatic region index search for pseudo entry or explicit entry states**.
- **Boost.Parameter interface definition** for msm::back::state_machine<> template arguments.
- **Possibility to provide a container** for the event and deferred event queues. A policy implementation based on a more efficient Boost.CircularBuffer is provided.
- msm::back::state_machine<>::is_flag_active method made const.
- added possibility to **enqueue events** for delayed processing.
- Bugfixes
    - Trac 4926
    - stack overflow using the Defer functor
    - anonymous transition of a submachine not called for the initial state

From V2.10 to V2.12 (Boost 1.45)

- Support for **serialization**
- **Possibility to use normal functors** (from functor front-end) in eUML.
- **New constructors** where substates / submachines can be taken as arguments. This allows passing arguments to the constructor of a submachine.
- Bugfixes

From V2.0 to V2.12 (Boost 1.44)

- New documentation
- Internal transitions. Either as part of the transition table or using a state's internal transition table
- increased dispatch and copy speed
- **new row types** for the basic front-end
- new eUML syntax, better attribute support, macros to ease developer's life. Even VC8 seems to like it better.
- New policy for reduced compile-time at the cost of dispatch speed
- Support for base events
- possibility to choose the initial event

Part II. Reference

Table of Contents

Chapter 9. External references to MSM

An interesting mapping UML <-> MSM from Takatoshi Kondo can be found at **Redboltz [http://redboltz.wikidot.com/boost-msm-guide]**.

Chapter 10. eUML operators and basic helpers

The following table lists the supported operators:

Table 10.1. Operators and state machine helpers

eUML function / operator	Description	Functor
&&	Calls lazily Action1&& Action2	And_
||	Calls lazily Action1|| Action2	Or_
!	Calls lazily !Action1	Not_
!=	Calls lazily Action1 != Action2	NotEqualTo_
==	Calls lazily Action1 == Action2	EqualTo_
>	Calls lazily Action1 > Action2	Greater_
>=	Calls lazily Action1 >= Action2	Greater_Equal_
<	Calls lazily Action1 < Action2	Less_
<=	Calls lazily Action1 <= Action2	Less_Equal_
&	Calls lazily Action1 & Action2	Bitwise_And_
|	Calls lazily Action1 | Action2	Bitwise_Or_
^	Calls lazily Action1 ^ Action2	Bitwise_Xor_
--	Calls lazily --Action1 / Action1--	Pre_Dec_ / Post_Dec_
++	Calls lazily ++Action1 / Action1++	Pre_Inc_ / Post_Inc_
/	Calls lazily Action1 / Action2	Divides_
/=	Calls lazily Action1 /= Action2	Divides_Assign_
*	Calls lazily Action1 * Action2	Multiplies_
*=	Calls lazily Action1 *= Action2	Multiplies_Assign_
+ (binary)	Calls lazily Action1 + Action2	Plus_
+ (unary)	Calls lazily +Action1	Unary_Plus_
+=	Calls lazily Action1 += Action2	Plus_Assign_
- (binary)	Calls lazily Action1 - Action2	Minus_
- (unary)	Calls lazily -Action1	Unary_Minus_
-=	Calls lazily Action1 -= Action2	Minus_Assign_
%	Calls lazily Action1 % Action2	Modulus_
%=	Calls lazily Action1 %= Action2	Modulus_Assign_
>>	Calls lazily Action1 >> Action2	ShiftRight_
>>=	Calls lazily Action1 >>= Action2	ShiftRight_Assign_
<<	Calls lazily Action1 << Action2	ShiftLeft_
<<=	Calls lazily Action1 <<= Action2	ShiftLeft_Assign_
[] (works on vector, map, arrays)	Calls lazily Action1 [Action2]	Subscript_

eUML function / operator	Description	Functor
if_then_else_(Condition,Action1,Action2)	Returns either the result of calling Action1 or the result of calling Action2	If_Else_
if_then_(Condition,Action)	Returns the result of calling Action if Condition	If_Then_
while_(Condition, Body)	While Condition(), calls Body(). Returns nothing	While_Do_
do_while_(Condition, Body)	Calls Body() while Condition(). Returns nothing	Do_While_
for_(Begin,Stop,EndLoop,Body)	Calls for(Begin;Stop;EndLoop) {Body;}	For_Loop_
process_(Event [,fsm1] [,fsm2] [,fsm3] [,fsm4])	Processes Event on the current state machine (if no fsm specified) or on up to 4 state machines returned by an appropriate functor.	Process_
process2_(Event, Data [,fsm1] [,fsm2] [,fsm3])	Processes Event on the current state machine (if no fsm specified) or on up to 2 state machines returned by an appropriate functor. The event is copy-constructed from what Data() returns.	Process2_
is_flag_(Flag [,fsm])	Calls is_flag_active() on the current state machine or the one returned by calling fsm.	Get_Flag_
event_ [(attribute name)]	Returns the current event (as const reference)	GetEvent_
source_ [(attribute name)]	Returns the source state of the currently triggered transition (as reference). If an attribute name is provided, returns the attribute by reference.	GetSource_
target_ [(attribute name)]	Returns the target state of the currently triggered transition (as reference). If an attribute name is provided, returns the attribute by reference.	GetTarget_
state_ [(attribute name)]	Returns the source state of the currently active state (as reference). Valid inside a state entry/exit action. If an attribute name is provided, returns the attribute by reference.	GetState_
fsm_ [(attribute name)]	Returns the current state machine (as reference). Valid inside a state entry/exit action or a transition. If an attribute name is provided, returns the attribute by reference.	GetFsm_

eUML function / operator	Description	Functor
substate_(state_name [,fsm])	Returns (as reference) the state state_name referenced in the current state machine or the one given as argument.	SubState_

To use these functions, you need to include:

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

Chapter 11. Functional programming

To use these functions, you need to include:

```
#include <msm/front/euml/stl.hpp>
```

or the specified header in the following tables.

The following tables list the supported STL algorithms:

Table 11.1. STL algorithms

STL algorithms in querying.hpp	Functor
find_(first, last, value)	Find_
find_if_(first, last, value)	FindIf_
lower_bound_(first, last, value [,op#])	LowerBound_
upper_bound_(first, last, value [,op#])	UpperBound_
equal_range_(first, last, value [,op#])	EqualRange_
binary_search_(first, last, value [,op#])	BinarySearch_
min_element_(first, last[,op#])	MinElement_
max_element_(first, last[,op#])	MaxElement_
adjacent_find_(first, last[,op#])	AdjacentFind_
find_end_(first1, last1, first2, last2 [,op #])	FindEnd_
find_first_of_(first1, last1, first2, last2 [,op #])	FindFirstOf_
equal_(first1, last1, first2 [,op #])	Equal_
search_(first1, last1, first2, last2 [,op #])	Search_
includes_(first1, last1, first2, last2 [,op #])	Includes_
lexicographical_compare_ (first1, last1, first2, last2 [,op #])	LexicographicalCompare_
count_(first, last, value [,size])	Count_
count_if_(first, last, op # [,size])	CountIf_
distance_(first, last)	Distance_
mismatch _(first1, last1, first2 [,op #])	Mismatch_

Table 11.2. STL algorithms

STL algorithms in iteration.hpp	Functor
for_each_(first,last, unary op#)	ForEach_
accumulate_first, last, init [,op#])	Accumulate_

Table 11.3. STL algorithms

STL algorithms in transformation.hpp	Functor
copy_(first, last, result)	Copy_
copy_backward_(first, last, result)	CopyBackward_
reverse_(first, last)	Reverse_
reverse_copy_(first, last , result)	ReverseCopy_
remove_(first, last, value)	Remove_

STL algorithms in transformation.hpp	**Functor**
remove_if_(first, last , op#)	RemoveIf_
remove_copy_(first, last , output, value)	RemoveCopy_
remove_copy_if_(first, last, output, op#)	RemoveCopyIf_
fill_(first, last, value)	Fill_
fill_n_(first, size, value)#	FillN_
generate_(first, last, generator#)	Generate_
generate_(first, size, generator#)#	GenerateN_
unique_(first, last [,op#])	Unique_
unique_copy_(first, last, output [,op#])	UniqueCopy_
random_shuffle_(first, last [,op#])	RandomShuffle_
rotate_copy_(first, middle, last, output)	RotateCopy_
partition_ (first, last [,op#])	Partition_
stable_partition_ (first, last [,op#])	StablePartition_
stable_sort_(first, last [,op#])	StableSort_
sort_(first, last [,op#])	Sort_
partial_sort_(first, middle, last [,op#])	PartialSort_
partial_sort_copy_ (first, last, res_first, res_last [,op#])	PartialSortCopy_
nth_element_(first, nth, last [,op#])	NthElement_
merge_(first1, last1, first2, last2, output [,op #])	Merge_
inplace_merge_(first, middle, last [,op#])	InplaceMerge_
set_union_(first1, last1, first2, last2, output [,op #])	SetUnion_
push_heap_(first, last [,op #])	PushHeap_
pop_heap_(first, last [,op #])	PopHeap_
make_heap_(first, last [,op #])	MakeHeap_
sort_heap_(first, last [,op #])	SortHeap_
next_permutation_(first, last [,op #])	NextPermutation_
prev_permutation_(first, last [,op #])	PrevPermutation_
inner_product_(first1, last1, first2, init [,op1#] [,op2#])	InnerProduct_
partial_sum_(first, last, output [,op#])	PartialSum_
adjacent_difference_(first, last, output [,op#])	AdjacentDifference_
replace_(first, last, old_value, new_value)	Replace_
replace_if_(first, last, op#, new_value)	ReplaceIf_
replace_copy_(first, last, result, old_value, new_value)	ReplaceCopy_
replace_copy_if_(first, last, result, op#, new_value)	ReplaceCopyIf_
rotate_(first, middle, last)#	Rotate_

Table 11.4. STL container methods

STL container methods(common) in container.hpp	**Functor**
container::reference front_(container)	Front_
container::reference back_(container)	Back_
container::iterator begin_(container)	Begin_
container::iterator end_(container)	End_
container::reverse_iterator rbegin_(container)	RBegin_
container::reverse_iterator rend_(container)	REnd_
void push_back_(container, value)	Push_Back_
void pop_back_(container, value)	Pop_Back_
void push_front_(container, value)	Push_Front_
void pop_front_(container, value)	Pop_Front_
void clear_(container)	Clear_
size_type capacity_(container)	Capacity_
size_type size_(container)	Size_
size_type max_size_(container)	Max_Size_
void reserve_(container, value)	Reserve _
void resize_(container, value)	Resize _
iterator insert_(container, pos, value)	Insert_
void insert_(container , pos, first, last)	Insert_
void insert_(container , pos, number, value)	Insert_
void swap_(container , other_container)	Swap_
void erase_(container , pos)	Erase_
void erase_(container , first, last)	Erase_
bool empty_(container)	Empty_

