The Meta State Machine Library V2

(MSM)

Contents

- Model-Driven vs. Classical methods
- MSM Design
- State machine concepts
- Functor front-end
- eUML
- Examples

Conventional development process (for example Unified Process): theory

- Requirements engineering
- Analysis
- Design
- Implementation
- Test
- Deployment
- UP is *iterative* and *incremental*

Conventional development process (for example Unified Process): theory

At the end of the project:

- We have a product
- We have documents: requirements, analysis, design
- And of course we are under budget and finish early
- The maintenance team is trained correctly and takes over smoothly

Kidding!

Conventional development process (real-world)

At the end of the project:

- We usually have some kind of product (not the one we planned)
- There is no documentation or it is either hopelessly outdated or automatically generated.
- The project is late and well above budget
- The maintenance team was never trained and most key players are gone.
- What the maintenance team inherits:

Conventional development process (result)

What happened?

- At the beginning, all is well, developers are excited
- Requirements, analysis and design are (often) written
- Then come the first milestones, and the corresponding pressure
- Code has a higher priority and documents lag behind.
- Repeat a few times (milestones) and documents fall into oblivion
- We like to compare software with architecture. Does it reflect reality?

A Model-Driven-Process

- Requirements, Analysis, Design, Code are all models
- Mapping functions transform a model into another model
- Mapping functions are necessary for the process to work!
- Should be *repeatable* and *reversible*

A Model-Driven-Process

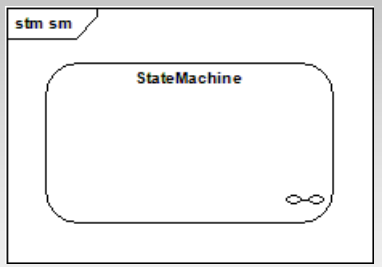
- Static view of a software: class diagram
  - Easy to generate code from model
  - Slightly harder to generate a model from code
- Dynamic view of a software: state machines, activity diagrams, etc.
  - Easy to generate code from model
  - Much more complicated to generate a model from code
  - Even harder to synchronize code and model
- MSM provides a solution for the dynamic view

MSM Design

- Separation between front- and back-end
- So far one back-end
- Three front-ends:
  - Basic: roughly MSM v1
  - Functor: more powerful
  - eUML: experimental proto-based DSEL
- Principles:
  - Declarativeness
  - Expressiveness
  - Efficiency

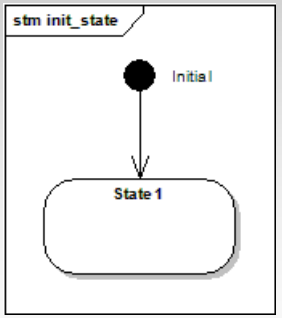
UML State Machine concepts: state machine

- Describes the behavior of a system
- Composed of a finite number of states and transitions



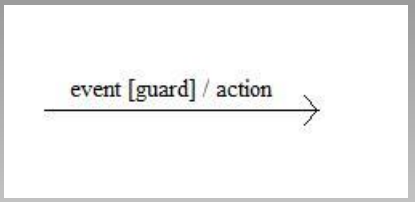
UML State Machine concepts: state

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



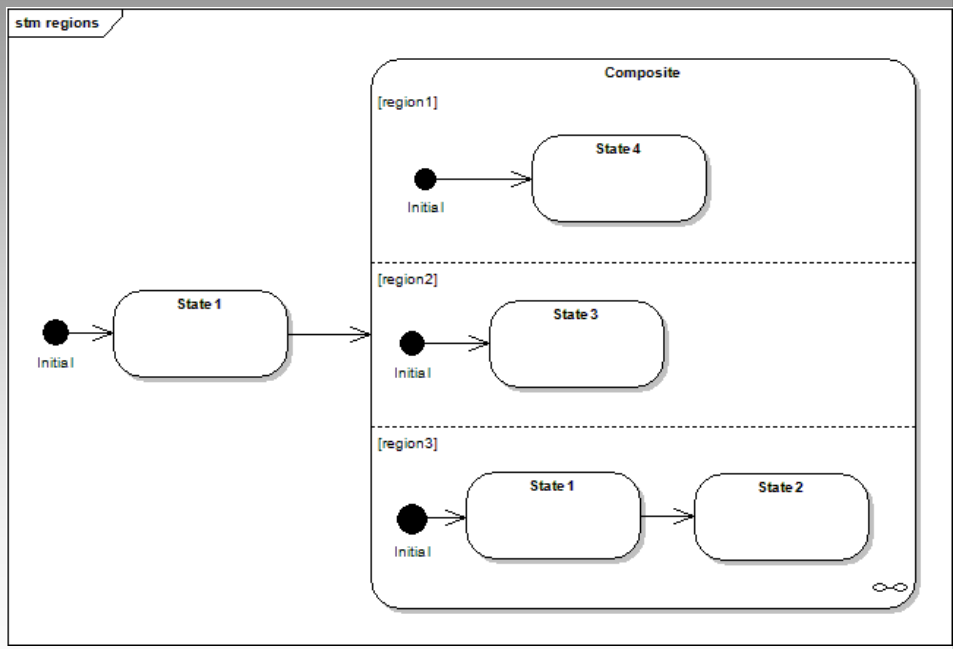
UML State Machine concepts: transition

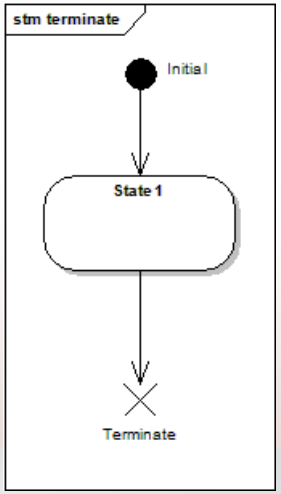
- Switching between active states
- Triggered by an event
- Can have action / guard



