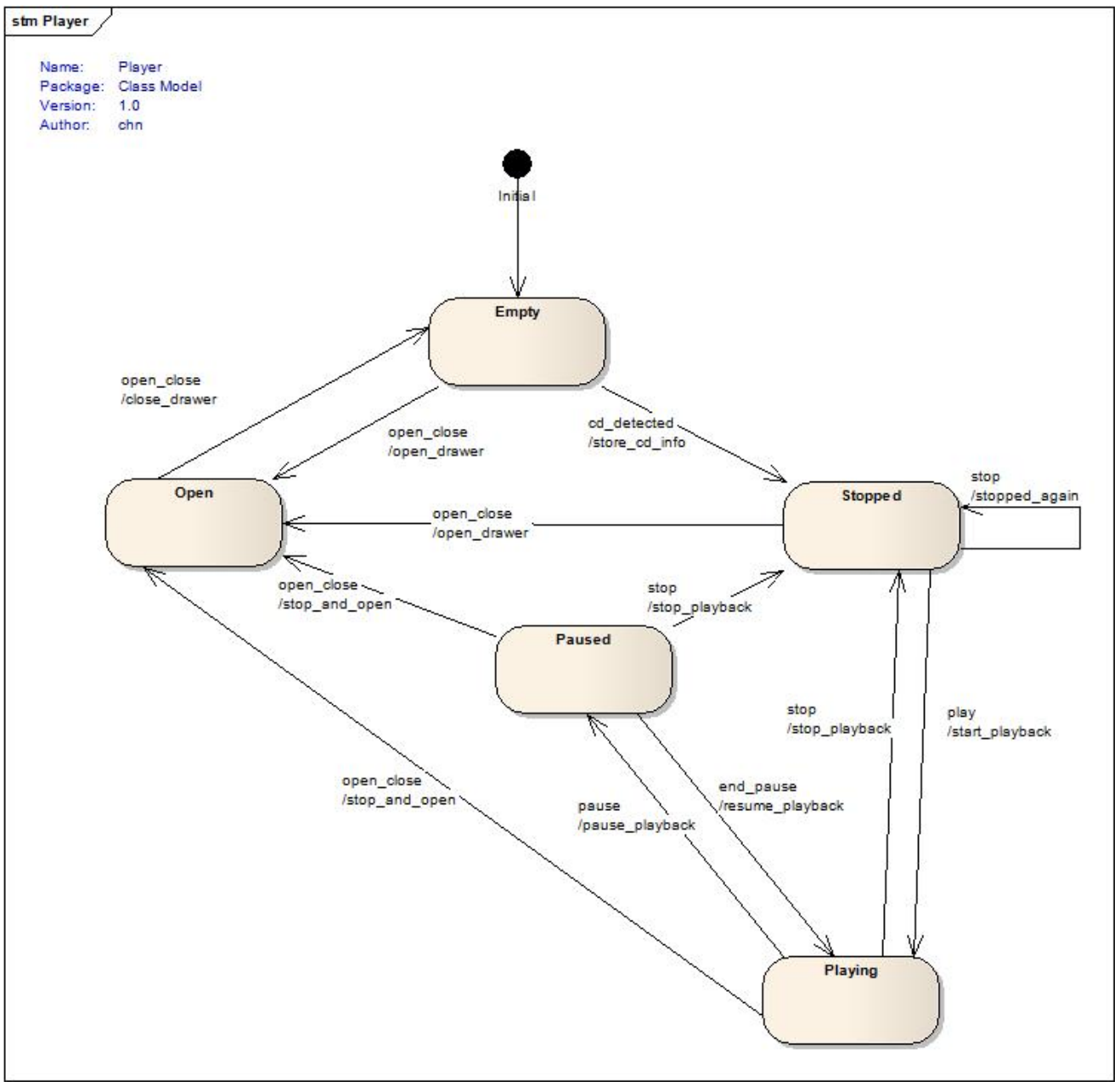
Basic front-end

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 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_::good_disk_format >,
  row< Empty   , cd_detected,  Playing  , &player_::store_cd_info   , &player_::auto_start       >,
//   +---------+-------------+-----------+---------------------------+----------------------------+
