Tutorial: The Meta State Machine (Msm) library

Christophe Henry

christophe.j.henry@googlemail.com

ABSTRACT

The first part of this session is a tutorial aiming to teach attendees how to use the Msm (Meta State Machine) library, a framework allowing developers to easily define high-performance UML 2.0 state machines in C++ with the stated goal of facilitating a MDA (Model-Driven-Architecture) process.

The tutorial first briefly explains the objectives and principles of the library. It then proceeds to describe the concepts defined by the UML 2.0 standard for state machine diagrams, their use in the light of a Model-Driven Development process, and their implementation using Msm.

The second part of the session is a hands-on course in which we will try to implement a mock-up of an iPod nano's user interface. While this sounds like a lot of work, we will see how easy it becomes using Msm.

Keywords

UML 2.0, State machine diagram, C++ template metaprogramming, MDA (Model-Driven-Architecture), MDD (Model-Driven-Development)

1. INTRODUCTION

1.1 A MDD Process

In a conventional software development process, one (hopefully) engineers requirements, elaborates a design, then writes code. Documents other than code serve documentation and communication purposes but have otherwise no other value. The process is mostly one-way, from design to code, inconsistencies between both are unfortunately common and tend to increase with time as deadline pressure lets the documentation work slip behind the more important code delivery. This happens even in iterative processes like the Unified Process.

In a Model-Driven-Development (or Architecture, MDA) process, requirements, design and code are all models. Mapping functions transform one model into another, which implies that they are a prerequisite for the whole process to work. These mapping functions are said to represent *repeatable design decisions* [5].

Sometimes, one also requires these functions to be *reversible*, meaning that they can for example deliver a design model based

on code. This property is of course the most difficult to achieve as it implies reverse-engineering capabilities. It is however most useful, enabling synchronization between design model and code and seamlessly incorporating legacy code.

Msm aims to help in one -tiny- part of such process by making it easier to build a tool (which could also be a "human" tool) serving as transformation function between an UML State Machine Diagram and code, all while keeping the generated code readable and easy to understand and thus, allowing reverse engineering and reversible mapping functions.

Of course, writing such tools to generate code from a State Machine Diagram is nothing new. Free and commercial software, for example in the automotive industry already achieve this.

Unfortunately, these tools are often failing at providing the *repeatable* and *reversible* properties, let alone generate readable code.

They might generate full code from a model and overwrite whatever was in the source code file. This means that workarounds, corrections and additions all have to be merged by hand after automatic code generation. Clearly, this is not a *repeatable* process.

The main reason for these shortcomings is the non-descriptive nature of the underlying state machine implementation.

Often, the state machine framework mixes structure with code. For example this code from the Boost.Statechart documentation [5] clearly makes it hard to build a *repeatable* and *reversible* mapping function as a mapping function would have to sort out structure information from the rest of the code:

```
if ( context< Camera >().IsMemoryAvailable() )
{
  return transit< Storing >();
}
else
{
  // The following is actually a mixture between an in-state
  // reaction and a transition. See later on how to implement
  // proper transition actions.
  std::cout << "Cache memory full. Please wait...\n";
  return transit< Focused >();
}
```

1.2 The principles

To better explain what the different features bring, the tutorial is planned as a journey from the Player state machine example as found in the Boost.MPL documentation to the full-blown UML state machine library that Msm offers.

Msm aims to respect as many as possible the main design goals as defined by David Abrahams and Aleksey Gurtovoy [1] and presented in the order of importance as seen by the author of this article:

1. Declarativeness: The focus of a user should be on the structure of the state machine rather than the implementation. This greatly helps with the desired *repeatable* and *reversible* property of a MDD process.
2. Expressiveness: The framework must offer a syntax which is easy to understand by an UML designer.
3. Maintainability: Simple changes to the state machine design should result in simple changes to its implementation. Better, a mapping function should be able to handle this automatically.
4. Scalability: the framework should allow an easy extension. Msm constitutes the proof that this goal was achievable as it was written by someone not connected to the authors of [1].
5. Efficiency: the speed of the framework should never be able to be used as an excuse to use ad-hoc logic instead of the framework. The documentation of Boost.Msm [4] provides some performance measurements to address this usual critic.
6. Static Type safety: Most problems must be caught at compile-time.

The principles stated in this book constitute a perfect base for the construction of a framework adding the features that UML designers expect while following the stated goal of Model-Driven capabilities. In particular, designers expect support for:

1. Entry / exit actions in each state.
2. Composite states: to allow functional decomposition of a problem.
3. Reusable state machines: in order to build parts reusable at will and at different level of abstraction, each based on lower abstraction levels.
4. History: for increased flexibility in the handling of events.
5. Deferred events: to simplify concrete state machines by reducing the number of states / transitions the only purpose of which would be to save event information.
6. Orthogonal zones in composites: to group state machine parts which conceptually belong together.
7. Terminate (and Final) pseudo-states: to stop event handling.
8. The different kinds of entries / exit into / from a composite state (explicit, fork, entry/exit pseudo states).
9. Proper handling of transition conflicts (different possible transitions for a given incoming event).