Table 11.5. STL list methods

std::list methods in container.hpp	**Functor**
void list_remove_(container, value)	ListRemove_
void list_remove_if_(container, op#)	ListRemove_If_
void list_merge_(container, other_list)	ListMerge_
void list_merge_(container, other_list, op#)	ListMerge_
void splice_(container, iterator, other_list)	Splice_
void splice_(container, iterator, other_list, iterator)	Splice_
void splice_(container, iterator, other_list, first, last)	Splice_
void list_reverse_(container)	ListReverse_
void list_unique_(container)	ListUnique_
void list_unique_(container, op#)	ListUnique_
void list_sort_(container)	ListSort_
void list_sort_(container, op#)	ListSort_

Table 11.6. STL associative container methods

Associative container methods in container.hpp	**Functor**
iterator insert_(container, pos, value)	Insert_
void insert_(container , first, last)	Insert_
pair<iterator, bool> insert_(container , value)	Insert_
void associative_erase_(container , pos)	Associative_Erase_
void associative_erase_(container , first, last)	Associative_Erase_
size_type associative_erase_(container , key)	Associative_Erase_
iterator associative_find_(container , key)	Associative_Find_
size_type associative_count_(container , key)	AssociativeCount_
iterator associative_lower_bound_(container , key)	Associative_Lower_Bound_
iterator associative_upper_bound_(container , key)	Associative_Upper_Bound_
pair<iterator, iterator> associative_equal_range_(container , key)	Associative_Equal_Range_

Table 11.7. STL pair

std::pair in container.hpp	**Functor**
first_type first_(pair<T1, T2>)	First_
second_type second_(pair<T1, T2>)	Second_

Table 11.8. STL string

STL string method	**std::string method in container.hpp**	**Functor**
substr (size_type pos, size_type size)	string substr_(container, pos, length)	Substr_
int compare(string)	int string_compare_(container, another_string)	StringCompare_
int compare(char*)	int string_compare_(container, another_string)	StringCompare_
int compare(size_type pos, size_type size, string)	int string_compare_(container, pos, size, another_string)	StringCompare_
int compare (size_type pos, size_type size, string, size_type length)	int string_compare_(container, pos, size, another_string, length)	StringCompare_
string& append(const string&)	string& append_(container, another_string)	Append_
string& append (charT*)	string& append_(container, another_string)	Append_
string& append (string , size_type pos, size_type size)	string& append_(container, other_string, pos, size)	Append_
string& append (charT*, size_type size)	string& append_(container, another_string, length)	Append_
string& append (size_type size, charT)	string& append_(container, size, char)	Append_

STL string method	**std::string method in container.hpp**	**Functor**
string& append (iterator begin, iterator end)	string& append_(container, begin, end)	Append_
string& insert (size_type pos, charT*)	string& string_insert_(container, pos, other_string)	StringInsert_
string& insert(size_type pos, charT*,size_type n)	string& string_insert_(container, pos, other_string, n)	StringInsert_
string& insert(size_type pos,size_type n, charT c)	string& string_insert_(container, pos, n, c)	StringInsert_
string& insert (size_type pos, const string&)	string& string_insert_(container, pos, other_string)	StringInsert_
string& insert (size_type pos, const string&, size_type pos1, size_type n)	string& string_insert_(container, pos, other_string, pos1, n)	StringInsert_
string& erase(size_type pos=0, size_type n=npos)	string& string_erase_(container, pos, n)	StringErase_
string& assign(const string&)	string& string_assign_(container, another_string)	StringAssign_
string& assign(const charT*)	string& string_assign_(container, another_string)	StringAssign_
string& assign(const string&, size_type pos, size_type n)	string& string_assign_(container, another_string, pos, n)	StringAssign_
string& assign(const charT*, size_type n)	string& string_assign_(container, another_string, n)	StringAssign_
string& assign(size_type n, charT c)	string& string_assign_(container, n, c)	StringAssign_
string& assign(iterator first, iterator last)	string& string_assign_(container, first, last)	StringAssign_
string& replace(size_type pos, size_type n, const string&)	string& string_replace_(container, pos, n, another_string)	StringReplace_
string& replace(size_type pos, size_type n, const charT*, size_type n1)	string& string_replace_(container, pos, n, another_string, n1)	StringReplace_
string& replace(size_type pos, size_type n, const charT*)	string& string_replace_(container, pos, n, another_string)	StringReplace_
string& replace(size_type pos, size_type n, size_type n1, charT c)	string& string_replace_(container, pos, n, n1, c)	StringReplace_

STL string method	**std::string method in container.hpp**	**Functor**
string& replace(iterator first, iterator last, const string&)	string& string_replace_(container, first, last, another_string)	StringReplace_
string& replace(iterator first, iterator last, const charT*, size_type n)	string& string_replace_(container, first, last, another_string, n)	StringReplace_
string& replace(iterator first, iterator last, const charT*)	string& string_replace_(container, first, last, another_string)	StringReplace_
string& replace(iterator first, iterator last, size_type n, charT c)	string& string_replace_(container, first, last, n, c)	StringReplace_
string& replace(iterator first, iterator last, iterator f, iterator l)	string& string_replace_(container, first, last, f, l)	StringReplace_
const charT* c_str()	const charT* c_str_(container)	CStr_
const charT* data()	const charT* string_data_(container)	StringData_
size_type copy(charT* buf, size_type n, size_type pos = 0)	size_type string_copy_(container, buf, n, pos); size_type string_copy_(container, buf, n)	StringCopy_
size_type find(charT* s, size_type pos, size_type n)	size_type string_find_(container, s, pos, n)	StringFind_
size_type find(charT* s, size_type pos=0)	size_type string_find_(container, s, pos); size_type string_find_(container, s)	StringFind_
size_type find(const string& s, size_type pos=0)	size_type string_find_(container, s, pos) size_type string_find_(container, s)	StringFind_
size_type find(charT c, size_type pos=0)	size_type string_find_(container, c, pos) size_type string_find_(container, c)	StringFind_
size_type rfind(charT* s, size_type pos, size_type n)	size_type string_rfind_(container, s, pos, n)	StringRFind_
size_type rfind(charT* s, size_type pos=npos)	size_type string_rfind_(container, s, pos); size_type string_rfind_(container, s)	StringRFind_
size_type rfind(const string& s, size_type pos=npos)	size_type string_rfind_(container, s, pos); size_type string_rfind_(container, s)	StringRFind_
size_type rfind(charT c, size_type pos=npos)	size_type string_rfind_(container, c, pos)	StringRFind_

STL string method	**std::string method in container.hpp**	**Functor**
	size_type string_rfind_(container, c)	
size_type find_first_of(charT* s, size_type pos, size_type n)	size_type find_first_of_(container, s, pos, n)	StringFindFirstOf_
size_type find_first_of (charT* s, size_type pos=0)	size_type find_first_of_(container, s, pos); size_type find_first_of_(container, s)	StringFindFirstOf_
size_type find_first_of (const string& s, size_type pos=0)	size_type find_first_of_(container, s, pos); size_type find_first_of_(container, s)	StringFindFirstOf_
size_type find_first_of (charT c, size_type pos=0)	size_type find_first_of_(container, c, pos) size_type find_first_of_(container, c)	StringFindFirstOf_
size_type find_first_not_of(charT* s, size_type pos, size_type n)	size_type find_first_not_of_(container, s, pos, n)	StringFindFirstNotOf_
size_type find_first_not_of (charT* s, size_type pos=0)	size_type find_first_not_of_(container, s, pos); size_type find_first_not_of_(container, s)	StringFindFirstNotOf_
size_type find_first_not_of (const string& s, size_type pos=0)	size_type find_first_not_of_(container, s, pos); size_type find_first_not_of_(container, s)	StringFindFirstNotOf_
size_type find_first_not_of (charT c, size_type pos=0)	size_type find_first_not_of_(container, c, pos); size_type find_first_not_of_(container, c)	StringFindFirstNotOf_
size_type find_last_of(charT* s, size_type pos, size_type n)	size_type find_last_of_(container, s, pos, n)	StringFindLastOf_
size_type find_last_of (charT* s, size_type pos=npos)	size_type find_last_of_(container, s, pos); size_type find_last_of_(container, s)	StringFindLastOf_
size_type find_last_of (const string& s, size_type pos=npos)	size_type find_last_of_(container, s, pos); size_type find_last_of_(container, s)	StringFindLastOf_
size_type find_last_of (charT c, size_type pos=npos)	size_type find_last_of_(container, c, pos); size_type find_last_of_(container, c)	StringFindLastOf_
size_type find_last_not_of(charT* s, size_type pos, size_type n)	size_type find_last_not_of_(container, s, pos, n)	StringFindLastNotOf_

STL string method	**std::string method in container.hpp**	**Functor**
size_type find_last_not_of (charT* s, size_type pos=npos)	size_type find_last_not_of_(container, s, pos); size_type find_last_of_(container, s)	StringFindLastNotOf_
size_type find_last_not_of (const string& s, size_type pos=npos)	size_type find_last_not_of_(container, s, pos); size_type find_last_not_of_(container, s)	StringFindLastNotOf_
size_type find_last_not_of (charT c, size_type pos=npos)	size_type find_last_not_of_(container, c, pos); size_type find_last_not_of_(container, c)	StringFindLastNotOf_

Notes:

- #: algorithms requiring a predicate need to make them eUML compatible by wrapping them inside a Predicate_ functor. For example, std::less<int> => Predicate_<std::less<int> >()
- #: If using the SGI STL implementation, these functors use the SGI return value

Name

Common headers — The common types used by front- and back-ends

msm/common.hpp

This header provides one type, wrap, which is an empty type whose only reason to exist is to be cheap to construct, so that it can be used with mpl::for_each, as shown in the Metaprogramming book, chapter 9.

```
 template <class Dummy> wrap{}; {
}
```

msm/row_tags.hpp

This header contains the row type tags which front-ends can support partially or totally. Please see the **Internals** section for a description of the different types.

Name

Back-end — The back-end headers

msm/back/state_machine.hpp

This header provides one type, state_machine, MSM's state machine engine implementation.

```
 template <class Derived,class HistoryPolicy=NoHistory,class
                              CompilePolicy=favor_runtime_speed> state_machine {
}
```

Template arguments

Derived

The name of the front-end state machine definition. All three front-ends are possible.

HistoryPolicy

The desired history. This can be: AlwaysHistory, NoHistory, ShallowHistory. Default is NoHistory.