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_machine<launcher_> &>(*this);
  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
```

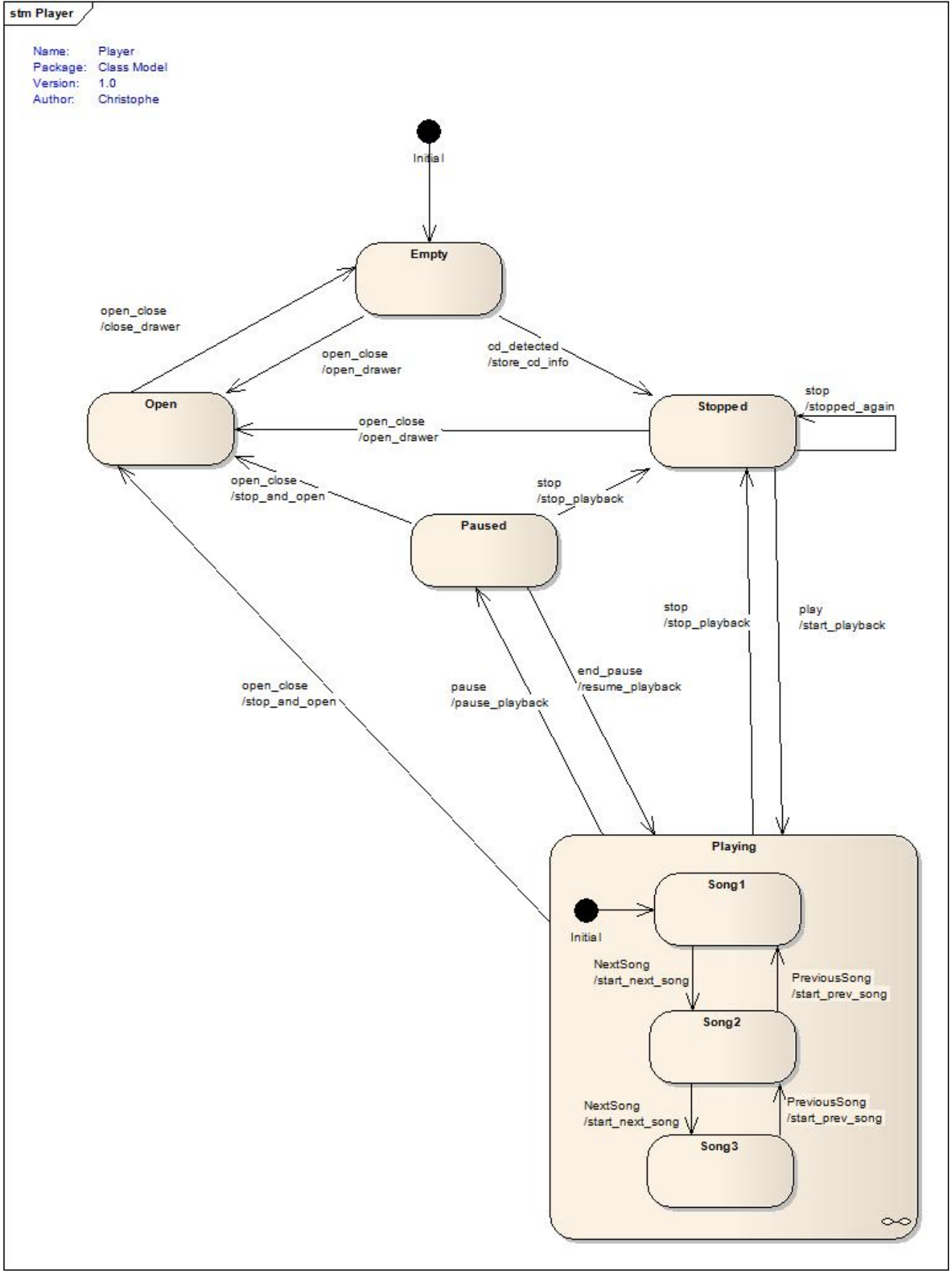
When an unexpected event is fired, the `no_transition(event, state machine, state id)` method of the state machine is called . By default, this method simply asserts when called. It is