2. TUTORIAL

2.1 Founding example

Let us have a look at the definition of a concrete state machine, as presented in [1] §11.4.2 :

```
class player : public state_machine<player>
{
private:
   // the list of states
   enum states {Empty, Stopped..., initial_state = Empty};
   // transition actions
   void start_playback(play const&);
   void open_drawer(open_close const&);
   …

   friend class state_machine<player>;
```

```
    typedef player p; // makes transition table cleaner

    // transition table
    struct transition_table : mpl::vector11<
   row <Stopped , play           , Playing , &p::start_playback  >,
   row <Stopped , open_close , Open    , &p::open_drawer    >,
    …
    > {};
};
```

What is truly impressive in this example is the very high signal / noise ratio. Almost all of the necessary information, the state machine structure, is contained and easy to read, in the transition table. One can reconstruct the structure in a few minutes. It also does not take longer to write the necessary code. The stated goals of declarativeness, expressiveness and maintainability are clearly met.

Furthermore, this can also be used in a context of a MDD process. The structure being separated from the code (and thus additions/modifications hand-made in the code easy to recognize or even better to ignore), the process is clearly *repeatable*. It is equally clear that it would be quite easy to generate an UML model from this code, thus allowing a *reversible* mapping function. Concretely, to generate a model from this code, one needs only parse the transition table. The table is simply a sequence of rows made of a limited number of parameters, separated by a comma. This is clearly an easy process.

This example shows that this technique constitutes a good base for the construction of a full-blown UML 2.0 state machine library facilitating a MDD process. The following parts of this paper will present important UML 2.0 concepts which were unsupported in this example and their implementation with Msm, all while keeping in mind the previously described important MDD-enabling properties.

2.2 A simple state machine with Msm

This example will introduce the following UML concepts [6]:

1. State machine: a concrete, UML-conform concrete model describing the behavior of a system. A state machine is composed of a finite number of states.
2. Simple state: A state that does not have substates. A state can have data, entry and exit behaviors and deferrable triggers.
3. Entry: an *optional* behavior that is executed whenever a state is entered, regardless of the transition taken to reach the state.
4. Exit: an *optional* behavior that is executed whenever a state is exited, regardless of the transition taken out of the state.
5. Deferrable Triggers: A list of triggers (also called events) that are candidates to be retained by the state machine if they trigger no transition out of the state. A deferred trigger is retained until the state machine reaches a configuration where it is no longer deferred.
6. A transition is the switching between active states, triggered by an event and is graphically represented by an arrow ending on a state boundary above which a description is added in the form: `event [guard] / action`
7. Guard: an *optional* constraint evaluated when an event occurrence is dispatched by the state machine. If the guard is true, the transition may be *enabled*, otherwise it is *disabled*.
8. Action: an *optional* behavior to be performed when the transition fires.
9. Initial state: the first active state of a state machine.

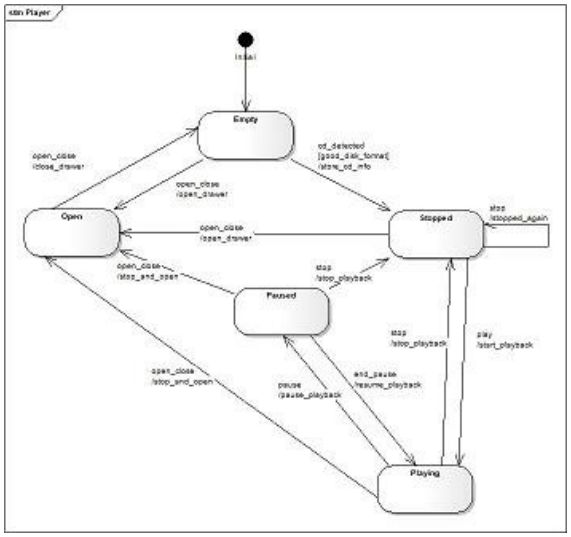


Figure 1: a simple state machine

A state machine is in UML a graphical model of a system. It is easy to understand, provided one knows the basic grammar defined by UML. A state machine is a collection of states in which the system can be. The system changes state when something (called an event) happens. It starts in a default configuration (an initial state) and when a relevant event occurs, it changes its state. By doing this, it can execute some code. This code can be divided in code always executed when a state is left (an exit action), when the transition is executed (a transition action) or when the new state becomes active (an entry action), in this order. A guard condition is used to block a transition (a row in a Msm transition table) and thus exit and entry actions, or to resolve a conflict between two possible transitions triggered by the same event with the same active state.
A deferred trigger is an event which is detected and relevant, which cannot trigger anything in the current state, but that the designer would want to keep "warm" for later use (in the first state in which the event is not marked as deferred). A common example is a user input which is not immediately usable (because the system is not ready yet) but will be as soon as the described system reaches a state where the event can be used.

The implementation of these concepts is simple and tries to stay as close as possible to the original design goals:

1. State machine: a struct/class deriving from `boost::msm::state_machine<Derived,HistoryPolicy,BaseState,CopyPolicy>` where `Derived` is the name of the concrete state machine (here Player) and `HistoryPolicy` (see later) and `CopyPolicy` policy-based tuning capabilities. `BaseState` allows customization of contained states. A Msm-described concrete state machine is made (at minimum) of a transition table, action and guard methods, and (an) initial state(s).
2. State: a struct/class deriving from `boost::msm:state< BaseState,Policies >` and optionally defining an entry and an exit method.