CompilePolicy

The trade-off performance / compile-time. There are two predefined policies, favor_runtime_speed and favor_compile_time. Default is favor_runtime_speed, best performance, longer compile-time. See the backend.

methods

start

The start methods must be called before any call to process_event. It activates the entry action of the initial state(s). This allows you to choose when a state machine can start. See backend.

```
  void start();
```

process_event

The event processing method implements the double-dispatch. Each call to this function with a new event type instantiates a new dispatch algorithm and increases compile-time.

```
  template <class Event> HandledEnum
                                  process_event(Event const&);
```

current_state

Returns the ids of currently active states. You will typically need it only for debugging or logging purposes.

```
  const int* current_state const();
```

get_state_by_id

Returns the state whose id is given. As all states of a concrete state machine share a common base state, the return value is a base state. If the id corresponds to no state, a null pointer is returned.

```
  const BaseState* get_state_by_id const(int id);
```

is_contained

Helper returning true if the state machine is contained as a submachine of another state machine.

```
bool is_contained const();
```

get_state

Returns the required state of the state machine as a pointer. A compile error will occur if the state is not to be found in the state machine.

```
template <class State> State* get_state();
```

get_state

Returns the required state of the state machine as a reference. A compile error will occur if the state is not to be found in the state machine.

```
template <class State> State& get_state();
```

is_flag_active

Returns true if the given flag is currently active. A flag is active if the active state of one region is tagged with this flag (using OR as BinaryOp) or active states of all regions (using AND as BinaryOp)

```
template <class Flag,class BinaryOp> bool
                                is_flag_active();
```

is_flag_active

Returns true if the given flag is currently active. A flag is active if the active state of one region is tagged with this flag.

```
template <class Flag> bool is_flag_active();
```

visit_current_states

Visits all active states and their substates. A state is visited using the `accept` method without argument. The base class of all states must provide an `accept_sig` type.

```
void visit_current_states();
```

visit_current_states

Visits all active states and their substates. A state is visited using the `accept` method with arguments. The base class of all states must provide an `accept_sig` type defining the signature and thus the number and type of the parameters.

```
void visit_current_states(any-type param1, any-type param2,...);
```

defer_event

Defers the provided event. This method can be called only if at least one state defers an event or if the state machine provides the `activate_deferred_events`(see example [examples/Orthogonal-deferred2.cpp]) type either directly or using the deferred_events configuration of eUML (`configure_ << deferred_events`)

```
template <class Event> void defer_event(Event const&);
```

Types

nr_regions

The number of orthogonal regions contained in the state machine

entry_pt

This nested type provides the necessary typedef for entry point pseudostates. `state_machine<...>::entry_pt<state_name>` is a transition's valid target inside the containing state machine's transition table.

```
 entry_pt {
}
```

exit_pt

This nested type provides the necessary typedef for exit point pseudostates. `state_machine<...>::exit_pt<state_name>` is a transition's valid source inside the containing state machine's transition table.

```
 exit_pt {
}
```

direct

This nested type provides the necessary typedef for an explicit entry inside a submachine. `state_machine<...>::direct<state_name>` is a transition's valid target inside the containing state machine's transition table.

```
 direct {
}
```

stt

Calling state_machine<frontend>::stt returns a mpl::vector containing the transition table of the state machine. This type can then be used with generate_state_set or generate_event_set.

args.hpp

This header provides one type, args. which provides the necessary types for a visitor implementation.

msm/back/history_policies.hpp

This header provides the out-of-the-box history policies supported by MSM. There are 3 such policies.

Every history policy must implement the following methods:

set_initial_states

This method is called by msm::back::state_machine when constructed. It gives the policy a chance to save the ids of all initial states (passed as array).

```
void set_initial_states();

(int* const) ;
```

history_exit

This method is called by msm::back::state_machine when the submachine is exited. It gives the policy a chance to remember the ids of the last active substates of this submachine (passed as array).

```
void history_exit();

(int* const) ;
```

history_entry

This method is called by msm::back::state_machine when the submachine is entered. It gives the policy a chance to set the active states according to the policy's aim. The policy gets as parameter the event which activated the submachine and returns an array of active states ids.

```
template <class Event> int* const history_exit();

(Event const&) ;
```

Out-of-the-box policies:

NoHistory

This policy is the default used by state_machine. No active state of a submachine is remembered and at every new activation of the submachine, the initial state(s) are activated.

AlwaysHistory

This policy is a non-UML-standard extension. The active state(s) of a submachine is (are) always remembered at every new activation of the submachine.

ShallowHistory

This policy activates the active state(s) of a submachine if the event is found in the policy's event list.

msm/back/default_compile_policy.hpp

This header contains the definition of favor_runtime_speed. This policy has two settings:

- Submachines dispatch faster because their transitions are added into their containing machine's transition table instead of simply forwarding events.
- It solves transition conflicts at compile-time

msm/back/favor_compile_time.hpp

This header contains the definition of favor_compile_time. This policy has two settings:

- Submachines dispatch is slower because all events, even those with no dispatch chance, are forwarded to submachines. In exchange, no row is added into the containing machine's transition table, which reduces compile-time.
- It solves transition conflicts at run-time.

msm/back/metafunctions.hpp

This header contains metafunctions for use by the library. Three metafunctions can be useful for the user:

- `generate_state_set< stt >`: generates the list of all states referenced by the transition table stt. If stt is a recursive table (generated by `recursive_get_transition_table`), the metafunction finds recursively all states of the submachines. A non-recursive table can be obtained with some_backend_fsm::stt.
- `generate_event_set< stt>`: generates the list of all events referenced by the transition table stt. If stt is a recursive table (generated by `recursive_get_transition_table`), the

metafunction finds recursively all events of the submachines. A non-recursive table can be obtained with some_backend_fsm::stt.

- `recursive_get_transition_table<fsm>`: recursively extends the transition table of the state machine fsm with tables from the submachines.

msm/back/tools.hpp

This header contains a few metaprogramming tools to get some information out of a state machine.

fill_state_names

attributes

fill_state_names has for attribute:

- `char const** m_names`: an already allocated array of const char* where the typeid-generated names of a state machine states will be witten.

constructor

```
char const** names_to_fill(char const** names_to_fill);
```

usage

fill_state_names is made for use in a mpl::for_each iterating on a state list and writing inside a pre-allocated array the state names. Example:

```
typedef some_fsm::stt Stt;
typedef msm::back::generate_state_set<Stt>::type all_states; //states
static char const* state_names[mpl::size<all_states>::value];
// array to fill with names
// fill the names of the states defined in the state machine
mpl::for_each<all_states,boost::msm::wrap<mpl::placeholders::_1> >
    (msm::back::fill_state_names<Stt>(state_names));
// display all active states
for (unsigned int i=0;i<some_fsm::nr_regions::value;++i)
{
    std::cout << " -> "
              << state_names[my_fsm_instance.current_state()[i]]
              << std::endl;
}
```

get_state_name

attributes

get_state_name has for attributes:

- std::string& m_name: the return value of the iteration
- int m_state_id: the searched state's id

constructor

The constructor takes as argument a reference to the string to fill with the state name and the id which must be searched.

```
string& name_to_fill,int state_id(string& name_to_fill,int state_id);
```

usage

This type is made for the same search as in the previous example, using a mpl::for_each to iterate on states. After the iteration, the state name reference has been set.

```
// we need a fsm's table
typedef player::stt Stt;
typedef msm::back::generate_state_set<Stt>::type all_states; //all states
std::string name_of_open; // id of Open is 1
// fill name_of_open for state of id 1
boost::mpl::for_each<all_states,boost::msm::wrap<mpl::placeholders::_1> >
          (msm::back::get_state_name<Stt>(name_of_open,1));
std::cout << "typeid-generated name Open is: " <<  name_of_open << std::endl;
```

display_type

attributes

none

usage

Reusing the state list from the previous example, we can output all state names:

```
mpl::for_each<all_states,boost::msm::wrap<mpl::placeholders::_1>
>(msm::back::display_type ());
```

Name

Front-end — The front-end headers

msm/front/common_states.hpp

This header contains the predefined types to serve as base for states or state machines:

- default_base_state: non-polymorphic empty type.
- polymorphic_state: type with a virtual destructor, which makes all states polymorphic.

msm/front/completion_event.hpp

This header contains one type, `none`. This type has several meanings inside a transition table:

- as action or guard: that there is no action or guard
- as target state: that the transition is an internal transition
- as event: the transition is an anonymous (completion) transition

msm/front/functor_row.hpp

This header implements the functor front-end's transitions and helpers.

Row

definition

```
 template <class Source,class Event,class Target,class
                                        Action,class Guard> Row {
}
```

tags

row_type_tag is defined differently for every specialization:

- all 5 template parameters means a normal transition with action and guard: `typedef row_tag row_type_tag;`
- Row<Source,Event,Target,none,none> a normal transition without action or guard: `typedef _row_tag row_type_tag;`
- Row<Source,Event,Target,Action,none> a normal transition without guard: `typedef a_row_tag row_type_tag;`
- Row<Source,Event,Target,none,Guard> a normal transition without action: `typedef g_row_tag row_type_tag;`
- Row<Source,Event,none,Action,none> an internal transition without guard: `typedef a_irow_tag row_type_tag;`
- Row<Source,Event,none,none,Guard> an internal transition without action: `typedef g_irow_tag row_type_tag;`
- Row<Source,Event,none,none,Guard> an internal transition with action and guard: `typedef irow_tag row_type_tag;`
- Row<Source,Event,none,none,none> an internal transition without action or guard: `typedef _irow_tag row_type_tag;`

methods

Like any other front-end, Row implements the two necessary static functions for action and guard call. Each function receives as parameter the (deepest-level) state machine processsing the event, the event itself, the source and target states and all the states contained in a state machine.

```
template <class Fsm,class SourceState,class TargetState, class
AllStates> static void action_call();

(Fsm& fsm,Event const& evt,SourceState&,TargetState,AllStates&) ;

template <class Fsm,class SourceState,class TargetState, class
AllStates> static bool guard_call();

(Fsm& fsm,Event const& evt,SourceState&,TargetState,AllStates&) ;
```

Internal

definition

```
 template <class Event,class Action,class Guard>
                                     Internal {
}
```

tags

row_type_tag is defined differently for every specialization:

- all 3 template parameters means an internal transition with action and guard: `typedef sm_i_row_tag row_type_tag;`
- Internal<Event,none,none> an internal transition without action or guard: `typedef sm__i_row_tag row_type_tag;`
- Internal<Event,Action,none> an internal transition without guard: `typedef sm_a_i_row_tag row_type_tag;`
- Internal<Event,none,Guard> an internal transition without action: `typedef sm_g_i_row_tag row_type_tag;`

methods

Like any other front-end, Internal implements the two necessary static functions for action and guard call. Each function receives as parameter the (deepest-level) state machine processsing the event, the event itself, the source and target states and all the states contained in a state machine.

```
template <class Fsm,class SourceState,class TargetState, class
AllStates> static void action_call();

(Fsm& fsm,Event const& evt,SourceState&,TargetState,AllStates&) ;

template <class Fsm,class SourceState,class TargetState, class
AllStates> static bool guard_call();

(Fsm& fsm,Event const& evt,SourceState&,TargetState,AllStates&) ;
```

ActionSequence_

This functor calls every element of the template Sequence (which are also callable functors) in turn. It is also the underlying implementation of the eUML sequence grammar (action1,action2,...).

definition

```
 template <class Sequence> ActionSequence_ {
}
```

methods

This helper functor is made for use in a transition table and in a state behavior and therefore implements an operator() with 3 and with 4 arguments:

```
template <class Evt,class Fsm,class SourceState,class TargetState>
operator()();

Evt const& ,Fsm& ,SourceState& ,TargetState& ;

template <class Evt,class Fsm,class State> operator()();

Evt const&, Fsm&, State&;
```

Defer

definition

```
 Defer {
}
```

methods

This helper functor is made for use in a transition table and therefore implements an operator() with 4 arguments:

```
template <class Evt,class Fsm,class SourceState,class TargetState>
operator()();

Evt const&, Fsm& , SourceState&, TargetState&;
```

msm/front/internal_row.hpp

This header implements the internal transition rows for use inside an internal_transition_table. All these row types have no source or target state, as the backend will recognize internal transitions from this internal_transition_table.

methods

Like any other front-end, the following transition row types implements the two necessary static functions for action and guard call. Each function receives as parameter the (deepest-level) state machine processsing the event, the event itself, the source and target states and all the states contained in a state machine.

```
template <class Fsm,class SourceState,class TargetState, class
AllStates> static void action_call();

(Fsm& fsm,Event const& evt,SourceState&,TargetState,AllStates&) ;

template <class Fsm,class SourceState,class TargetState, class
AllStates> static bool guard_call();

(Fsm& fsm,Event const& evt,SourceState&,TargetState,AllStates&) ;
```

a_internal

definition

This is an internal transition with an action called during the transition.

```
 template< class Event, class CalledForAction, void
                                        (CalledForAction::*action)(Event const&)>
                                        a_internal {
}
```

template parameters

- Event: the event triggering the internal transition.
- CalledForAction: the type on which the action method will be called. It can be either a state of the containing state machine or the state machine itself.
- action: a pointer to the method which CalledForAction provides.

g_internal

This is an internal transition with a guard called before the transition and allowing the transition if returning true.

definition

```
 template< class Event, class CalledForGuard, bool
                                        (CalledForGuard::*guard)(Event const&)>
                                        g_internal {
}
```

template parameters

- Event: the event triggering the internal transition.
- CalledForGuard: the type on which the guard method will be called. It can be either a state of the containing state machine or the state machine itself.
- guard: a pointer to the method which CalledForGuard provides.

internal

This is an internal transition with a guard called before the transition and allowing the transition if returning true. It also calls an action called during the transition.

definition

```
 template< class Event, class CalledForAction, void
                                        (CalledForAction::*action)(Event const&), c
                                        CalledForGuard, bool (CalledForGuard::*guar
                                        internal {
}
```

template parameters

- Event: the event triggering the internal transition
- CalledForAction: the type on which the action method will be called. It can be either a state of the containing state machine or the state machine itself.

- action: a pointer to the method which CalledForAction provides.
- CalledForGuard: the type on which the guard method will be called. It can be either a state of the containing state machine or the state machine itself.
- guard: a pointer to the method which CalledForGuard provides.

_internal

This is an internal transition without action or guard. This is equivalent to an explicit "ignore event".

definition

```
 template< class Event > _internal {
}
```

template parameters

- Event: the event triggering the internal transition.

msm/front/row2.hpp

This header contains the variants of row2, which are an extension of the standard row transitions for use in the transition table. They offer the possibility to define action and guard not only in the state machine, but in any state of the state machine. They can also be used in internal transition tables through their irow2 variants.

methods

Like any other front-end, the following transition row types implements the two necessary static functions for action and guard call. Each function receives as parameter the (deepest-level) state machine processsing the event, the event itself, the source and target states and all the states contained in a state machine.

```
template <class Fsm,class SourceState,class TargetState, class
AllStates> static void action_call();

(Fsm& fsm,Event const& evt,SourceState&,TargetState,AllStates&) ;

template <class Fsm,class SourceState,class TargetState, class
AllStates> static bool guard_call();

(Fsm& fsm,Event const& evt,SourceState&,TargetState,AllStates&) ;
```

_row2

This is a transition without action or guard. The state machine only changes active state.

definition

```
 template< class Source, class Event, class Target >
                                      _row2 {
}
```

template parameters

- Event: the event triggering the transition.
- Source: the source state of the transition.
- Target: the target state of the transition.

a_row2

This is a transition with action and without guard.

definition

```
 template< class Source, class Event, class Target,
                                    {
}

 class CalledForAction, void
                                       (CalledForAction::*action)(Event const&) > _
}
```

template parameters

- Event: the event triggering the transition.
- Source: the source state of the transition.
- Target: the target state of the transition.
- CalledForAction: the type on which the action method will be called. It can be either a state of the containing state machine or the state machine itself.
- action: a pointer to the method which CalledForAction provides.

g_row2

This is a transition with guard and without action.

definition

```
 template< class Source, class Event, class Target,
                                    {
}

 class CalledForGuard, bool (CalledForGuard::*guard)(Event
                                       const&) > _row2 {
}
```

template parameters

- Event: the event triggering the transition.
- Source: the source state of the transition.
- Target: the target state of the transition.
- CalledForGuard: the type on which the guard method will be called. It can be either a state of the containing state machine or the state machine itself.
- guard: a pointer to the method which CalledForGuard provides.

row2

This is a transition with guard and action.

definition

```
 template< class Source, class Event, class Target,
```

```
                                    {
}

 class CalledForAction, void
                                        (CalledForAction::*action)(Event const&),
}

 class CalledForGuard, bool (CalledForGuard::*guard)(Event
                                        const&) > _row2 {
}
```

template parameters

- Event: the event triggering the transition.
- Source: the source state of the transition.
- Target: the target state of the transition.
- CalledForAction: the type on which the action method will be called. It can be either a state of the containing state machine or the state machine itself.
- action: a pointer to the method which CalledForAction provides.
- CalledForGuard: the type on which the guard method will be called. It can be either a state of the containing state machine or the state machine itself.
- guard: a pointer to the method which CalledForGuard provides.

a_irow2

This is an internal transition for use inside a transition table, with action and without guard.

definition

```
 template< class Source, class Event,  {
}

 class CalledForAction, void
                                        (CalledForAction::*action)(Event const&) > _
}
```

template parameters

- Event: the event triggering the transition.
- Source: the source state of the transition.
- CalledForAction: the type on which the action method will be called. It can be either a state of the containing state machine or the state machine itself.
- action: a pointer to the method which CalledForAction provides.

g_irow2

This is an internal transition for use inside a transition table, with guard and without action.

definition

```
 template< class Source, class Event,  {
```

```
}

 class CalledForGuard, bool (CalledForGuard::*guard)(Event
                                      const&) > _row2 {
}
```

template parameters

- Event: the event triggering the transition.
- Source: the source state of the transition.
- CalledForGuard: the type on which the guard method will be called. It can be either a state of the containing state machine or the state machine itself.
- guard: a pointer to the method which CalledForGuard provides.

irow2

This is an internal transition for use inside a transition table, with guard and action.

definition

```
 template< class Source, class Event,  {
}

 class CalledForAction, void
                                      (CalledForAction::*action)(Event const&),

}

 class CalledForGuard, bool (CalledForGuard::*guard)(Event
                                      const&) > _row2 {
}
```

template parameters

- Event: the event triggering the transition.
- Source: the source state of the transition.
- CalledForAction: the type on which the action method will be called. It can be either a state of the containing state machine or the state machine itself.
- action: a pointer to the method which CalledForAction provides.
- CalledForGuard: the type on which the guard method will be called. It can be either a state of the containing state machine or the state machine itself.
- guard: a pointer to the method which CalledForGuard provides.

msm/front/state_machine_def.hpp

This header provides the implementation of the **basic front-end**. It contains one type, `state_machine_def`

state_machine_def definition

This type is the basic class for a basic (or possibly any other) front-end. It provides the standard row types (which includes internal transitions) and a default implementation of the required methods and typedefs.

```
 template <class Derived,class BaseState =
                                    default_base_state> state_machine_def {
}
```

typedefs

- flag_list: by default, no flag is set in the state machine
- deferred_events: by default, no event is deferred.
- configuration: by default, no configuration customization is done.

row methods

Like any other front-end, the following transition row types implements the two necessary static functions for action and guard call. Each function receives as parameter the (deepest-level) state machine processsing the event, the event itself, the source and target states and all the states contained in a state machine (ignored).

```
template <class Fsm,class SourceState,class TargetState, class
AllStates> static void action_call();

(Fsm& fsm,Event const& evt,SourceState&,TargetState,AllStates&) ;

template <class Fsm,class SourceState,class TargetState, class
AllStates> static bool guard_call();

(Fsm& fsm,Event const& evt,SourceState&,TargetState,AllStates&) ;
```

a_row

This is a transition with action and without guard.

```
template< class Source, class Event, class Target, void
(Derived::*action)(Event const&) > a_row
```

- Event: the event triggering the transition.
- Source: the source state of the transition.
- Target: the target state of the transition.
- action: a pointer to the method provided by the concrete front-end (represented by `Derived`).

g_row

This is a transition with guard and without action.

```
template< class Source, class Event, class Target, bool
(Derived::*guard)(Event const&) > g_row
```

- Event: the event triggering the transition.
- Source: the source state of the transition.
- Target: the target state of the transition.
- guard: a pointer to the method provided by the concrete front-end (represented by `Derived`).

row

This is a transition with guard and action.

```
template< class Source, class Event, class Target, void
(Derived::*action)(Event const&), bool (Derived::*guard)(Event
const&) > row
```

- Event: the event triggering the transition.
- Source: the source state of the transition.
- Target: the target state of the transition.
- action: a pointer to the method provided by the concrete front-end (represented by `Derived`).
- guard: a pointer to the method provided by the concrete front-end (represented by `Derived`).