possible to overwrite the `no_transition` method to define a different handling:

```
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 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:

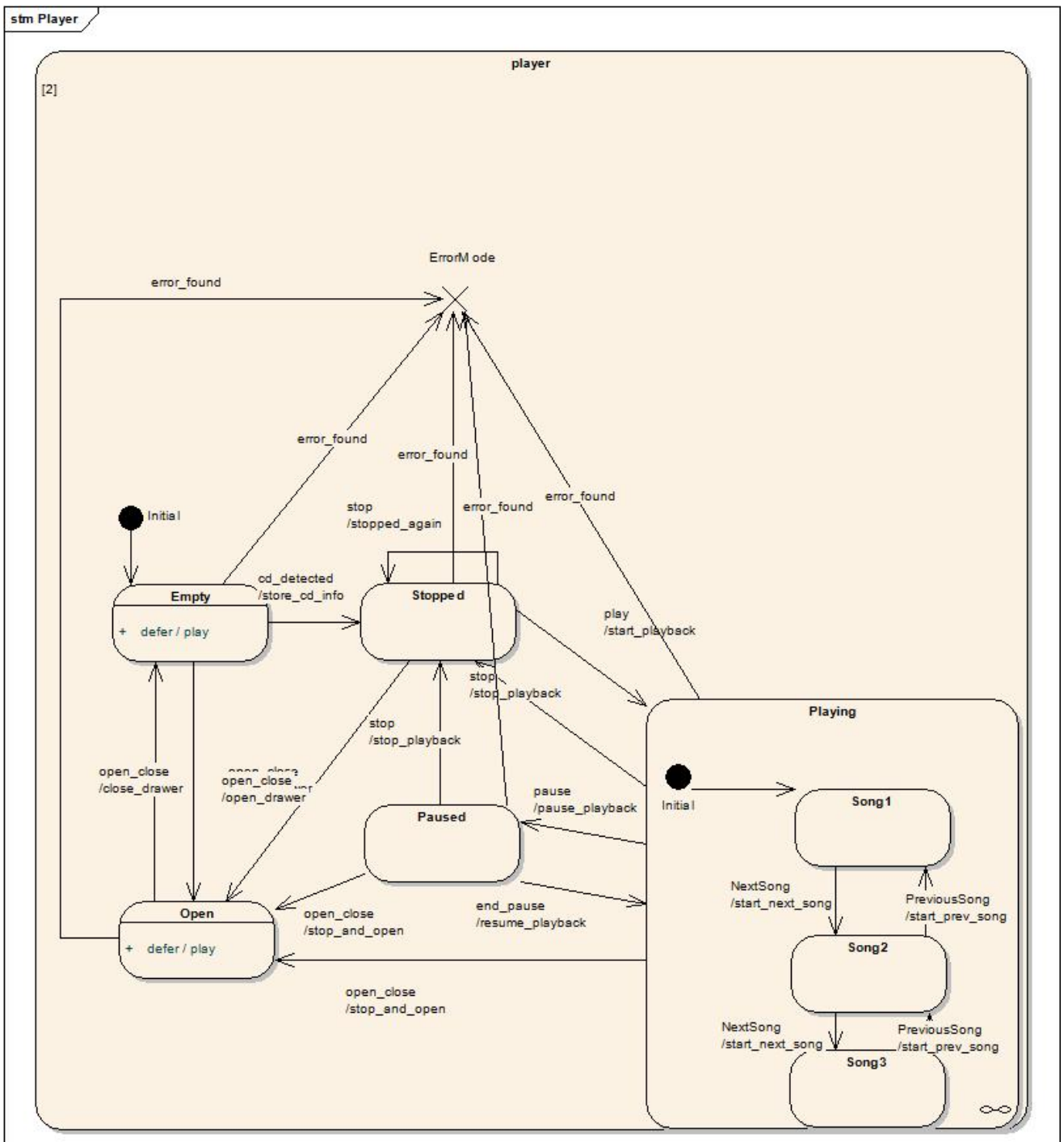
```
// define a property for use with enable_if
BOOST_MPL_HAS_XXX_TRAIT_DEF(some_event_property)

// 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>::type
