# Asynchronous Programming with The Meta State Machine Library

# (MSM)

# Contents

# A New Project

- Specification
- Design
- Code
- Test
- Documentation

# A New Project

At the end of the project, we have:

- A Working Product
- Documentation, User Manual
- We are under budget and done early
- The maintenance team is trained and takes over smoothly

# What we really have

# What went wrong?

- At the beginning, all is well
- Some documentation is even written
- Then come the first milestones
- Code has high prio, documentation no.
- Even for code, just „get it to work“
- Iterate a few milestones

# Why State Machines?

State machines help us:

- Design
- Document
- Debug
- Think asynchronously

# Our Pattern Of The Day

- A Manager implemented as a state machine runs in its own thread
- The Manager is non-blocking.
- Target hardware is controlled asynchronously and lives in other threads or machine.

To achieve this we need:

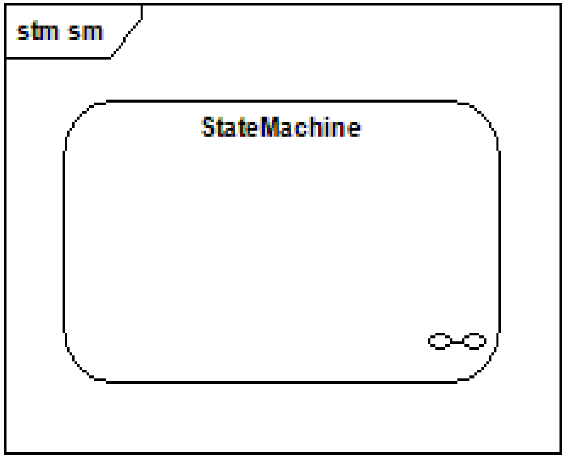
- A state machine library
- Tools to manage asynchronous behavior.

# MSM Design

- Separation between front- and back-end
- So far one back-end
- Three front-ends:
  - Basic: deprecated
  - Functor: more powerful
  - eUML: experimental proto-based DSEL
- Principles:
  - Declarativeness (focus on structure rather than implementation)
  - Expressiveness (easy to understand syntax)
  - Efficiency (no an excuse for ad-hoc logic)

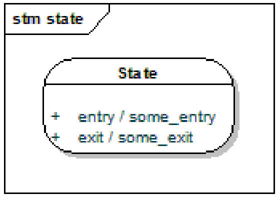
# UML State Machine concepts: state machine

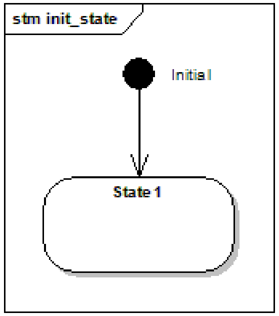
- Describes the behavior of a system
- Composed of a finite number of states and transitions
- Can have data, entry/exit behavior, internal transitions



# UML State Machine concepts: state

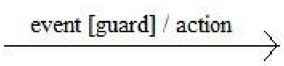
- Has no substates
- Can have data, entry/exit behavior, internal transitions
- An initial state is marked using an initial pseudo-state



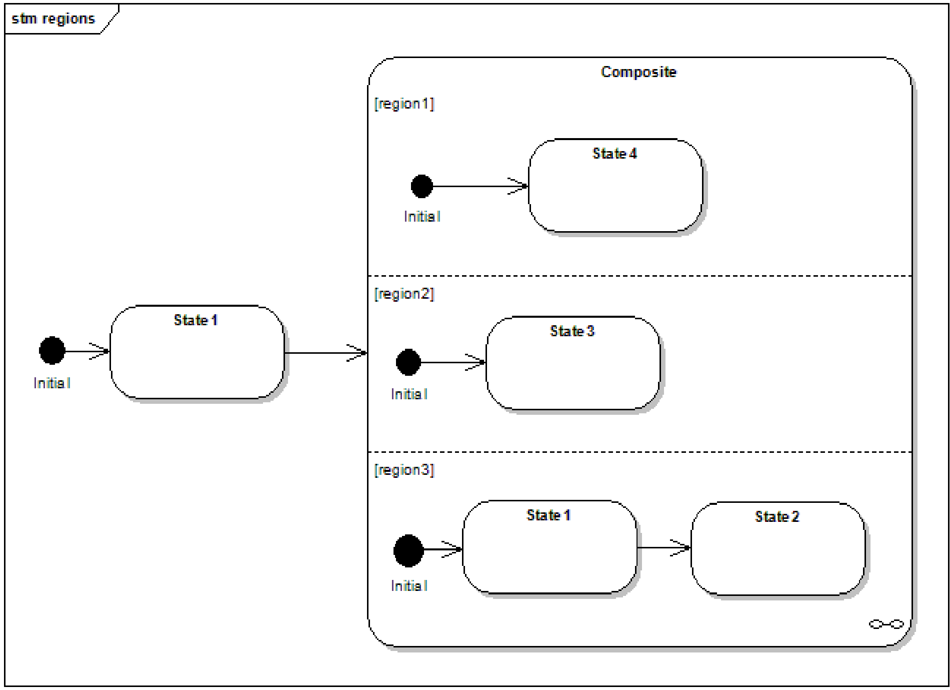
# UML State Machine concepts: transition