3. Entry: a (template) method of a state called `on_entry` and getting the event as argument:
```
template <class Event> void on_entry
(Event const&);
```
4. Exit: a (template) method of a state called on_exit and getting the event as argument:
```
template <class Event> void on_exit
(Event const&);
```
5. Deferrable Triggers: realized by a typedef defining a compile-time sequence of events to be deferred for a given state:
```
typedef mpl::vector<some_event>
deferred_events;
```
6. Transition: implemented as a row in the transition table:
```
a_row<Stopped, play, Playing,&p:start>
```
7. Guard: an optional method of the *state machine* returning a boolean to indicate if a transition is enabled / disabled:
```
bool good_disk_format
                  (cd_detected const&);
```
8. Action: an optional method of the *state machine* returning nothing and taking the event as argument:
```
void start_playback (play const &);
```
9. Initial state: Msm does not define a special type for this pseudo-state concept and requires instead a *mandatory* typedef inside the *state machine* definition:
```
typedef Empty initial_state;
```

Please note that states do not need to know in which state machine they are used, therefore increasing reuse. The Msm library automatically creates upon state machine initialization all the states it finds in either the transition table or the `initial_state` sequence. It also customizes itself to deactivate parts a concrete machine does not need.

A small change compared to the version presented in [1] induced by the addition of a guard condition in the rows is the definition of several types of rows:

- row takes 5 arguments: Start, Event, Target, action method, guard condition
- a_row ("a" for action): Start, Event, Target, action method
- g_row ("g" for guard): Start, Event, Target, guard condition
- _row: Start, Event, Target

How would one implement the same behavior with Boost.Statechart?
First of all, one would need to define a part of the transition table in every state, for example:
```
struct Empty: sc::simple_state<Empty,Player>
{
typedef mpl::list<
    sc::transition<open_close,Open>,
  sc::transition<cd_detected,Stopped,player,&
player::store_cd_info>
> reactions;
};
```
Notice how Empty must now know not only the state machine where it is used, but also the events triggering a transition from it, and all the target states. This makes for a fair amount of forward declaration noise. This also makes the states impossible to reuse elsewhere.

Second, notice that there is no possibility to define a guard condition. So, as we previously saw, if a transition needed a guard we would have to change it to:
```
sc::custom_reaction<open_close>
```
Then add a react function:
```
sc::result react (const open_close&);
```
And implement it:
```
sc::result Empty::react(const open_close&)
{
if (…)
   return transit<Open>();
return forward_event();
}
```

Now, the structure of the state machine is completely hidden inside the code. Not only is this considerably more code, it is also less readable and mixes code with structure, thus giving tools a harder time and reducing MDD capabilities.
As there is no centralized transition table, it also becomes harder to build some metaprogramming tool to analyze a state machine structure at compile-time.
Another problem is that it is error-prone as one does not necessary know that this works only when Empty has no entry or exit actions. It is therefore deceptively easy to build a non-UML compliant state machine. It is also unclear how to use entry, exit and guards at the same time.

2.3 Using composite states

The previously defined concepts are what most designers are actually using. And it is a sad fact as UML state machines are much richer. Let us discover more interesting concepts which make this model-driven approach more useful.

This example will introduce the following UML concepts as described by the standard (hold your breath while reading) [6]:

1. Composite State: a state containing a region or decomposed in two or more orthogonal regions. A composite state contains its own set of states and transitions.
2. Orthogonal regions: Parts of a composite state, each having its own set of mutually exclusive disjoint subvertices and transitions.
3. Submachine state: a state machine inserted as a state in another state machine. The same state machine can be inserted more than once in the context of a single containing state machine.
4. Terminate pseudostate: Entering this state implies that the execution of this state machine is terminated.
5. Interrupt state: An addition provided by Msm and missing from the UML specification: a temporary terminate state resumable by a given event.
6. Shallow / Deep history: shallow History represents the most recent active substate of its containing state (but *not* the substates of that substate) . A composite state can have at most one shallow history vertex. A transition coming into the shallow history vertex is equivalent to a transition coming into the most recent active substate of a state. At most one transition may originate from the history connector to the *default* shallow history state. This transition is taken in case the composite state had never been active before. Deep History is a shallow history with representation of the substates of the most recent active substate.
7. Exit point pseudostate: an exit point of a state machine or composite state. Entering an exit point within any region of

> the composite state or state machine referenced by a submachine state implies the exit of this composite state or submachine state and the triggering of the transition that has this exit point as source in the state machine enclosing the submachine or composite state .

This is standardese, a barely human-readable language, and deserves a bit of explanation. Composite states and submachine states are very similar and can safely be assumed to be the same concept (the difference being that the later can have instances where the former cannot. If you don't get it completely, you are not alone, UML tools sometimes get it wrong too). They are both state machines embedded as a state in another state machine. Why would one need this? First of all, to make diagrams simpler. A diagram with more than 10 states and 30 transitions can be unreadable. The second reason is factorization of behavior into a common, reusable unit. This is similar to writing sub functions. Orthogonal regions are lightweight submachines contained in a state machine/composite state. These submachines have no common state or transitions (meaning they are orthogonal) and run mostly independently. Mostly but not completely. They are all activated at the same time when their containing machine is activated, meaning we now have more active states at a time. There are also exited at the same time when the containing machine becomes inactive. And they also can have influence on each other. When in one region a terminate/interrupt state becomes active, all regions are terminated. When a region is exited through an exit point, all regions are exited. Orthogonal regions are very practical as they can easily send events to each other and share common data (of their containing state machine), a bit like threads do. Exit points are useful to define a way for a state machine to define itself when it wants to be exited.