   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,void>::type
   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:

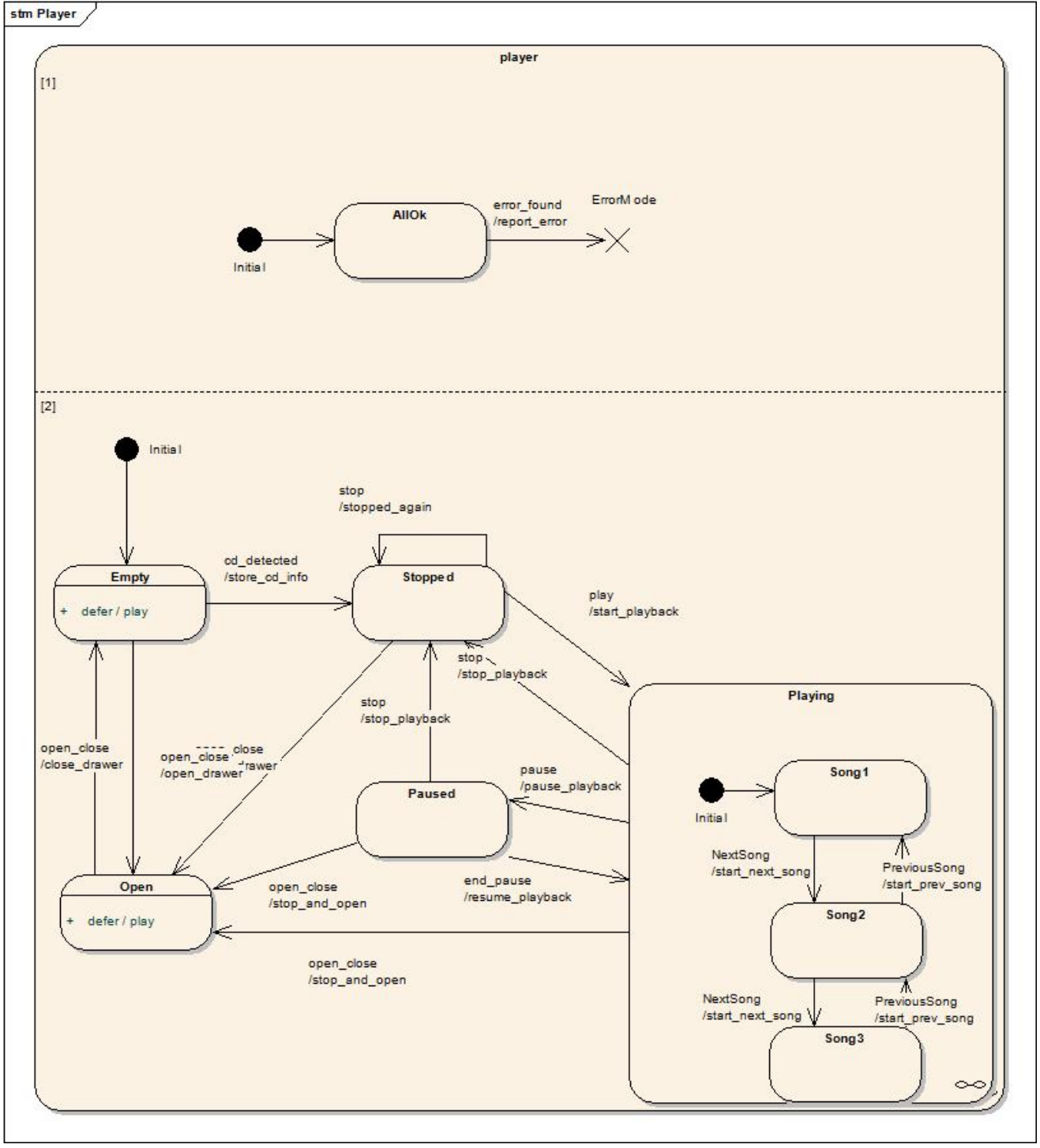


This is neither very readable nor beautiful. And we do not even have any action on the transitions yet to make it even less readable.

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.