- Switching between active states
- Triggered by an event
- Can have actions / guard condition



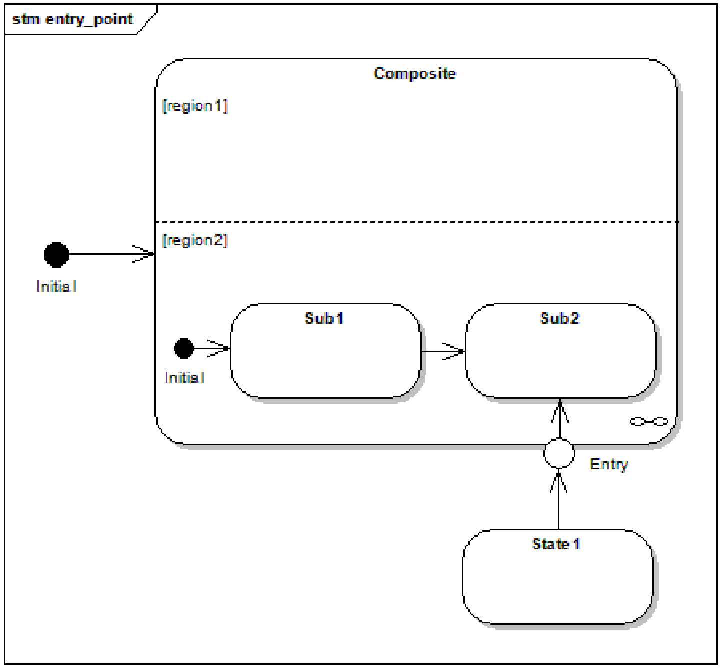
# UML State Machine concepts: sub machines, regions

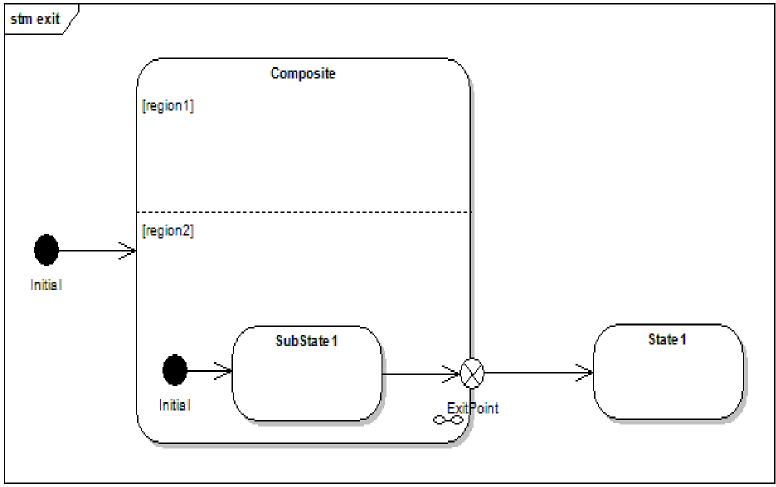
- A submachine is a state machine inserted in another state machine
- Can be inserted more than once
- Orthogonal regions are (logically) concurrent parts of the state machine