_row

This is a transition without action or guard. The state machine only changes active state.

```
template< class Source, class Event, class Target > _row
```

- Event: the event triggering the transition.
- Source: the source state of the transition.
- Target: the target state of the transition.

a_irow

This is an internal transition for use inside a transition table, with action and without guard.

```
template< class Source, class Event, void (Derived::*action)(Event
const&) > a_irow
```

- Event: the event triggering the transition.
- Source: the source state of the transition.
- action: a pointer to the method provided by the concrete front-end (represented by `Derived`).

g_irow

This is an internal transition for use inside a transition table, with guard and without action.

```
template< class Source, class Event, bool (Derived::*guard)(Event
const&) > g_irow
```

- Event: the event triggering the transition.
- Source: the source state of the transition.
- guard: a pointer to the method provided by the concrete front-end (represented by `Derived`).

irow

This is an internal transition for use inside a transition table, with guard and action.

```
template< class Source, class Event, void (Derived::*action)(Event
const&), bool (Derived::*guard)(Event const&) > irow
```

- Event: the event triggering the transition.
- Source: the source state of the transition.

- action: a pointer to the method provided by the concrete front-end (represented by `Derived`).
- guard: a pointer to the method provided by the concrete front-end (represented by `Derived`).

_irow

This is an internal transition without action or guard. As it does nothing, it means "ignore event".

```
template< class Source, class Event > _irow
```

- Event: the event triggering the transition.
- Source: the source state of the transition.

methods

`state_machine_def` provides a default implementation in case of an event which cannot be processed by a state machine (no transition found). The implementation is using a `BOOST_ASSERT` so that the error will only be noticed in debug mode. Overwrite this method in your implementation to change the behavior.

```
template <class Fsm,class Event> static void no_transition();

(Event const& ,Fsm&, int state) ;
```

`state_machine_def` provides a default implementation in case an exception is thrown by a state (entry/exit) or transition (action/guard) behavior. The implementation is using a `BOOST_ASSERT` so that the error will only be noticed in debug mode. Overwrite this method in your implementation to change the behavior. This method will be called only if exception handling is not deactivated (default) by defining `has_no_message_queue`.

```
template <class Fsm,class Event> static void exception_caught();

(Event const& ,Fsm&, std::exception&) ;
```

msm/front/states.hpp

This header provides the different states (except state machines) for the basic front-end (or mixed with other front-ends).

types

This header provides the following types:

no_sm_ptr

deprecated: default policy for states. It means that states do not need to save a pointer to their containing state machine.

sm_ptr

deprecated: state policy. It means that states need to save a pointer to their containing state machine. When seeing this flag, the back-end will call set_sm_ptr(fsm*) and give itself as argument.

state

Basic type for simple states. Inherit from this type to define a simple state. The first argument is needed if you want your state (and all others used in a concrete state machine) to inherit a basic type for logging or providing a common behavior.

```
template<class Base = default_base_state,class
```

```
                                        SMPtrPolicy = no_sm_ptr> state {
}
```

terminate_state

Basic type for terminate states. Inherit from this type to define a terminate state. The first argument is needed if you want your state (and all others used in a concrete state machine) to inherit a basic type for logging or providing a common behavior.

```
 template<class Base = default_base_state,class
                                        SMPtrPolicy = no_sm_ptr> terminate_state {
}
```

interrupt_state

Basic type for interrupt states. Interrupt states prevent any further event handling until EndInterruptEvent is sent. Inherit from this type to define a terminate state. The first argument is the name of the event ending the interrupt. The second argument is needed if you want your state (and all others used in a concrete state machine) to inherit a basic type for logging or providing a common behavior.

The EndInterruptEvent can also be a sequence of events: mpl::vector<EndInterruptEvent,EndInterruptEvent2>.

```
 template<class EndInterruptEvent,class Base =
                                        default_base_state, {
}

 class SMPtrPolicy = no_sm_ptr>
                                        interrupt_state {
}
```

explicit_entry

Inherit from this type in addition to the desired state type to enable this state for direct entering. The template parameter gives the region id of the state (regions are numbered in the order of the `initial_state` typedef).

```
 template <int ZoneIndex=-1> explicit_entry {
}
```

entry_pseudo_state

Basic type for entry pseudo states. Entry pseudo states are an predefined entry into a submachine and connect two transitions. The first argument is the id of the region entered by this state (regions are numbered in the order of the `initial_state` typedef). The second argument is needed if you want your state (and all others used in a concrete state machine) to inherit a basic type for logging or providing a common behavior.

```
 template<int RegionIndex=-1,class Base =
                                        default_base_state, {
}

 class SMPtrPolicy = no_sm_ptr>
                                        entry_pseudo_state {
}
```

exit_pseudo_state

Basic type for exit pseudo states. Exit pseudo states are an predefined exit from a submachine and connect two transitions. The first argument is the name of the event which will be "thrown" out of the

exit point. This event does not need to be the same as the one sent by the inner region but must be convertible from it. The second argument is needed if you want your state (and all others used in a concrete state machine) to inherit a basic type for logging or providing a common behavior.

```
 template<class Event,class Base =
                                     default_base_state, {
}

 class SMPtrPolicy = no_sm_ptr>
                                     exit_pseudo_state {
}
```

msm/front/euml/euml.hpp

This header includes all of eUML except the STL functors.

msm/front/euml/stl.hpp

This header includes all the functors for STL support in eUML. These **tables** show a full description.

msm/front/euml/algorithm.hpp

This header includes all the functors for STL algorithms support in eUML. These **tables** show a full description.

msm/front/euml/iteration.hpp

This header includes iteration functors for STL support in eUML. This **tables** shows a full description.

msm/front/euml/querying.hpp

This header includes querying functors for STL support in eUML. This **tables** shows a full description.

msm/front/euml/transformation.hpp

This header includes transformation functors for STL support in eUML. This **tables** shows a full description.

msm/front/euml/container.hpp

This header includes container functors for STL support in eUML (functors calling container methods). This **tables** shows a full description. It also provides npos for strings.

Npos_<container type>

Functor returning npos for transition or state behaviors. Like all constants, only the functor form exists, so parenthesis are necessary. Example:

```
string_find_(event_(m_song),Char_<'S'>(),Size_t_<0>())           !=
Npos_<string>() // compare result of string::find with npos
```

msm/front/euml/stt_grammar.hpp

This header provides the transition table grammars. This includes internal transition tables.

functions

build_stt

The function build_stt evaluates the grammar-conform expression as parameter. It returns a transition table, which is a mpl::vector of transitions (rows) or, if the expression is ill-formed (does not match the grammar), the type `invalid_type`, which will lead to a compile-time static assertion when this transition table is passed to a state machine.

```
template<class          Expr>          [mpl::vector<...>          /
msm::front::euml::invalid_type] build_stt();

Expr const& expr;
```

build_internal_stt

The function build_internal_stt evaluates the grammar-conform expression as parameter. It returns a transition table, which is a mpl::vector of transitions (rows) or, if the expression is ill-formed (does not match the grammar), the type `invalid_type`, which will lead to a compile-time static assertion when this transition table is passed to a state machine.

```
template<class          Expr>          [mpl::vector<...>          /
msm::front::euml::invalid_type] build_internal_stt();

Expr const& expr;
```

grammars

transition table

The transition table accepts the following grammar:

```
Stt := Row | (Stt ',' Stt)
Row := (Target '==' (SourcePlusEvent)) /* first syntax*/
       | ( (SourcePlusEvent) '==' Target ) /* second syntax*/
       | (SourcePlusEvent) /* internal transitions */
SourcePlusEvent := (BuildSource '+' BuildEvent)/* standard transition*/
                   | (BuildSource) /* anonymous transition */
BuildSource := state_tag | (state_tag '/' Action) | (state_tag '[' Guard ']')
            | (state_tag '[' Guard ']' '/' Action)
BuildEvent := event_tag | (event_tag '/' Action) | (event_tag '[' Guard ']')
            | (event_tag '[' Guard ']' '/' Action)
```

The grammars Action and Guard are defined in state_grammar.hpp and guard_grammar.hpp respectively. state_tag and event_tag are inherited from euml_state (or other state variants) and euml_event respectively. For example, following declarations are possible:

```
target == source + event [guard] / action,
source + event [guard] / action == target,
source + event [guard] / (action1,action2) == target,
target == source + event [guard] / (action1,action2),
target == source + event,
source + event == target,
target == source + event [guard],
source + event [guard] == target,
target == source + event / action,
source + event /action == target,
source / action == target, /*anonymous transition*/
target == source / action, /*anonymous transition*/
source + event /action, /* internal transition*/
```

internal transition table

The internal transition table accepts the following grammar:

```
IStt := BuildEvent | (IStt ',' IStt)
```

BuildEvent being defined for both internal and standard transition tables.

msm/front/euml/guard_grammar.hpp

This header contains the `Guard` grammar used in the previous section. This grammar is long but pretty simple:

```
Guard := action_tag | (Guard '&&' Guard)
        | (Guard '||' Guard) | ... /* operators*/
        | (if_then_else_(Guard,Guard,Guard)) | (function (Action,...Action))
```

Most C++ operators are supported (address-of is not). With `function` is meant any eUML predefined function or any self-made (using `MSM_EUML_METHOD` or `MSM_EUML_FUNCTION`). Action is a grammar defined in state_grammar.hpp.

msm/front/euml/state_grammar.hpp

This header provides the grammar for actions and the different grammars and functions to build states using eUML.

action grammar

Like the guard grammar, this grammar supports relevant C++ operators and eUML functions:

```
Action := action_tag | (Action '+' Action)
          | ('--' Action) | ... /* operators*/
          | if_then_else_(Guard,Action,Action) | if_then_(Action)
          | while_(Guard,Action)
          | do_while_(Guard,Action) | for_(Action,Guard,Action,Action)
          | (function(Action,...Action))
ActionSequence := Action | (Action ',' Action)
```

Relevant operators are: ++ (post/pre), -- (post/pre), dereferencing, + (unary/binary), - (unary/binary), *, /, %, &(bitwise), | (bitwise), ^(bitwise), +=, -=, *=, /=, %=, <<=, >>=, <<, >>, =, [].

attributes

This grammar is used to add attributes to states (or state machines) or events: It evaluates to a fusion::map. You can use two forms:

- `attributes_ << no_attributes_`
- `attributes_ << attribute_1 << ... << attribute_n`

Attributes can be of any default-constructible type (fusion requirement).

configure

This grammar also has two forms:

- `configure_ << no_configure_`
- `configure_ << type_1 << ... << type_n`

This grammar is used to create inside one syntax:

- flags: `configure_ << some_flag` where some_flag inherits from `euml_flag<some_flag>` or is defined using BOOST_MSM_EUML_FLAG.
- deferred events: `configure_ << some_event` where some_event inherits from `euml_event<some_event>` or is defined using BOOST_MSM_EUML_EVENT or BOOST_MSM_EUML_EVENT_WITH_ATTRIBUTES.
- configuration (message queue, manual deferring, exception handling): `configure_ << some_config` where some_config inherits from `euml_config<some_config>`. At the moment, three predefined objects exist (in msm//front/euml/common.hpp):
  - no_exception: disable catching exceptions
  - no_msg_queue: disable message queue
  - deferred_events: manually enable handling of deferred events

initial states

The grammar to define initial states for a state machine is: `init_ << state_1 << ... << state_n` where state_1...state_n inherit from euml_state or is defined using BOOST_MSM_EUML_STATE, BOOST_MSM_EUML_INTERRUPT_STATE, BOOST_MSM_EUML_TERMINATE_STATE, BOOST_MSM_EUML_EXPLICIT_ENTRY_STATE, BOOST_MSM_EUML_ENTRY_STATE or BOOST_MSM_EUML_EXIT_STATE.

functions

build_sm

This function has several overloads. The return type is not relevant to you as only decltype (return type) is what one needs.