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 >
```

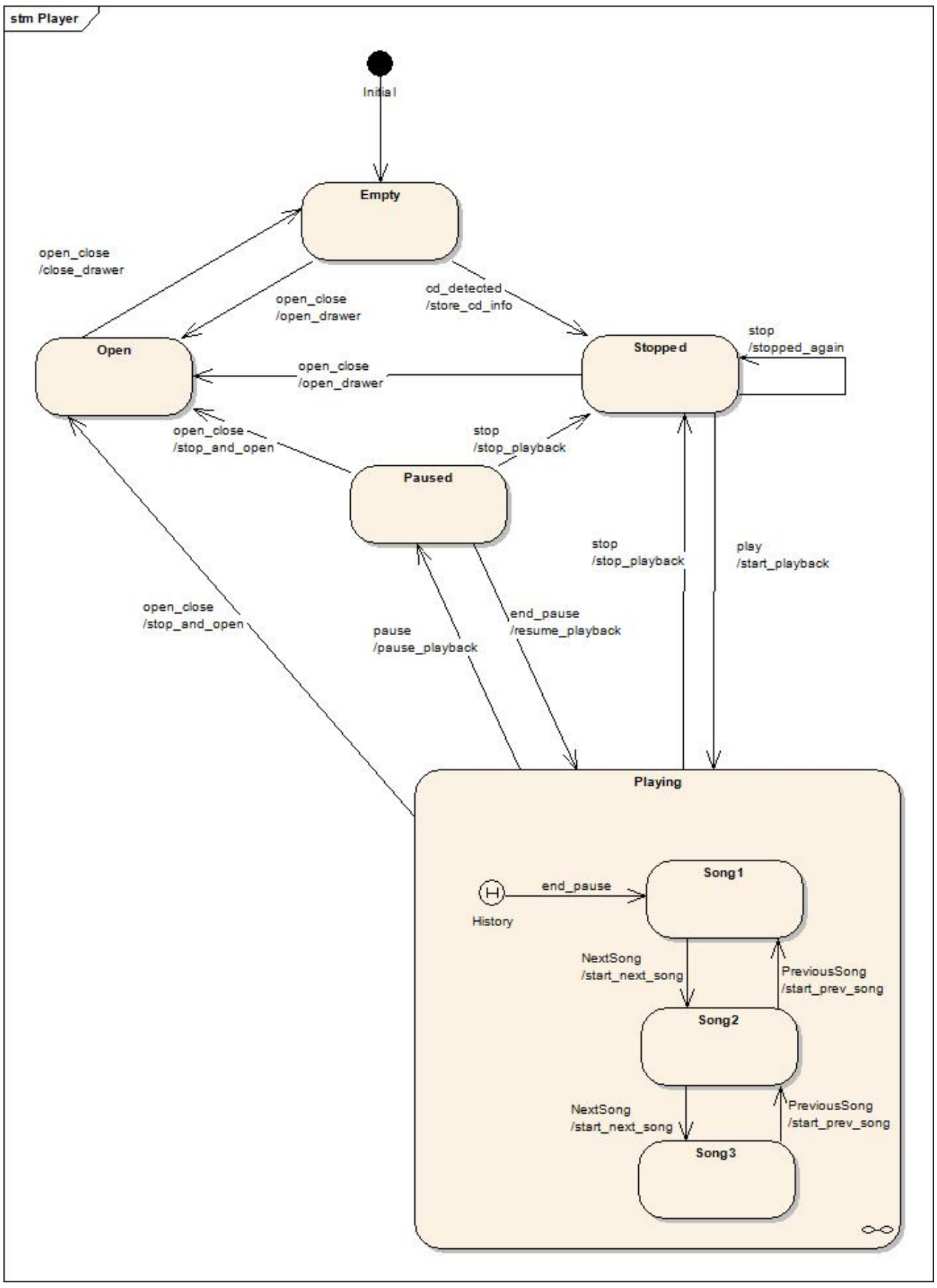
Please have a look at this possible implementation.

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_>
```

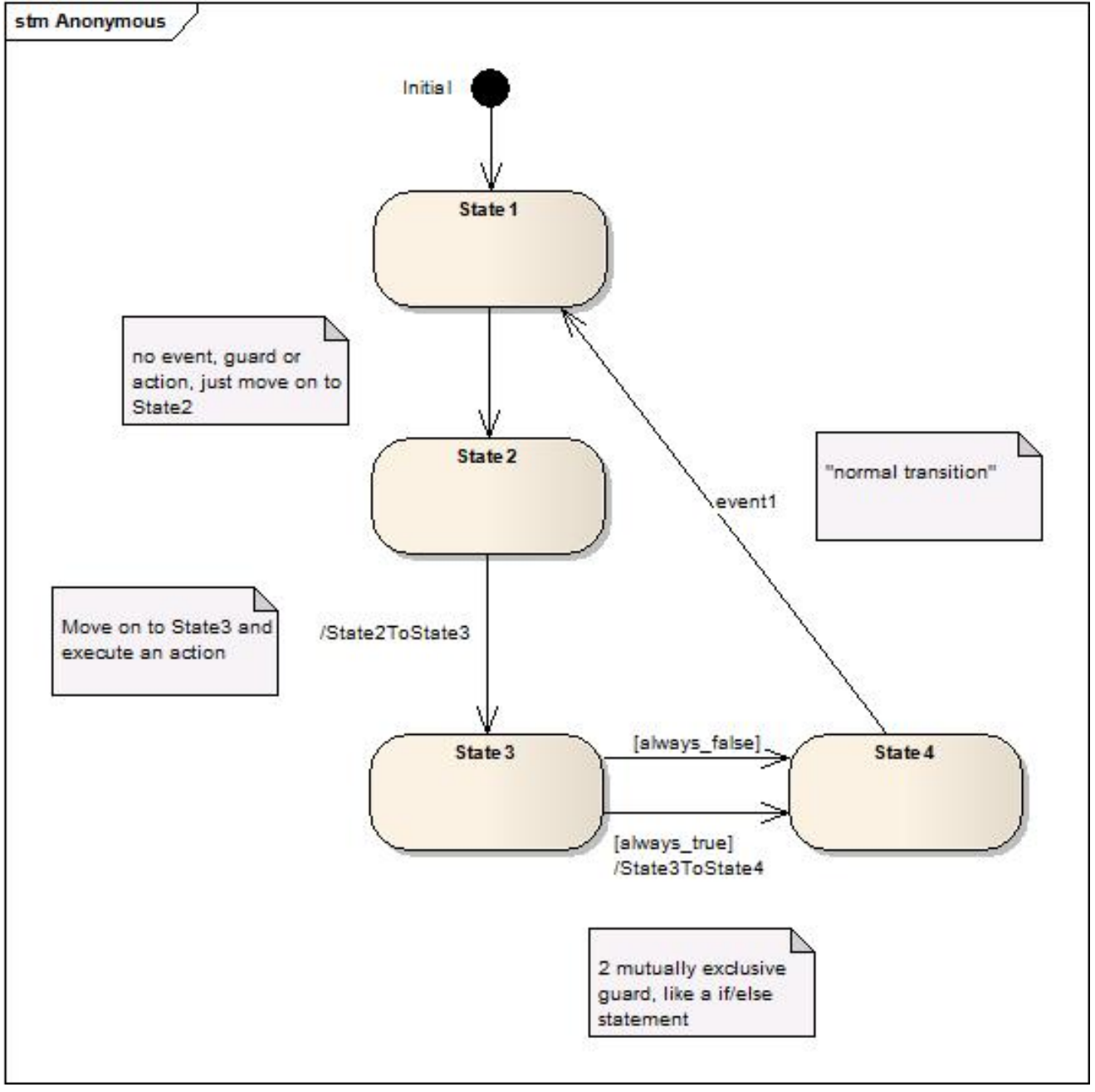
You then add the policy to the backend as second parameter:

```
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 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 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.