# UML State Machine concepts: sub machines, regions (2)

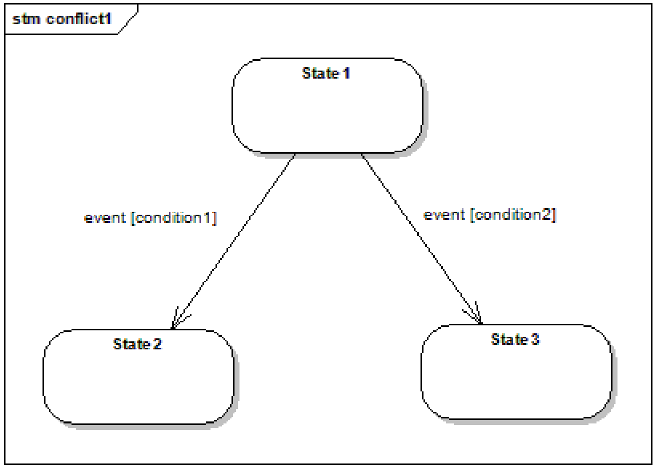
- An entry point connects a transition outside the submachine to one inside
- Allows only one region to be entered
- An exit point also connects two transitions
- Forces termination of all regions

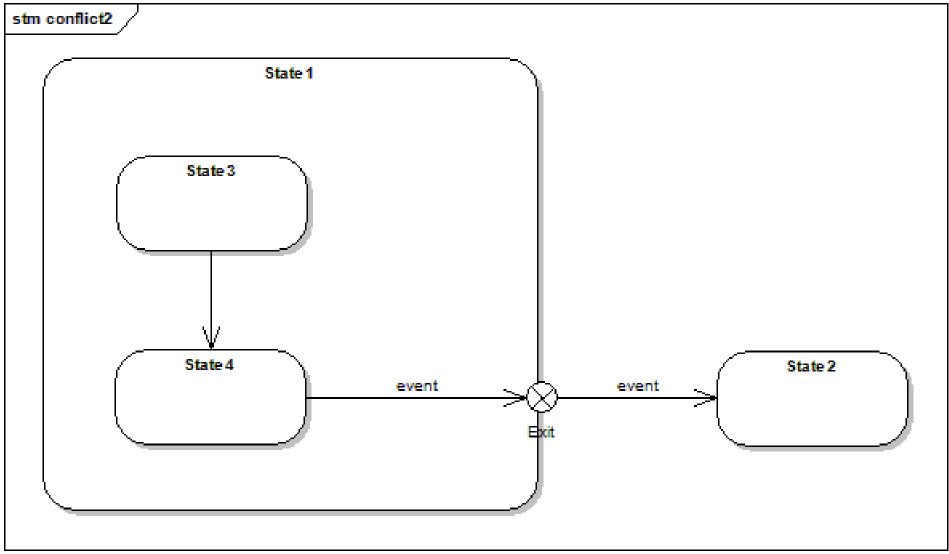


# UML State Machine concepts: conflicting transitions

- Happens if, for a given event, several transitions are possible
- Conflict can be between an internal and an external transition
- Conflict can be between an outer transition and one from a nested submachine
- First case solved by mutually exclusive guards
- If not exclusive, undefined behavior.
- In the second case, internal tried first
- In the third, deepest tried first.



# UML State Machine concepts: Deferred events

- A state can mark events as deferred
- If this state is active and an event is processed, it is deferred until the machine is in a state where it can be processed
- MSM allows you to do it this way or through the transition table (conditional deferring is then possible).

# UML State Machine concepts: more stuff from MSM

- Terminate, Interrupt States
- Different Terminate States
- Internal transitions
- State Machine Structure Checking
- Message queues
- Deferred events as transitions
- Flags
- Functioning History concept
- Tons of transition capabilities
- Several DSELs
- …

# Functor Front-End

- *Define a state machine using the front-end*
- *Pick a back-end, which will be the concrete state machine type:*

```
typedef
msm::back::state_machine<front_end> fsm;
```

# Functor Front-End

- *Based on a transition table:*

```
struct transition_table : mpl::vector<
//    Source Event       Target  Action    Guard
//  +------+----------+-------+------+----------+
Row < Open ,open_close, Empty ,none  ,some_guard>,
Row < Open ,open_close, Empty ,none  ,other_guard>,
…
> {};
```