Defines a state machine without entry or exit:

```
template <class StateNameTag,class Stt,class Init>
func_state_machine<...> build_sm();

Stt ,Init;
```

Defines a state machine with entry behavior:

```
template <class StateNameTag,class Stt,class Init,class Expr1>
func_state_machine<...> build_sm();

Stt ,Init,Expr1 const&;
```

Defines a state machine with entry and exit behaviors:

```
template <class StateNameTag,class Stt,class Init,class Expr1, class
Expr2> func_state_machine<...> build_sm();

Stt ,Init,Expr1 const&,Expr2 const&;
```

Defines a state machine with entry, exit behaviors and attributes:

```
template <class StateNameTag,class Stt,class Init,class Expr1, class
Expr2, class Attributes> func_state_machine<...> build_sm();

Stt ,Init,Expr1 const&, Expr2 const&, Attributes const&;
```

Defines a state machine with entry, exit behaviors, attributes and configuration (deferred events, flags):

```
template <class StateNameTag,class Stt,class Init,class Expr1, class
Expr2, class Attributes, class Configure> func_state_machine<...>
build_sm();

Stt ,Init,Expr1 const&, Expr2 const&, Attributes const&, Configure
const&;
```

Defines a state machine with entry, exit behaviors, attributes, configuration (deferred events, flags) and a base state:

```
template <class StateNameTag,class Stt,class Init,class Expr1,
class Expr2, class Attributes, class Configure, class Base>
func_state_machine<...> build_sm();

Stt ,Init,Expr1 const&, Expr2 const&, Attributes const&, Configure
const&, Base;
```

Notice that this function requires the extra parameter class StateNameTag to disambiguate state machines having the same parameters but still being different.

build_state

This function has several overloads. The return type is not relevant to you as only decltype (return type) is what one needs.

Defines a simple state without entry or exit:

```
func_state<class StateNameTag,...> build_state();

;
```

Defines a simple state with entry behavior:

```
template <class StateNameTag,class Expr1> func_state<...>
build_state();

Expr1 const&;
```

Defines a simple state with entry and exit behaviors:

```
template <class StateNameTag,class Expr1, class Expr2>
func_state<...> build_state();

Expr1 const&,Expr2 const&;
```

Defines a simple state with entry, exit behaviors and attributes:

```
template <class StateNameTag,class Expr1, class Expr2, class
Attributes> func_state<...> build_state();

Expr1 const&, Expr2 const&, Attributes const&;
```

Defines a simple state with entry, exit behaviors, attributes and configuration (deferred events, flags):

```
template <class StateNameTag,class Expr1, class Expr2, class
Attributes, class Configure> func_state<...> build_state();

Expr1 const&, Expr2 const&, Attributes const&, Configure const&;
```

Defines a simple state with entry, exit behaviors, attributes, configuration (deferred events, flags) and a base state:

```
template <class StateNameTag,class Expr1, class Expr2, class
Attributes, class Configure, class Base> func_state<...>
build_state();

Expr1 const&, Expr2 const&, Attributes const&, Configure const&,
Base;
```

Notice that this function requires the extra parameter class StateNameTag to disambiguate states having the same parameters but still being different.

build_terminate_state

This function has the same overloads as build_state.

build_interrupt_state

This function has several overloads. The return type is not relevant to you as only decltype (return type) is what one needs.

Defines an interrupt state without entry or exit:

```
template <class StateNameTag,class EndInterruptEvent>
func_state<...> build_interrupt_state();

EndInterruptEvent const&;
```

Defines an interrupt state with entry behavior:

```
template <class StateNameTag,class EndInterruptEvent,class Expr1>
func_state<...> build_interrupt_state();

EndInterruptEvent const&,Expr1 const&;
```

Defines an interrupt state with entry and exit behaviors:

```
template <class StateNameTag,class EndInterruptEvent,class Expr1,
class Expr2> func_state<...> build_interrupt_state();

EndInterruptEvent const&,Expr1 const&,Expr2 const&;
```

Defines an interrupt state with entry, exit behaviors and attributes:

```
template <class StateNameTag,class EndInterruptEvent,class
Expr1, class Expr2, class Attributes> func_state<...>
build_interrupt_state();

EndInterruptEvent const&,Expr1 const&, Expr2 const&, Attributes
const&;
```

Defines an interrupt state with entry, exit behaviors, attributes and configuration (deferred events, flags):

```
template <class StateNameTag,class EndInterruptEvent,class Expr1,
class Expr2, class Attributes, class Configure> func_state<...>
build_interrupt_state();

EndInterruptEvent const&,Expr1 const&, Expr2 const&, Attributes
const&, Configure const&;
```

Defines an interrupt state with entry, exit behaviors, attributes, configuration (deferred events, flags) and a base state:

```
template <class StateNameTag,class EndInterruptEvent,class Expr1,
class Expr2, class Attributes, class Configure, class Base>
func_state<...> build_interrupt_state();

EndInterruptEvent const&,Expr1 const&, Expr2 const&, Attributes
const&, Configure const&, Base;
```

Notice that this function requires the extra parameter class StateNameTag to disambiguate states having the same parameters but still being different.

build_entry_state

This function has several overloads. The return type is not relevant to you as only decltype (return type) is what one needs.

Defines an entry pseudo state without entry or exit:

```
template <class StateNameTag,int RegionIndex> entry_func_state<...>
build_entry_state();

;
```

Defines an entry pseudo state with entry behavior:

```
template <class StateNameTag,int RegionIndex,class Expr1>
entry_func_state<...> build_entry_state();

Expr1 const&;
```

Defines an entry pseudo state with entry and exit behaviors:

```
template <class StateNameTag,int RegionIndex,class Expr1, class
Expr2> entry_func_state<...> build_entry_state();

Expr1 const&,Expr2 const&;
```

Defines an entry pseudo state with entry, exit behaviors and attributes:

```
template <class StateNameTag,int RegionIndex,class Expr1, class
Expr2, class Attributes> entry_func_state<...> build_entry_state();

Expr1 const&, Expr2 const&, Attributes const&;
```

Defines an entry pseudo state with entry, exit behaviors, attributes and configuration (deferred events, flags):

```
template <class StateNameTag,int RegionIndex,class Expr1, class
Expr2, class Attributes, class Configure> entry_func_state<...>
build_entry_state();

Expr1 const&, Expr2 const&, Attributes const&, Configure const&;
```

Defines an entry pseudo state with entry, exit behaviors, attributes, configuration (deferred events, flags) and a base state:

```
template <class StateNameTag,int RegionIndex,class Expr1, class
Expr2, class Attributes, class Configure, class Base>
entry_func_state<...> build_entry_state();

Expr1 const&, Expr2 const&, Attributes const&, Configure const&,
Base;
```

Notice that this function requires the extra parameter class StateNameTag to disambiguate states having the same parameters but still being different.

build_exit_state

This function has several overloads. The return type is not relevant to you as only decltype (return type) is what one needs.

Defines an exit pseudo state without entry or exit:

```
template <class StateNameTag,class Event> exit_func_state<...>
build_exit_state();

Event const&;
```

Defines an exit pseudo state with entry behavior:

```
template <class StateNameTag,class Event,class Expr1>
exit_func_state<...> build_exit_state();

Event const&,Expr1 const&;
```

Defines an exit pseudo state with entry and exit behaviors:

```
template <class StateNameTag,class Event,class Expr1, class Expr2>
exit_func_state<...> build_exit_state();

Event const&,Expr1 const&,Expr2 const&;
```

Defines an exit pseudo state with entry, exit behaviors and attributes:

```
template <class StateNameTag,class Event,class Expr1, class Expr2,
class Attributes> exit_func_state<...> build_exit_state();

Event const&,Expr1 const&, Expr2 const&, Attributes const&;
```

Defines an exit pseudo state with entry, exit behaviors, attributes and configuration (deferred events, flags):

```
template <class StateNameTag,class Event,class Expr1, class
Expr2, class Attributes, class Configure> exit_func_state<...>
build_exit_state();

Event const&,Expr1 const&, Expr2 const&, Attributes const&, Configure
const&;
```

Defines an exit pseudo state with entry, exit behaviors, attributes, configuration (deferred events, flags) and a base state:

```
template <class StateNameTag,class Event,class Expr1, class Expr2,
class Attributes, class Configure, class Base> exit_func_state<...>
build_exit_state();

Event const&,Expr1 const&, Expr2 const&, Attributes const&, Configure
const&, Base;
```

Notice that this function requires the extra parameter class StateNameTag to disambiguate states having the same parameters but still being different.

build_explicit_entry_state

This function has the same overloads as build_entry_state and explicit_entry_func_state as return type.

msm/front/euml/common.hpp

types

euml_event

The basic type for events with eUML.

```
 template <class EventName> euml_event; {
}

struct play : euml_event<play>{};
```

euml_state

The basic type for states with eUML. You will usually not use this type directly as it is easier to use BOOST_MSM_EUML_STATE, BOOST_MSM_EUML_INTERRUPT_STATE, BOOST_MSM_EUML_TERMINATE_STATE, BOOST_MSM_EUML_EXPLICIT_ENTRY_STATE, BOOST_MSM_EUML_ENTRY_STATE or BOOST_MSM_EUML_EXIT_STATE.

```
 template <class StateName> euml_state; {
}
```

You can however use this type directly if you want to provide your state with extra functions or provide entry or exit behaviors without functors, for example:

```
struct Empty : public msm::front::state<> , public euml_state<Empty>
{
    void foo() {...}
    template <class Event,class Fsm>
    void on_entry(Event const& evt,Fsm& fsm){...}
};
```

euml_flag

The basic type for flags with eUML.

```
 template <class FlagName> euml_flag; {
}

struct PlayingPaused: euml_flag<PlayingPaused>{};
```

euml_action

The basic type for state or transition behaviors and guards with eUML.

```
 template <class AcionName> euml_action; {
}

struct close_drawer : euml_action<close_drawer>
{
    template <class Fsm,class Evt,class SourceState,class TargetState>
    void operator()(Evt const& , Fsm&, SourceState& ,TargetState& ) {...}
};
```

Or, as state entry or exit behavior:

```
struct Playing_Entry : euml_action<Playing_Entry>
```

```
{
    template <class Event,class Fsm,class State>
    void operator()(Event const&,Fsm& fsm,State& ){...}
};
```

euml_config

The basic type for configuration possibilities with eUML.

```
 template <class ConfigName> euml_config; {
}
```

You normally do not use this type directly but instead the instances of predefined configuration:

- no_exception: disable catching exceptions
- no_msg_queue: disable message queue. The message queue allows you to send an event for procesing while in an event processing.
- deferred_events: manually enable handling of deferred events

invalid_type

Type returned by grammar parsers if the grammar is invalid. Seeing this type will result in a static assertion.

no_action

Placeholder type for use in entry/exit or transition behaviors, which does absolutely nothing.

source_

Generic object or function for the source state of a given transition:

- as object: returns by reference the source state of a transition, usually to be used by another function (usually one created by MSM_EUML_METHOD or MSM_EUML_FUNCTION).