UML State Machine concepts: sub machines, regions

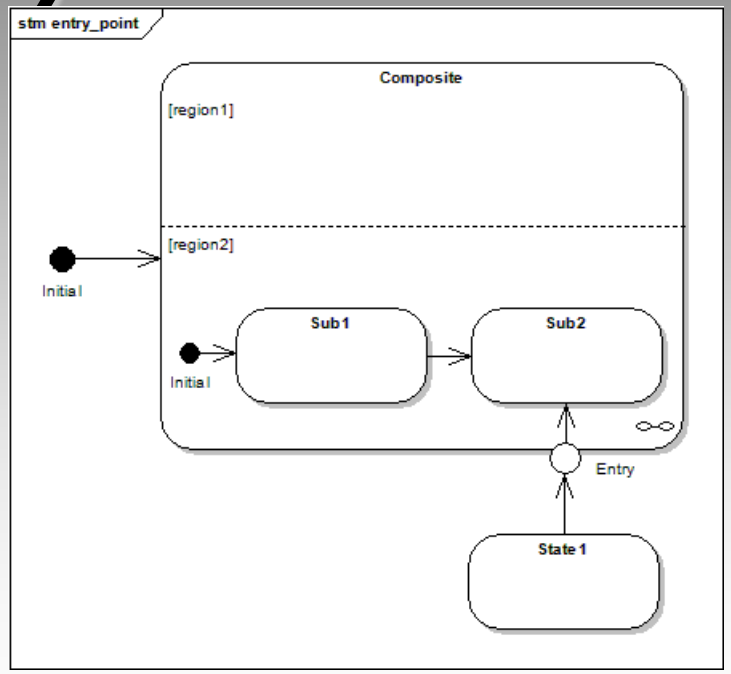
- A submachine is a state machine inserted in another state machine
- Can be inserted more than once
- Orthogonal regions are concurrent parts of the state machine
- The terminate pseudo-state terminates the execution of the state machines (in all regions)

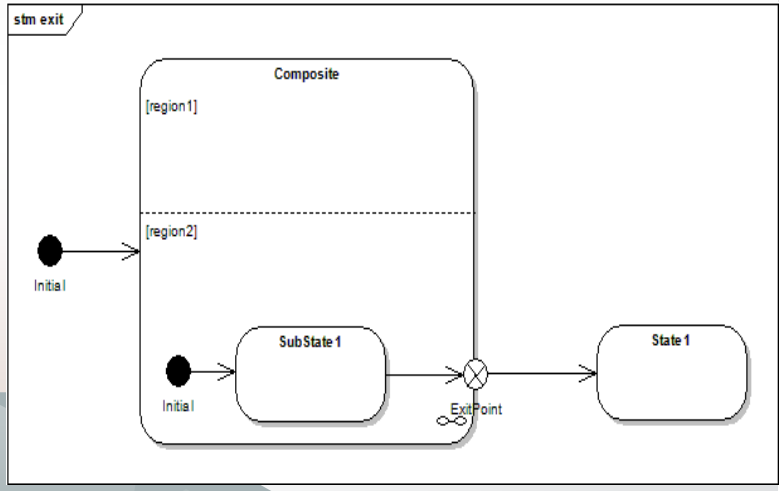




UML State Machine concepts: sub machines, regions (2)

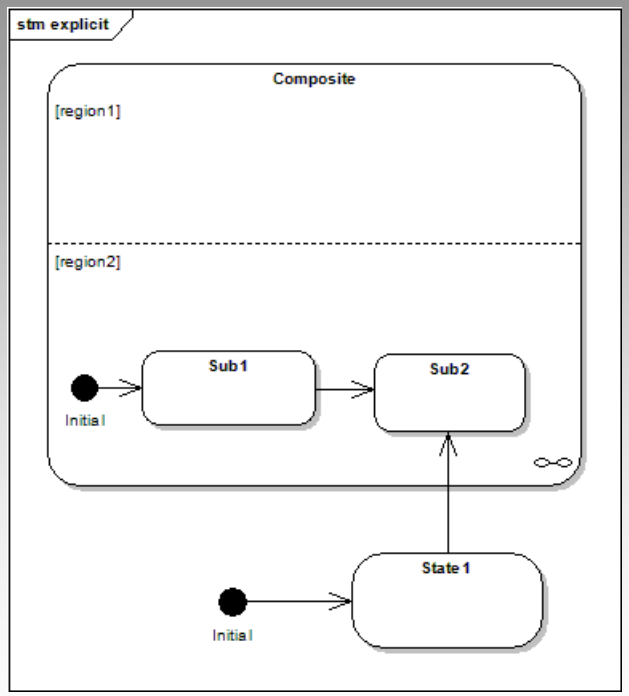
- An entry point connects a transition outside the submachine to one inside
- Allows only one region to be entered
- An exit point also connects two transitions
- Forces termination of all regions