⇒ *Readability*

⇒ *Lesser need to search the code*

⇒ *More information than a diagram*

⇒ *Defined way to solve conflicts*

# Functor Front-End

*Rows for every possible use:*

- *Row<source,event,target,* ***none,none****>* => *no action, no guard*
- *Row<source,event,target,* ***action,guard****>* => *action and guard*
- *Row<source,event,target,* ***action,none****>* => *no guard*
- *Row<source,event,target,* ***none,guard****>* => *no action*
- *Row<state,event,****none****, action,guard>* => *internal (2nd form)*
- *Row<state,* ***none****,target, action,guard>* => *anonymous transition*
- *Row<state,* ***any****,target, action,guard>* => *Kleene (any event)*
- *Row<Sub::exit_pt<PseudoExit>,…>* => *pseudo exit*

# Functor Front-End

- *More powerful constructs:*

```
struct transition_table : mpl::vector<
Row < Open ,open_close, Empty
  // actions
  ,ActionSequence_<mpl::vector<close_drawer,other_action> >
  // guard
  ,And_<condition1,condition2> >,
…
> {};
```

# eUML

- *Also based on a transition table:*

```
BOOST_MSM_EUML_TRANSITION_TABLE((
Playing  == Stopped  + play [some_guard] / start_playback ,//1
Stopped + open_close [other_guard]/ open_drawer == Open,//2
…
),transition_table)
```

⇒ *Better readability*

⇒ *No visible Boost.MPL*

⇒ *UML syntax*

# eUML

*Row definition for every possible use:*

- `source + event == target` => *no action, no guard*
- `(target == source + event` => *no action, no guard)*
- `source + event / action == target` => *no guard*
- `source + event [guard] == target` => *no action*
- `source + event [guard]/action == target` => *action and guard*
- `source + event [guard]/action` => *internal (2nd form)*
- `source [guard]/action == target` => *anonymous transition*
- `source` **+ *kleene*** `== target` => *Kleene (matches any) event*
- `(explicit_(Sub,target1), explicit_(Sub,target2))` => *fork*
- `exit_pt_(Sub,PseudoExit) + … =…` => *transition from pseudo exit*

# eUML

- *More powerful constructs:*

```
BOOST_MSM_EUML_TRANSITION_TABLE((
Open + open_close
[condition1 && condition2] / (close_drawer,other_action) == Empty
…
),transition_table)
```

# Asynchronous Programming

**std::async / boost::async**

```
std::future<int> f = std::async([](){return 42;}); // executes asynchronously
int res = f.get(); // wait for result, block until ready
```

Simple, but...

► Blocking is bad for state machines (no run to completion).

► Blocking prevents diagnostics.

► Blocking makes your program less responsive.

► Blocking reduces opportunities to parallelize.

Waiting is ok, blocking no.

# Asynchronous Programming

**std::async / boost::async**

We have for alternatives:

- *Block while waiting*
- *Poll*
- *Carry a bag of futures*

# Asynchronous Programming

**std::async / boost::async**

Do you spot a problem?

```
{
   std::async(std::launch::async, []{ f(); });
   std::async(std::launch::async, []{ g(); });
}
```

► 2nd line does not run until f() completes

# Asynchronous Programming

- Better with N3558 / N3650?

```
future<int> f1 = async([]() { return 123; });
future<string> f2 = f1.then([](future<int> f)
{
  return f.get().to_string(); // here .get() won't block
});
// and here?
string s= f2.get();
```

# Asynchronous Programming

Boost.Asio. Example:

```
// won't block
boost::asio::async_write(socket_, request_,
            boost::bind(&client::handle_write_request, this,
                        boost::asio::placeholders::error));

// callback, possibly much later
void handle_write_request(const boost::system::error_code& /*error*/)
{…}
```

# Asynchronous Programming

Boost.Asio. Disadvantages:

- Object lifetime
- Managing asynchrony
- No interrupting capability

# Asynchronous Programming

Maybe something like this would be better?

```
post_callback(
   threadpool_scheduler,          // where the task will execute
   [](){return 42;}               // long-lasting task
   my_scheduler,                  // my thread
   [](boost::future<int> res){...} // callback
);
```