  Example:

  ```
  some_user_function_(source_)
  ```

- as function: returns by reference the attribute passed as parameter.

  Example:

  ```
  source_(m_counter)++
  ```

target_

Generic object or function for the target state of a given transition:

- as object: returns by reference the target state of a transition, usually to be used by another function (usually one created by MSM_EUML_METHOD or MSM_EUML_FUNCTION).

  Example:

  ```
  some_user_function_(target_)
  ```

- as function: returns by reference the attribute passed as parameter.

  Example:

```
target_(m_counter)++
```

state_

Generic object or function for the state of a given entry / exit behavior. state_ means source_ while in the context of an exit behavior and target_ in the context of an entry behavior:

- as object: returns by reference the current state, usually to be used by another function (usually one created by MSM_EUML_METHOD or MSM_EUML_FUNCTION).

  Example:

  ```
  some_user_function_(state_) // calls some_user_function on the current state
  ```

- as function: returns by reference the attribute passed as parameter.

  Example:

  ```
  state_(m_counter)++
  ```

event_

Generic object or function for the event triggering a given transition (valid in a transition behavior, as well as in state entry/exit behaviors):

- as object: returns by reference the event of a transition, usually to be used by another function (usually one created by MSM_EUML_METHOD or MSM_EUML_FUNCTION).

  Example:

  ```
  some_user_function_(event_)
  ```

- as function: returns by reference the attribute passed as parameter.

  Example:

  ```
  event_(m_counter)++
  ```

fsm_

Generic object or function for the state machine containing a given transition:

- as object: returns by reference the event of a transition, usually to be used by another function (usually one created by MSM_EUML_METHOD or MSM_EUML_FUNCTION).

  Example:

  ```
  some_user_function_(fsm_)
  ```

- as function: returns by reference the attribute passed as parameter.

  Example:

  ```
  fsm_(m_counter)++
  ```

substate_

Generic object or function returning a state of a given state machine:

- with 1 parameter: returns by reference the state passed as parameter, usually to be used by another function (usually one created by MSM_EUML_METHOD or MSM_EUML_FUNCTION).

Example:

```
some_user_function_(substate_(my_state))
```

- with 2 parameters: returns by reference the state passed as first parameter from the state machine passed as second parameter, usually to be used by another function (usually one created by MSM_EUML_METHOD or MSM_EUML_FUNCTION). This makes sense when used in combination with attribute_.

  Example (equivalent to the previous example):

```
some_user_function_(substate_(my_state,fsm_))
```

attribute_

Generic object or function returning the attribute passed (by name) as second parameter of the thing passed as first (a state, event or state machine). Example:

```
attribute_(substate_(my_state),cd_name_attribute)++
```

True_

Functor returning true for transition or state behaviors. Like all constants, only the functor form exists, so parenthesis are necessary. Example:

```
if_then_(True_(),/* some action always called*/)
```

False_

Functor returning false for transition or state behaviors. Like all constants, only the functor form exists, so parenthesis are necessary. Example:

```
if_then_(False_(),/* some action never called */)
```

Int_<int value>

Functor returning an integer value for transition or state behaviors. Like all constants, only the functor form exists, so parenthesis are necessary. Example:

```
target_(m_ringing_cpt) = Int_<RINGING_TIME>() // RINGING_TIME is a constant
```

Char_<char value>

Functor returning a char value for transition or state behaviors. Like all constants, only the functor form exists, so parenthesis are necessary. Example:

```
// look for 'S' in event.m_song
[string_find_(event_(m_song),Char_<'S'>(),Size_t_<0>()) != Npos_<string>()]
```

Size_t_<size_t value>

Functor returning a size_t value for transition or state behaviors. Like all constants, only the functor form exists, so parenthesis are necessary. Example:

```
substr_(event_(m_song),Size_t_<1>()) // returns a substring of event.m_song
```

String_ < mpl::string >

Functor returning a string for transition or state behaviors. Like all constants, only the functor form exists, so parenthesis are necessary. Requires boost >= 1.40 for mpl::string.

Example:

```
// adds "Let it be" to fsm.m_src_container
push_back_(fsm_(m_src_container), String_<mpl::string<'Let','it ','be'> >())
```

Predicate_ < some_stl_compatible_functor >

This functor eUML-enables a STL functor (for use in an algorithm). This is necessary because all what is in the transition table must be a eUML terminal.

Example:

```
//equivalent to:
//std::accumulate(fsm.m_vec.begin(),fsm.m_vec.end(),1,std::plus<int>())== 1
accumulate_(begin_(fsm_(m_vec)),end_(fsm_(m_vec)),Int_<1>(),
            Predicate_<std::plus<int> >()) == Int_<1>())
```

process_

This function sends an event to up to 4 state machines by calling `process_event` on them:

- `process_(some_event)` : processes an event in the current (containing) state machine.
- `process_(some_event [,fsm1...fsm4] )` : processes the same event in the 1-4 state machines passed as argument.

process2_

This function sends an event to up to 3 state machines by calling `process_event` on them and copy-constructing the event from the data passed as second parameter:

- `process2_(some_event, some_data)` : processes an event in the current (containing) state machine.
- `process2_(some_event, some_data [,fsm1...fsm3] )` : processes the same event in the 1-3 state machines passed as argument.

Example:

```
// processes NotFound on current state machine,
// copy-constructed with event.m_song
process2_(NotFound,event_(m_song))
```

With the following definitions:

```
BOOST_MSM_EUML_DECLARE_ATTRIBUTE(std::string,m_song)//declaration of m_song
NotFound (const string& data) // copy-constructor of NotFound
```

is_flag_

This function tells if a flag is active by calling `is_flag_active` on the current state machine or one passed as parameter:

- `is_flag_(some_flag)` : calls `is_flag_active` on the current (containing) state machine.
- `is_flag_(some_flag, some_fsm)` :calls `is_flag_active` on the state machine.passed as argument.

defer_

This object defers the current event by calling `defer_event` on the current state machine. Example:

```
Empty() + play() / defer_
```

explicit_(submachine-name,state-name)

Used as transition's target, causes an explicit entry into the given state from the given submachine. Several explicit_ as targets, separated by commas, means a fork. The state must have been declared as such using BOOST_MSM_EUML_EXPLICIT_ENTRY_STATE.

entry_pt_(submachine-name,state-name)

Used as transition's target from a containing state machine, causes submachine-name to be entered using the given entry pseudo-state. This state must have been declared as pseudo entry using BOOST_MSM_EUML_ENTRY_STATE.

exit_pt_(submachine-name,state-name)

Used as transition's source from a containing state machine, causes submachine-name to be left using the given exit pseudo-state. This state must have been declared as pseudo exit using BOOST_MSM_EUML_EXIT_STATE.

MSM_EUML_FUNCTION

This macro creates a eUML function and a functor for use with the functor front-end, based on a free function:

- first parameter: the name of the functor
- second parameter: the underlying function
- third parameter: the eUML function name
- fourth parameter: the return type if used in a transition behavior
- fifth parameter: the return type if used in a state behavior (entry/exit)

Note that the function itself can take up to 5 arguments.

Example:

```
MSM_EUML_FUNCTION(BinarySearch_,std::binary_search,binary_search_,bool,bool)
```

Can be used like:

```
binary_search_(begin_(fsm_(m_var)),end_(fsm_(m_var)),Int_<9>())
```

MSM_EUML_METHOD

This macro creates a eUML function and a functor for use with the functor front-end, based on a method:

- first parameter: the name of the functor
- second parameter: the underlying function
- third parameter: the eUML function name
- fourth parameter: the return type if used in a transition behavior
- fifth parameter: the return type if used in a state behavior (entry/exit)

Note that the method itself can take up to 4 arguments (5 like for a free function - 1 for the object on which the method is called).

Example:

```
struct Empty : public msm::front::state<> , public euml_state<Empty>
{
     void activate_empty() {std::cout << "switching to Empty " << std::endl;}
...
};
MSM_EUML_METHOD(ActivateEmpty_,activate_empty,activate_empty_,void,void)
```

Can be used like:

```
Empty == Open + open_close / (close_drawer , activate_empty_(target_))
```

BOOST_MSM_EUML_ACTION(action-instance-name)

This macro declares a behavior type and a const instance for use in state or transition behaviors. The action implementation itself follows the macro declaration, for example:

```
BOOST_MSM_EUML_ACTION(good_disk_format)
{
     template <class Fsm,class Evt,class SourceState,class TargetState>
     void/bool operator()(Evt const& evt,Fsm&,SourceState& ,TargetState& ){...}
};
```

BOOST_MSM_EUML_FLAG(flag-instance-name)

This macro declares a flag type and a const instance for use in behaviors.

BOOST_MSM_EUML_FLAG_NAME(flag-instance-name)

This macro returns the name of the flag type generated by BOOST_MSM_EUML_FLAG. You need this where the type is required (usually with the back-end method is_flag_active). For example:

```
fsm.is_flag_active<BOOST_MSM_EUML_FLAG_NAME(CDLoaded)>()
```

BOOST_MSM_EUML_DECLARE_ATTRIBUTE(event-type,event-name)

This macro declares an attribute called event-name of type event-type. This attribute can then be made part of an attribute list using BOOST_MSM_EUML_ATTRIBUTES.