Let us now come to the History concept. To be frank, History is not a completely satisfying concept. First of all, there can be just one history pseudostate and only one transition may originate from it. So they do not mix well with orthogonal zones. Just one zone gets history, others get default-initialized (the initial state). First fly in the ointment. The second one is even worse, deep history. It looks like a last-minute addition and does not fit in the plan. If history has to be activated by a transition and only one transition originates from it, how to model deep history? The small '*' does not tell us which inner state has to be selected (if in our example Song1 was itself a submachine), unless the implementation of Song1 has to be known to the outside world (Playing in the example) so that a direct transition can lead to it, which is not really a proof of encapsulation and forces external influence in its definition. Actually, Shallow History is also breaching encapsulation as it requires a transition leading to it from “outside”. 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, but the UML version is not quite making the cut. And therefore, Msm provides its own version of it, a policy template argument of `state_machine` which solves the previously described problems and allows user-defined histories. It would for example be possible to have an history remembering the states which have been active in a state machine for diagnostic functions.

Terminate states are supposed to terminate a state machine when becoming active. And here comes the third fly. If the state machine is terminated (or even destroyed), can we use history? Alas, the UML standard forgets to explain this point, and thus Msm also provides its own definition of terminate states.

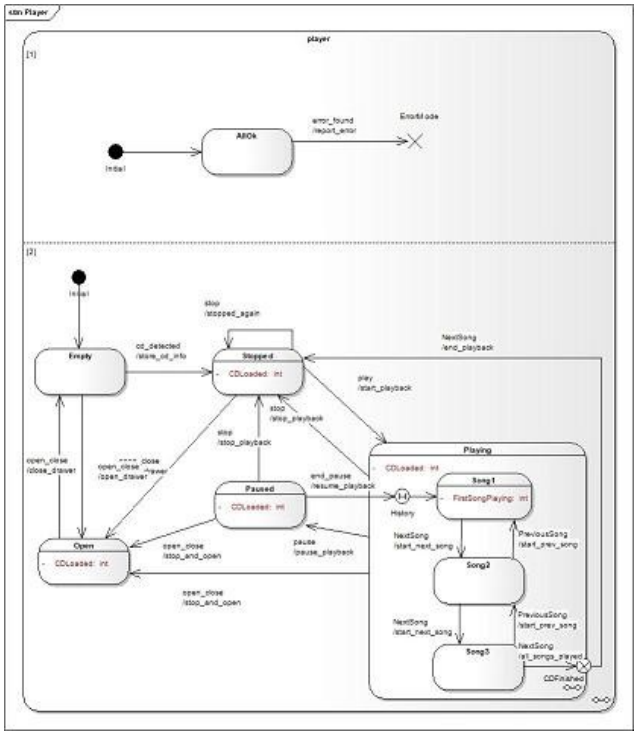


Figure 2: a state machine with two orthogonal zones and containing a composite state with history and exit pseudo state

Again, in the implementation of these concepts, Msm tries to stay close to the original concepts of [1] and is therefore based on compile-time parameters and on the transition table:

1. Composite states are simply state machines, meaning they simply need to derive from `msm::state_machine<>` instead of `msm::state<>`. This fusion of state machines and composite states allows not only functional decomposition, as it is possible to define a state machine as a state machine of substates and submachines, made themselves of substates and submachines, but also reuse as a state machine can be used as submachine or as a stand-alone state machine. Msm automatically detects if a state is itself a state machine and handles event forwarding, exit points, transition conflicts and other (later defined) UML features. It also detects whether a state machine is used as stand-alone.
2. Orthogonal regions are defined by adding more initial states to the `initial_state` typedef. So, for example, the player state machine's initial state typedef becomes:
```
typedef mpl::vector<Empty,AllOk>
                    initial_state;
```
Please note that regions are themselves not a type and that all the states of a region are instead directly attached to the same state machine. They are a conceptually no more than a list of active states. Thus they can easily share common data located in the common containing state machine.
3. Submachine states are implemented like composite states for Msm and thus defined by also deriving from `msm::state_machine<>` . It is also possible to define

instances. For example, if we needed another `Playing` submachine, we could derive `Playing1` and `Playing2` from `Playing`. These would be automatically constructed if appearing in the transition table. Msm is type-based and two state machines disguised as different types are indeed different instances of a state machine.

4. Terminate pseudostate: to make a state a terminate state, simply derive it from `msm::terminate_state<>` instead of `msm::state<>`. If this state becomes active, event handling in all regions is disabled but nothing is destroyed, thus still allowing History and reading from data in states or machines.
5. Interrupt state: to make a state an interrupt state, simply derive it from `msm::interrupt_state< EndInterruptEvent >` instead of `msm::state<>` where `EndInterruptEvent` is the only event which can be processed and will unlock the state machine. Of course, as the framework needs to exit the interrupt state, a row must be added in the transition table, with the interrupt state as source and a non-interrupt state as target, with `EndInterruptEvent` as the triggering event.
6. Shallow / Deep history: For simplicity (to avoid defining another state), to allow new types of histories and so that history can apply to all orthogonal zones, Msm defines history as a policy. There are currently three predefined policies: `NoHistory`, `AlwaysHistory`(where history is always activated, no matter through which event), `ShallowHistory` (almost the UML shallow history). So, to use a shallow history, Player needs only a small change in its definition, precisely to inherit from:
   `state_machine<Derived, ShallowHistory<mpl::vector<end_pause> > >`
   where `end_pause` is the incoming event triggering the history activation. The template argument is a compile-time sequence so that several events can activate the history. There is no `DeepHistory` policy to avoid conflicts with policies from submachines and in order to let these submachines define their history needs, so to support deep history, the submachines simply need to also use the `ShallowHistory` policy.
7. Exit point pseudostate: To make a state an exit point pseudostate, one derives it from `msm::exit_pseudo_state<ContainingSM, ForwardedEvent>`.
   The UML standard does not define what happens if no outgoing transition is defined from the exit point pseudostate. In an effort to bring a solution, Msm marks the exit point pseudostate as a terminate state and will call `no_transition` while forwarding the event. This way, at least the submachine will behave a coherent way and will not further process events. UML also wants that the transitions leading to the pseudostate and the one going from it to be triggered by the same event, which is a little irritating given that the goal of the exit point pseudostate is to allow encapsulation. To improve this, Msm weakens this rule and accepts that the event triggering the incoming transition be only convertible to the one of the outgoing transition (which must be the `ForwardedEvent` event).