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 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
- Empty header and cpp
- Open header and cpp
- events definition

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.

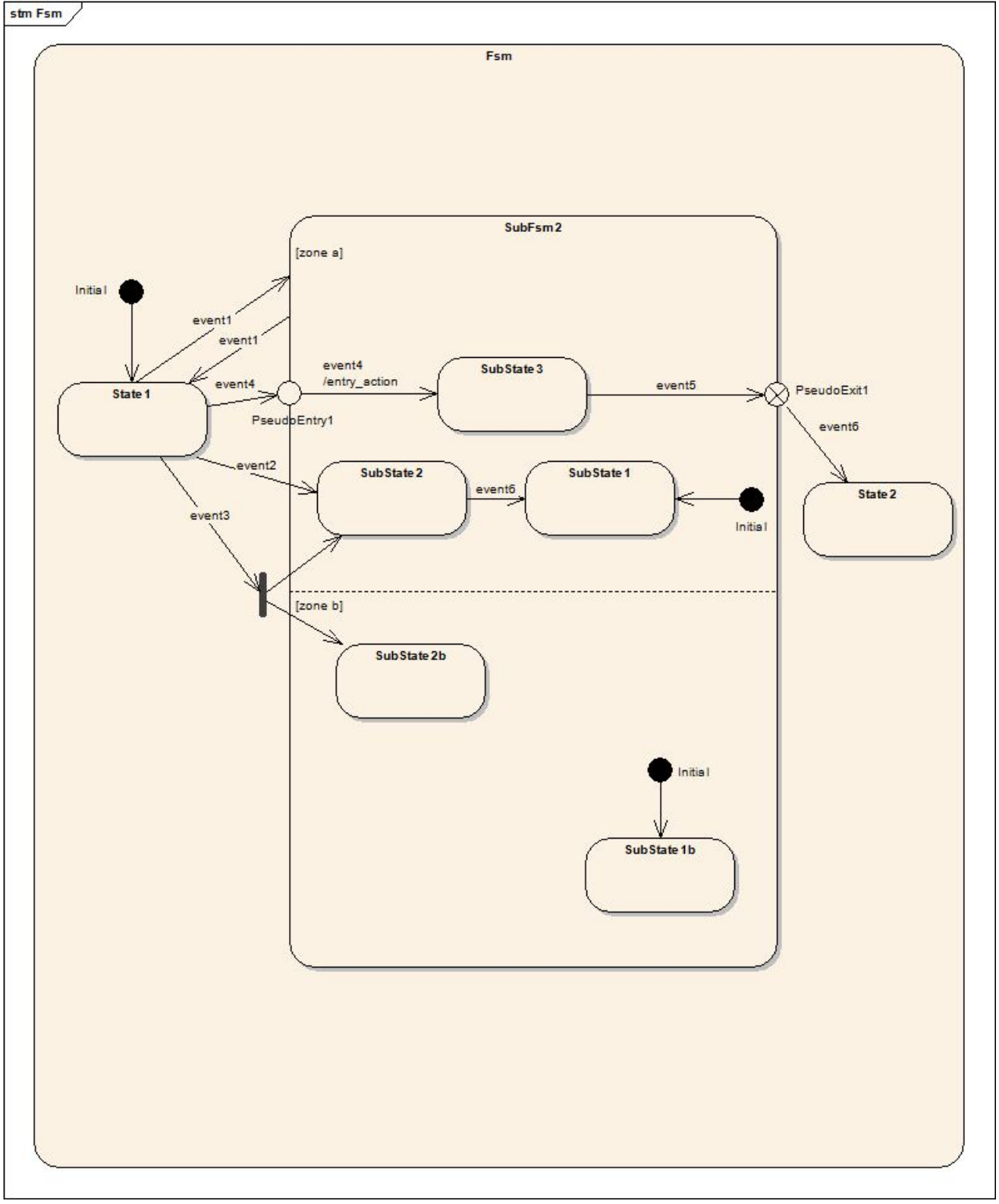
These row types allow us to distribute the state machine code among states, making them reusable and more useful. Using transition tables inside states also contributes to this possibility. An example of these new rows is also provided.