BOOST_MSM_EUML_ATTRIBUTES(attributes-expression,attributes-name)

This macro declares an attribute list called attributes-name based on the expression as first argument. These attributes can then be made part of an event using BOOST_MSM_EUML_EVENT_WITH_ATTRIBUTES, of a state as 3rd parameter of BOOST_MSM_EUML_STATE or of a state machine as 5th parameter of BOOST_MSM_EUML_DECLARE_STATE_MACHINE.

Attributes are added using left-shift, for example:

```
// m_song is of type std::string
BOOST_MSM_EUML_DECLARE_ATTRIBUTE(std::string,m_song)
// contains one attribute, m_song
BOOST_MSM_EUML_ATTRIBUTES((attributes_ << m_song ), FoundDef)
```

BOOST_MSM_EUML_EVENT(event-instance name)

This macro defines an event type (event-instance-name_helper) and declares a const instance of this event type called event-instance-name for use in a transition table or state behaviors.

BOOST_MSM_EUML_EVENT_WITH_ATTRIBUTES(event-instance-name,attributes)

This macro defines an event type (event-instance-name_helper) and declares a const instance of this event type called event-instance-name for use in a transition table or state behaviors. The event will have as attributes the ones passed by the second argument:

```
BOOST_MSM_EUML_EVENT_WITH_ATTRIBUTES(Found,FoundDef)
```

The created event instance supports operator()(attributes) so that

```
my_back_end.process_event(Found(some_string))
```

is possible.

BOOST_MSM_EUML_EVENT_NAME(event-instance-name)

This macro returns the name of the event type generated by BOOST_MSM_EUML_EVENT or BOOST_MSM_EUML_EVENT_WITH_ATTRIBUTES. You need this where the type is required (usually inside a back-end definition). For example:

```
typedef msm::back::state_machine<Playing_,
msm::back::ShallowHistory<mpl::vector<BOOST_MSM_EUML_EVENT_NAME(end_pause)
> > > Playing_type;
```

BOOST_MSM_EUML_STATE(build-expression,state-instance-name)

This macro defines a state type (state-instance-name_helper) and declares a const instance of this state type called state-instance-name for use in a transition table or state behaviors.

There are several possibilitites for the expression syntax:

- (): state without entry or exit action.
- (Expr1): state with entry but no exit action.
- (Expr1,Expr2): state with entry and exit action.
- (Expr1,Expr2,Attributes): state with entry and exit action, defining some attributes.
- (Expr1,Expr2,Attributes,Configure): state with entry and exit action, defining some attributes and flags (standard MSM flags) or deferred events (standard MSM deferred events).
- (Expr1,Expr2,Attributes,Configure,Base): state with entry and exit action, defining some attributes, flags and deferred events (plain msm deferred events) and a non-default base state (as defined in standard MSM).

BOOST_MSM_EUML_INTERRUPT_STATE(build-expression,state-instance-name)

This macro defines an interrupt state type (state-instance-name_helper) and declares a const instance of this state type called state-instance-name for use in a transition table or state behaviors.

There are several possibilitites for the expression syntax. In all of them, the first argument is the name of the event (generated by one of the previous macros) ending the interrupt:

- (end_interrupt_event): interrupt state without entry or exit action.
- (end_interrupt_event,Expr1): interrupt state with entry but no exit action.

- (end_interrupt_event,Expr1,Expr2): interrupt state with entry and exit action.
- (end_interrupt_event,Expr1,Expr2,Attributes): interrupt state with entry and exit action, defining some attributes.
- (end_interrupt_event,Expr1,Expr2,Attributes,Configure): interrupt state with entry and exit action, defining some attributes and flags (standard MSM flags) or deferred events (standard MSM deferred events).
- (end_interrupt_event,Expr1,Expr2,Attributes,Configure,Base): interrupt state with entry and exit action, defining some attributes, flags and deferred events (plain msm deferred events) and a non-default base state (as defined in standard MSM).

BOOST_MSM_EUML_TERMINATE_STATE(build-expression,state-instance-name)

This macro defines a terminate pseudo-state type (state-instance-name_helper) and declares a const instance of this state type called state-instance-name for use in a transition table or state behaviors.

There are several possibilitites for the expression syntax:

- (): terminate pseudo-state without entry or exit action.
- (Expr1): terminate pseudo-state with entry but no exit action.
- (Expr1,Expr2): terminate pseudo-state with entry and exit action.
- (Expr1,Expr2,Attributes): terminate pseudo-state with entry and exit action, defining some attributes.
- (Expr1,Expr2,Attributes,Configure): terminate pseudo-state with entry and exit action, defining some attributes and flags (standard MSM flags) or deferred events (standard MSM deferred events).
- (Expr1,Expr2,Attributes,Configure,Base): terminate pseudo-state with entry and exit action, defining some attributes, flags and deferred events (plain msm deferred events) and a non-default base state (as defined in standard MSM).

BOOST_MSM_EUML_EXIT_STATE(build-expression,state-instance-name)

This macro defines an exit pseudo-state type (state-instance-name_helper) and declares a const instance of this state type called state-instance-name for use in a transition table or state behaviors.

There are several possibilitites for the expression syntax:

- (forwarded_event):exit pseudo-state without entry or exit action.
- (forwarded_event,Expr1): exit pseudo-state with entry but no exit action.
- (forwarded_event,Expr1,Expr2): exit pseudo-state with entry and exit action.
- (forwarded_event,Expr1,Expr2,Attributes): exit pseudo-state with entry and exit action, defining some attributes.
- (forwarded_event,Expr1,Expr2,Attributes,Configure): exit pseudo-state with entry and exit action, defining some attributes and flags (standard MSM flags) or deferred events (standard MSM deferred events).
- (forwarded_event,Expr1,Expr2,Attributes,Configure,Base): exit pseudo-state with entry and exit action, defining some attributes, flags and deferred events (plain msm deferred events) and a non-default base state (as defined in standard MSM).

Note that the forwarded_event must be constructible from the event sent by the submachine containing the exit point.

BOOST_MSM_EUML_ENTRY_STATE(int region-index,build-expression,state-instance-name)

This macro defines an entry pseudo-state type (state-instance-name_helper) and declares a const instance of this state type called state-instance-name for use in a transition table or state behaviors.

There are several possibilitites for the expression syntax:

- (): entry pseudo-state without entry or exit action.
- (Expr1): entry pseudo-state with entry but no exit action.
- (Expr1,Expr2): entry pseudo-state with entry and exit action.
- (Expr1,Expr2,Attributes): entry pseudo-state with entry and exit action, defining some attributes.
- (Expr1,Expr2,Attributes,Configure): entry pseudo-state with entry and exit action, defining some attributes and flags (standard MSM flags) or deferred events (standard MSM deferred events).
- (Expr1,Expr2,Attributes,Configure,Base): entry pseudo-state with entry and exit action, defining some attributes, flags and deferred events (plain msm deferred events) and a non-default base state (as defined in standard MSM).

BOOST_MSM_EUML_EXPLICIT_ENTRY_STATE(int region-index,build-expression,state-instance-name)

This macro defines a submachine's substate type (state-instance-name_helper), which can be explicitly entered and also declares a const instance of this state type called state-instance-name for use in a transition table or state behaviors.

There are several possibilitites for the expression syntax:

- (): state without entry or exit action.
- (Expr1): state with entry but no exit action.
- (Expr1,Expr2): state with entry and exit action.
- (Expr1,Expr2,Attributes): state with entry and exit action, defining some attributes.
- (Expr1,Expr2,Attributes,Configure): state with entry and exit action, defining some attributes and flags (standard MSM flags) or deferred events (standard MSM deferred events).
- (Expr1,Expr2,Attributes,Configure,Base): state with entry and exit action, defining some attributes, flags and deferred events (plain msm deferred events) and a non-default base state (as defined in standard MSM).

BOOST_MSM_EUML_STATE_NAME(state-instance-name)

This macro returns the name of the state type generated by BOOST_MSM_EUML_STATE or other state macros. You need this where the type is required (usually using a backend function). For example:

```
fsm.get_state<BOOST_MSM_EUML_STATE_NAME(StringFind)&>().some_state_function();
```

BOOST_MSM_EUML_DECLARE_STATE(build-expression,state-instance-name)

Like BOOST_MSM_EUML_STATE but does not provide an instance, simply a type declaration.

BOOST_MSM_EUML_DECLARE_INTERRUPT_STATE(build-expression,state-instance-name)

Like BOOST_MSM_EUML_INTERRUPT_STATE but does not provide an instance, simply a type declaration.

BOOST_MSM_EUML_DECLARE_TERMINATE_STATE(build-expression,state-instance-name)

Like BOOST_MSM_EUML_TERMINATE_STATE but does not provide an instance, simply a type declaration.

BOOST_MSM_EUML_DECLARE_EXIT_STATE(build-expression,state-instance-name)

Like BOOST_MSM_EUML_EXIT_STATE but does not provide an instance, simply a type declaration.

BOOST_MSM_EUML_DECLARE_ENTRY_STATE(int region-index,build-expression,state-instance-name)

Like BOOST_MSM_EUML_ENTRY_STATE but does not provide an instance, simply a type declaration.

BOOST_MSM_EUML_DECLARE_EXPLICIT_ENTRY_STATE(int region-index,build-expression,state-instance-name)

Like BOOST_MSM_EUML_EXPLICIT_ENTRY_STATE but does not provide an instance, simply a type declaration.

BOOST_MSM_EUML_TRANSITION_TABLE(expression, table-instance-name)

This macro declares a transition table type and also declares a const instance of the table which can then be used in a state machine declaration (see BOOST_MSM_EUML_DECLARE_STATE_MACHINE).The expression must follow the **transition table grammar**.

BOOST_MSM_EUML_DECLARE_TRANSITION_TABLE(iexpression,table-instance-name)

Like BOOST_MSM_EUML_TRANSITION_TABLE but does not provide an instance, simply a type declaration.

BOOST_MSM_EUML_INTERNAL_TRANSITION_TABLE(expression, table-instance-name)

This macro declares a transition table type and also declares a const instance of the table.The expression must follow the **transition table grammar**. For the moment, this macro is not used.

BOOST_MSM_EUML_DECLARE_INTERNAL_TRANSITION_TABLE(iexpression,table-instance-name)

Like BOOST_MSM_EUML_TRANSITION_TABLE but does not provide an instance, simply a type declaration. This is currently the only way to declare an internal transition table with eUML. For example:

```
BOOST_MSM_EUML_DECLARE_STATE((Open_Entry,Open_Exit),Open_def)
struct Open_impl : public Open_def
```

```
{
    BOOST_MSM_EUML_DECLARE_INTERNAL_TRANSITION_TABLE((
          open_close [internal_guard1] / internal_action1 ,
          open_close [internal_guard2] / internal_action2
    ))
};
```