Boost.Statechart has no composite states the way Msm sees them. Instead, the structure is again spread among states, which makes it harder to see at first glance. For example, the Song1 state would be defined as:

```
struct Song1: sc::simple_state
<Song1,Playing>
```

The rest of the definition being like previously described. This means that not only needs Song1 know about all possible target states (in this case `Song2`) and events (`NextSong`) but also its containing state machine (`Playing`). This means increased coupling, reduced readability and a more complicated MDD process as tools (or developers) would have a harder time analyzing the structure.

Boost.Statechart has no terminate / interrupt pseudostates. The closest to it is a special reaction (terminate<>) and a special reaction function which then probably destructs the state machine and contained data.

Boost.Statechart implements history the UML-conform way with a transition leading to the history state, with all the previously described issues.

Boost.Statechart supports orthogonal zones quite similarly to Msm, with the difference that each state must indicate in which zone it is placed, which further reduces reuse.

Boost.Statechart has no direct support for exit point pseudostates and requires the user to define its own template state with the destination as argument. This probably means the user also has to write his own reaction.

2.4 Using flags and visitors

Until now, we saw how Msm helps building concrete UML state machines. But there is more and now that we have the main concepts, we will make a short digression, but an important one to further improve our MDD process. After generating the state machine structure, one will start implementing the action methods and then use his state machine in a given context. A first very common task is to check if the state machine is in one or more given states. For example, to know if in our example a CD is loaded, we could write, as done with Boost.Statechart:

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

This clearly does not look like it will help building a MDD process as it is an awful lot of not self-describing code. Furthermore, it hides what the user really wants to do, in this case to find out if the state machine currently is in a configuration where it fulfills a certain property (if a CD is loaded for example). Msm helps by providing a descriptive way to define state property, compile-time-defined markers it names flags. For example, to describe this “CD loaded” property, add in the four states named above the following typedef:

```
typedef   mpl::vector1<CDLoaded>   flag_list;
// CDLoaded is an empty struct
```

Now, not only is this less code and compile-time solved (meaning faster), it also more clearly shows the developer intent, making the code more maintainable. We then can replace the Boost.Statechart code above by (p being a player):

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

What happens if `player` has several orthogonal zones? Will the flag be present if one of the currently active flags has it, or if all of them do? By default, if just one, but you can provide a second template argument, `FLAG_AND` to give you the other behavior.

In the previous picture, `CDLoaded` has been defined for these four states, and a `FirstSongPlaying` flag into `Song1` to show that Msm can also work its way recursively inside a submachine.

Another very common task is to do something with the active states. While this is not directly linked to MDD, it is still useful and we will spend a few lines on it. To do something with the active state(s) is not strictly a visitor as described by the GoF [2] but still sort of visiting them and Msm names this pattern visitor. You might have noticed that msm::state_machine and msm::state have an extra not-yet-described parameter called `BaseState`. This parameter allows customizing of states, like making them polymorphic (as it is not by default) and also allows defining a visitor. To do this, you need to define a new base state with an accept function and a signature, for example:

```
struct my_visitable_state{
typedef    args<void,    int,const    char*>
accept_sig;
// default implementation for states not
doing anything useful while visited
// can be const or not const
void accept(int, const char*){} const
};
```

You now need to make your states inherit from my_visitable_state:

```
struct some_state: public
msm::state<my_visitable_state>
```

and inform your state machine:

```
struct player: public
msm:state_machine<player,NoHistory,my_visita
ble_state>
```

You can now simply call `visit_current_states(int, const char*)` on your concrete state machine.

To achieve the same result with Boost.Statechart you would need to:

1. define an interface with a virtual function
2. make all the states inherit from it
3. define the same function in your state machine and implement it using
   `state_cast<MyInterface&>().some_fct()`

This approach has the drawbacks of being much code, requiring a virtual call (and forcing a vtable onto all your states) and a RTTI-based `state_cast`. Again, Msm tries to be as descriptive as possible and does NOT require anything virtual. Furthermore, how Boost.Statechart handles the case of several active states (orthogonal zones) is unclear to the author of this paper.

2.5 Using different entries

There are in UML more ways to enter a composite state other that a transition ending on its boundaries and resulting in the initial states of every orthogonal zones to be activated. Let us count the ways as described by the standard[6]:

1. Default entry: An incoming transition terminates on the outer edge of a composite state. In every region, the initial state becomes active. This is the standard case we used until now.
2. Explicit entry: If the transition ends on a substate of the composite state, then that substate becomes active and its entry code is executed after the execution of the entry code of the composite state. This rule applies recursively if the transition terminates on a transitively nested substate. All of the other orthogonal zones (if any) become active by default (the initial state becomes active).
3. Fork: An explicit entry into one or more regions. Again, other regions become active by default (the initial state will become active for these regions).
4. Shallow / Deep History: see §2.3.
5. Entry point entry: If a transition enters a composite state through an entry point pseudostate, then the entry behavior is executed before the action associated with the internal transition emanating from the entry point.