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.

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_::SubState2: complete name of SubState2 (defined within SubFsm2_)
```

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 event not having any val
    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 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>()) ...
```

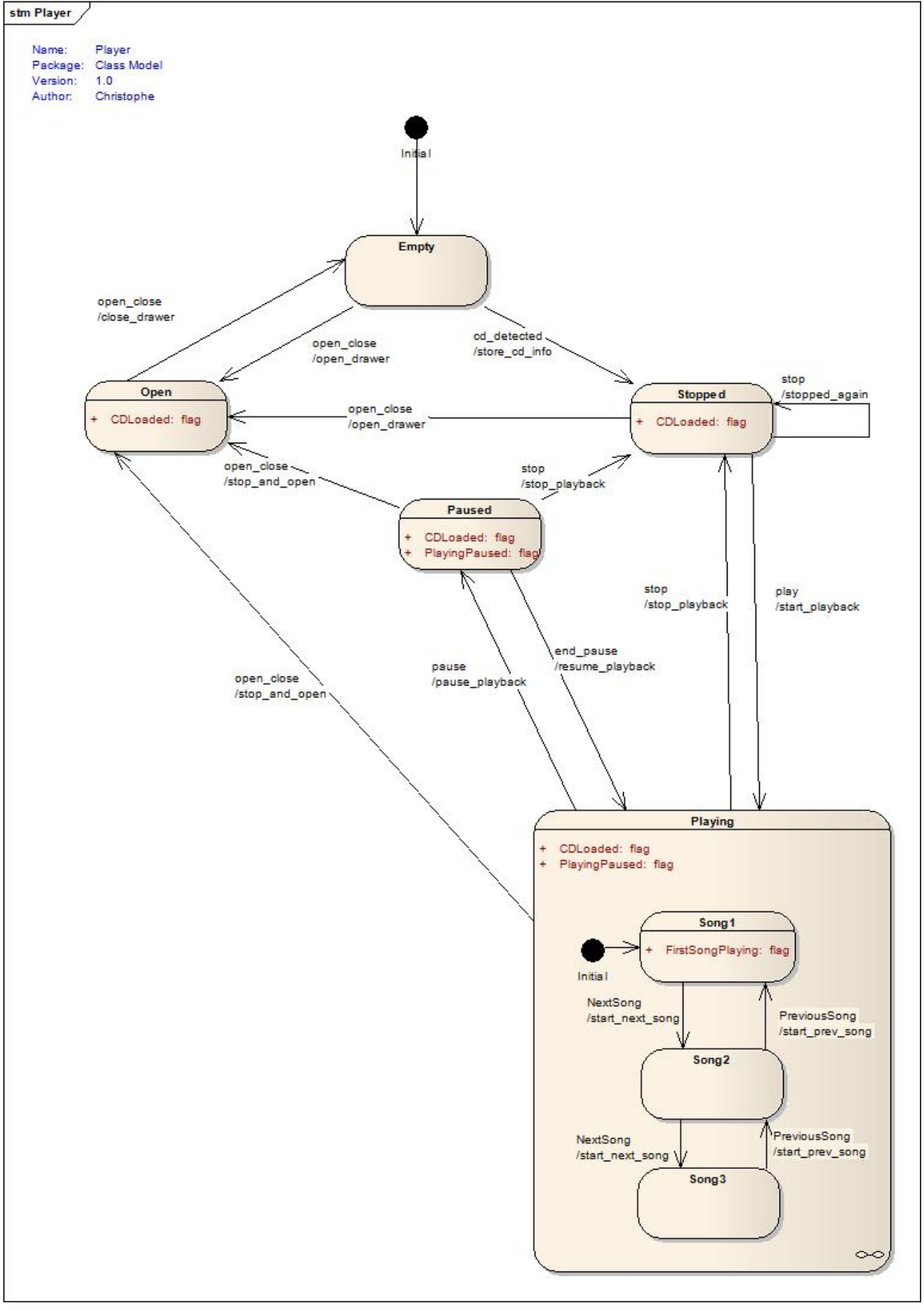