As this is quite abstract, and to illustrate the different entries, we will implement a state machine described by Harel [3] with slight modifications.

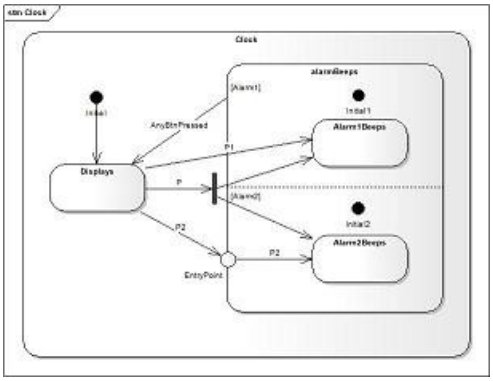


Figure 3: a clock implemented using explicit entry, fork and entry point.

P1 is triggering a direct entry and makes the first alarm active, P2 does the same using an entry point and P activates both using a fork.

As they are in the UML standard, these features are supported by Msm with some limitations (explicit entries cannot end deeper than the substate of a state and exiting a composite state that way is not possible). Does one really need these features? In most cases no, one could simply have the `alarmBeeps` submachine forward the event in its entry condition and would preserve the "secret" of the inner implementation. The author almost never uses them and warns that these features should be handled with care. For example, the fork could be implemented as:

```
struct alarmBeeps:...
{
   template <class Event>
   on_entry(Event const& evt)
   {
```

```
        process_event (evt);
    }
…
};
```

This would preserve encapsulation while giving `alarmBeeps` the chance to change its implementation at will or add special event handling.

How are these UML features implemented with Msm? Like always, a descriptive way:

1. Explicit entry: A state must communicate that it wants to be used as direct entry (and into which zero-based zone) and must therefore derive not only from `msm::state<>` but also from `msm::explicit_entry<Composite,int zone>`. For example,
   ```
   struct Alarm1Beeps: public state<>,
   public explicit_entry<alarmBeeps,0>
   ```
   It can then directly appear in a row of the "main" state machine, for example in Clock's transition table, using P1:
   ```
   _row< Displays, P1 , alarmBeeps::
                        Alarm1Beeps    >
   ```
2. Fork: Like above, a state must be marked for explicit entries. The only change is in the row definition, for example, again in Clock's transition table, using P:
   ```
   _row< Displays, P , mpl::vector<
        alarmBeeps::Alarm1Beeps,
        alarmBeeps::Alarm2Beeps> >
   ```
3. Entry point entry: An entry point is defined as a connection between two transitions, meaning we not only have to define two transitions, but also can use it to enter only one zone, which we must name. So, for example the definition of EntryPoint is:
   ```
   struct EntryPoint : public
   entry_pseudo_state <alarmBeeps,1>
   ```
   We then need one transition inside alarmBeeps' table:
   ```
   _row< EntryPoint, P2 , Alarm2Beeps>
   ```
   And one in Clock's table:
   ```
   _row< Displays, P2 , alarmBeeps::
                        EntryPoint >
   ```

It has not been possible to find out much information on this subject in the Boost.Statechart documentation but explicit entry should be possible due to the distributed nature of the state machine implementation. Entry points are said to be simulated with a typedef, and it is unclear to the author of this paper how to make this UML-compliant (connection between two transitions). Forks are not supported.

2.6 Conflicting transitions

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

1. 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.
2. The source state is a submachine or composite and the conflict is between a transition with this submachine as origin and a transition triggered by the same event and having a substate of this submachine as origin.

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.

As to the second one, there is more than meets the eye: this is not just the case of any transition defined inside the submachine transition table but also the case when using an exit point pseudostate. The composite is active and can either take the inner transition or the one leaving this composite. In this case, UML defines that the most inner transition gets selected first, which makes sense, otherwise no exit point pseudostate would be possible.

Msm handles all these cases itself, so the designer needs only concentrate on its state machine and the UML subtleties, not on implementing this behavior himself.

As Boost.Statechart does not automatically handle guards, it is up to the user to implement a UML-conform conflict handling. However, how to handle a conflict between a transition inside a submachine and one outside is unclear.

3. AND NOW?

3.1 State machines vs ad-hoc logic

We know know what features UML provides in a state machine diagram and we also know how to implement them with Msm. So far so good. But how does this help a programmer in his everyday job? Most people think that they do not need a state machine too often, that it is nice but not made for practical work. So, when can you use a state machine? It turns out, more often as one may think, sometimes is a state machine hidden behind what looks like implementation details.

For example, who never wrote a boolean flag to indicate if some condition has been met? Or if the end user pressed a button but did not complete some other action yet? Or if a thread has completed his work? Come on, do not be shy, raise your hand. Of course we all used these kinds of flags. We do not particularly like it but simply turn a blind eye and try to convince ourselves that well, this is code and it has to be, and we still have such a wonderful design, the rest just being "implementation details" because it does not sound so bad. And then we start adding another flag. And some weeks later another one. And we start having conditions on several flags. And sometimes we forget some flags in the condition. Not immediately of course, only some months later because we forgot. Or the culprit is our colleague who had to fix something in our code and he did it wrong even though the flag was so clearly named. But isn't the problem that we mixed the task to be done with the implementation detail instead of using a tool with more expressive semantics? If that starts sounding to you like the same argument about why to use an

algorithm from the STL instead of a hand-written-loop, congratulations!

Even worse, a developer usually has a client or a boss. And they often want explanations about how the code is working, but all we have to show is code, sometimes a (fun) 10-line-lambda expression. Every client will manage after a few minutes to understand a state machine model, but for code, it could take a little longer.

We usually have the same arguments to use flags or hard-coded logic instead of a state machine:

- We have no time to write a state machine and then implement it. So we have to make a choice, and surprisingly always decide to keep the coding part.
- We have such a simple case, writing a state machine will take way too long. It's only a flag. Good, let's say two flags. And for each value I only have one action. For the moment.

Thanks to Msm, the first argument is much weaker. After we decide to think in terms of a model and define the state machine structure, the code is just a few steps away. And some tool can even eliminate this too.

The second one is also looking bad. Define a struct, copy from your other project a transition table, with only two flags it will be a short one anyway and in a second you're done. If you later realize you actually need a few more states, then the extra work is quickly amortized. And now we have a better documentation and can talk with the client in a language he can understand. And most of call, the code is traceable and always in sync with the model. And this is exactly what MDD is all about.

Now, why Msm and not another framework? It should be clear by now that a very important goal of Msm is to separate the structure from code. A developer using Msm does not have to think about details like when to call his action method or the guard condition. He simply defines the structure and the body of the actions / guards, the rest is automatically done and can then concentrate on the job to be done and what he is getting paid for.

All the grunt work is done by the framework, and mostly at compile-time. The framework calls at the correct time the entry, exit and actions, automatically forwards to a composite state the events it defines in its transition table, handles differently the same state machine if it is stand-alone or embedded as a composite, detects and handles conflicts, generates the necessary code for state flags, prevents accidental copies, detects errors and so on.

And all this while offering readable and fast code, code which even the marketing department could understand.

3.2 Further on the Model-Driven path

We saw in the previous sections that state machines go a long way in helping realize a Model-Driven process. On the other hand, they usually have drawbacks which reduce their use:

- Once the code is generated, it will need to be edited but most frameworks are not *reversible*. This means that at some point, model and code will be out of sync. As a Model-Driven development sees code as a model itself, equivalent to a design model, it is problematic.
- One cannot fully write a software without writing code at some point. Usually right after the state machine structure definition.

Msm helps solve the first problem as the code can be edited at will, with only the transition table specifying the structure.
The second is much more complicated to solve. There are millions of possible systems, each with its particularities. In some "simpler" systems (like some simple embedded systems) with a reduced set of needs, it is however possible to build almost a whole software through a model-based approach. In more complicated systems with whole algorithms to be implemented, there is little to no help. And yes, let's face it, there will always be the need to write code at some point on many systems.

We can however try to make a few steps further by describing algorithms with models. UML brings some help with activity diagrams. The problem is that they are usually not precise enough to generate code directly from the model so they mostly serve as documentation help but are not useful in a Model-Driven context. State machines being much more precise, can we also provide some algorithm-writing help using state machines? After all, activity diagrams and state machines are sort of cousins.

We are now going to move into a highly experimental field, so the answer might not be totally satisfying but the idea could still deserve some attention.
Let us imagine a search function like it can be found in an iPod. The user can enter a string and the songs/artists/album names in which the string appears are selected. Could we define this simple algorithm as a state machine with enough precision to be able to automatically generate the whole code? In theory yes. For example, we could define the following diagram:

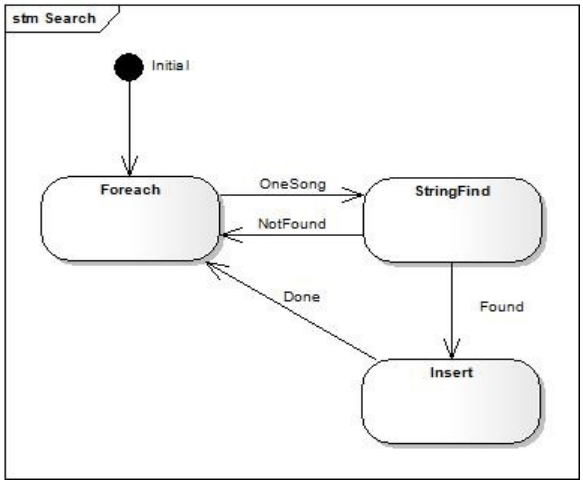


Figure 4: a search function implemented using a state machine

The Foreach state holds a set of the songs present in the player sends the `OneSong` event for every song (remember that states can have their own data). This event contains just one piece of data, a song title. The `StringFind` state has one piece of data, the text entered by the user, and sends a Found event if this piece of data is contained in the event string (a song title), `NotFound` otherwise. The `Insert` state adds the matching song to a result set and sends `Done` when finished. Then, `Foreach` will repeat

until all songs were tested. At the end of a single `process_event` call, the result set containing the candidates would be available.
For documentation purposes, and with some added comments, this would be sufficient. But to generate code?
Msm provides, as previously shown, some help as states do not physically depend on their containing state machine. If `Foreach`, `StringFind` and `Insert` can be defined generically enough (thanks to the STL or Boost), these states can be reused over and over again in different state machines. A simple example implementation is provided as attachment to this paper.
There are a few drawbacks to this approach:

- A bit of code must be provided to link some data of the containing state machine with the algorithmic states. Sounds self-defeating.
- It remains to be proven, that this approach delivers some gain against a standard STL-based coding approach.