And what if you have orthogonal regions? How to decide if a state machine is in a flagged state? By default, you keep the same code and the current states will be OR'ed, meaning if one of the active states has the flag, then is_flag_active returns true. Of course, in some cases, you might want that all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

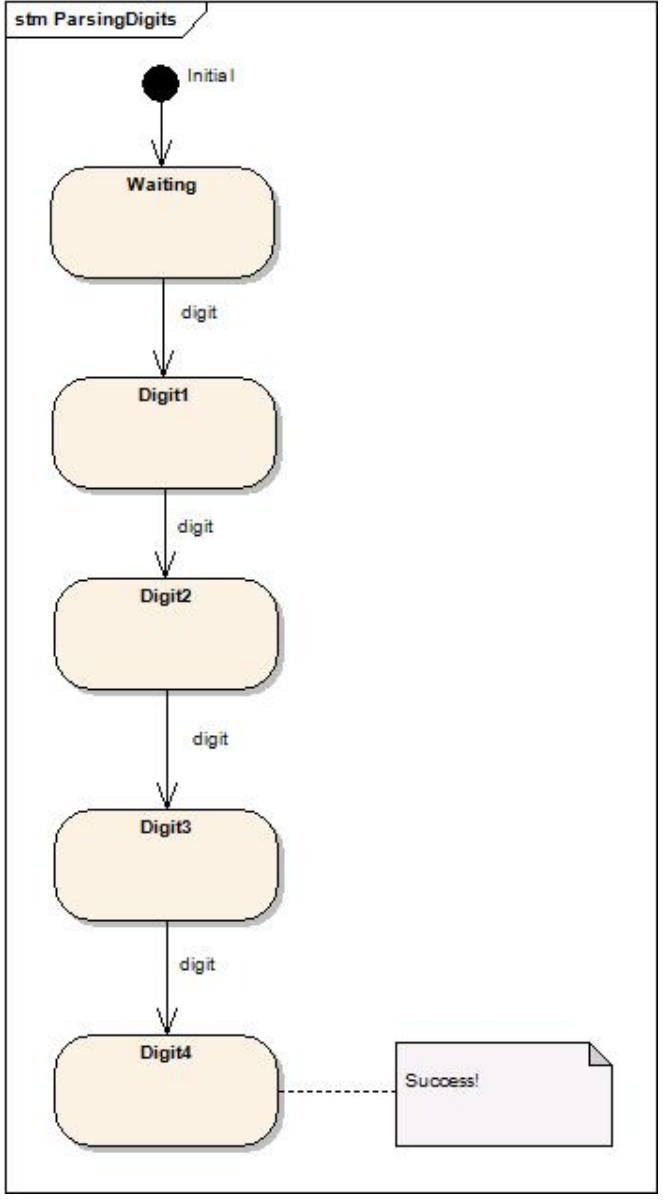
```
if (p.template is_flag_active<CDLoaded>()) ...
```

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 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.