The first drawback could be eliminated by a tool who would help the user define the links between the algorithm states and state machine attributes.
The second is harder to answer. Most Boost users would solve this sort of algorithmic problem in a few minutes. But many developers are far from this high standard. And there are still many people, not developers but domain experts who would love the possibility of being able to avoid writing code (probably not completely but to some extent would already be appreciable).
The point is still open, and if necessary, Msm would allow such uncommon use of an UML state machine.
Of course, we used here only a reduced set of algorithmic states. If every algorithm from the STL was implemented, this approach would start becoming more interesting.

4. CONCLUSION

Msm is still a young library and will offer even more in the future as users will hopefully want to get more and more out of their state machines. However, it already supports most of the features UML designers are using every day. Its descriptive nature makes it very simple to implement a state machine diagram or to reverse engineer a state machine written with Msm into a graphical model. Therefore, Msm facilitates a *repeatable* and *reversible* MDD process. Furthermore, it helps improve code quality by helping think about a model instead of code, by reducing the amount of time spent on documentation and by simply reducing the amount of code needed to complete a given task.

Model-Driven-Development will play a bigger role in software development in the future and Domain-Specific-Languages like the one Msm provides will give C++ developers an advantage compared to those using less powerful languages.

5. REFERENCES

[1] Abrahams,D. and Gurtovoy, A.
C++ Template Metaprogramming
Concepts, Tools, and Techniques from Boost and Beyond
2004

[2] Gamma, Helm,Johnson,Vlissides
Design Patterns: Elements of Reusable Object-Oriented Software
1994

[3] Harel, D.
Statecharts: A visual formalism for complex systems 1986

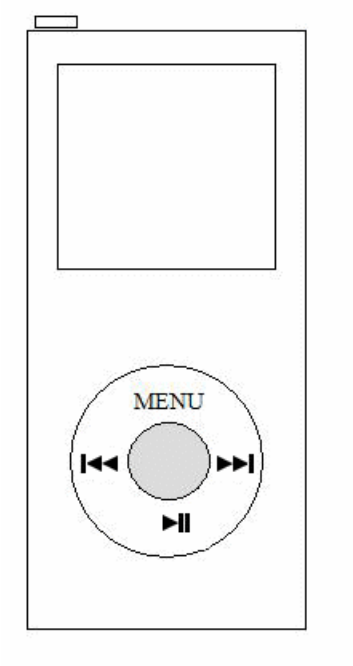
[4] Henry, C.
Boost.Msm (Meta State Machine) documentation
http://www.boostpro.com/vault/index.php?direction=0&order=&directory=Msm&

[5] Huber, A.
Boost.Statechart library

[6] Mellor Scott, Uhl, Weise
MDA Distilled, Principles of Model-Driven-Architecture
2004

[7] OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2

HANDS-ON SESSION



In this session, we will try to design enough of an iPod nano (2nd generation) functionality to provide a mock up of its software.

We will try to provide a specification for the iPod nano by reverse engineering it:

- The "Hold" switch deactivates all the other buttons, which have no effect until the switch is moved from this position.
- The wheel is used to set the volume level, position, etc. and will be left out for the mock up.
- On the wheel are also 4 multifunction buttons to be found, top, bottom, left, right.
- A button is located at the center of the wheel. This button is used either in the menu mode to select a song, or in the playing mode to provide useful features (navigation, rating of songs, etc.).
- The upper button, labeled "Menu" opens the Menu dialog, even while playing, where the user can is provided with a whole range of functions. We will only consider the function allowing to select a song (for example by id) and ignore the others. This means leaving the menu will simply mean choosing a song.
- The bottom button has a triple function, play, pause, and switching off. Pressing once starts playing, a second pressing pauses, etc.
- There is no Off button. Instead, pressing Play/Pause for a longer time will switch off the player. Actually, the player is not really switching off, only the screen and music turn off.
- A short pressing of the Play/Pause button will turn the player on again. So will switching to not Hold.
- Pressing the right button a short time plays the next song. Pressing a long time plays the current song faster.
- The left button is doing the same as the right one, but moves backwards.
- This middle button, used in the playing mode, will be specified as follows: pressing it once will activate a mode where the "cursor" of the currently played song can be set. In this mode, fast forward/backward moving is deactivated. A second pressing of the button will move the player to another mode, where the user can give a mark to the song. Simply using the button will have no influence on the playing/pausing status of the player. We will simply consider these 2 modes where the button can be used.
- This position setting mode is full of surprises: if the cursor is moved to another position in the song, it will (after a short time) move the song to this position. If the middle button is pressed after moving the cursor, the same thing will happen. If the position is not changed, pressing the middle button will move, as above described, to the rating mode.
- Whatever mode is triggered by the middle button while playing, it will close after 5s without user action.
- If the player was playing at the time it was switched off, it will restart in paused mode. Other playing modes will wake up active.
- At the end of playing (when the last song is finished), the playing mode is exited and the menu screen becomes active.
- The same happens while moving the player past the last song of before the first song with the forward/backward buttons.
- While in paused mode, it is possible to choose another song using the menu. In this case, the selected song is automatically started.

All this sounds like an awful lot of work if one has to program all this with conventional methods. Furthermore, this is only a simplified subset of the features provided by an iPod. Designing only the main structure of such a software could take a long time.

Fortunately, using state machines and a few useful features from Msm will make this a quite manageable problem. We will try to provide a mock up in only 90 minutes.