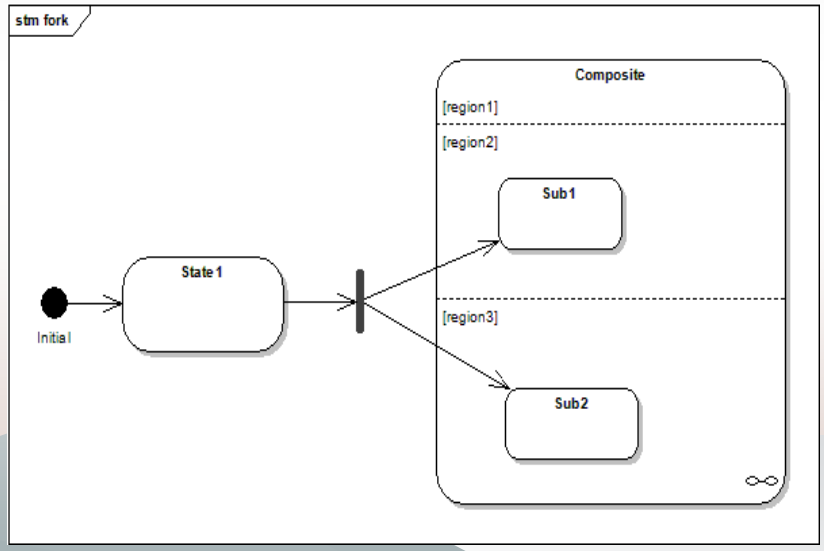




UML State Machine concepts: sub machines, regions (3)

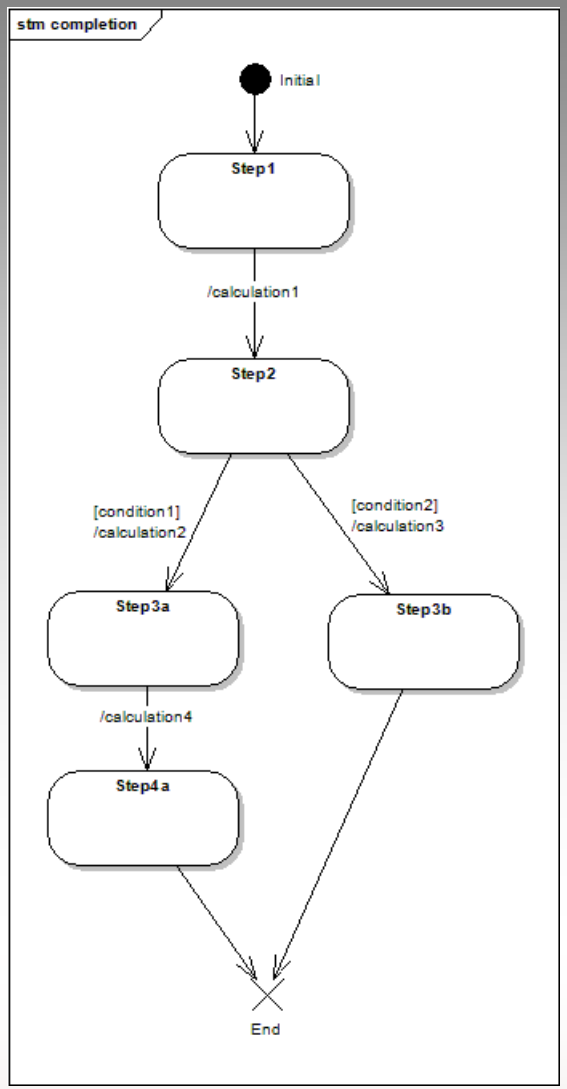
- An explicit entry is a single transition ending on a substate of the composite
- Allows only one region to be entered
- A fork is an explicit entry into several regions
- Other regions are entered using the initial state





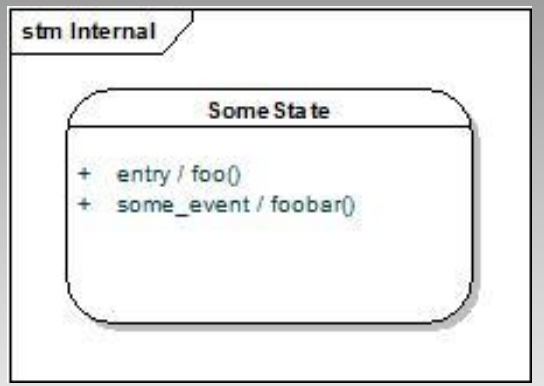
UML State Machine concepts: completion events

- Transitions without a named event
- Fire immediately (guard permitting)
- Can be used to model algorithms (no need of activity diagrams)



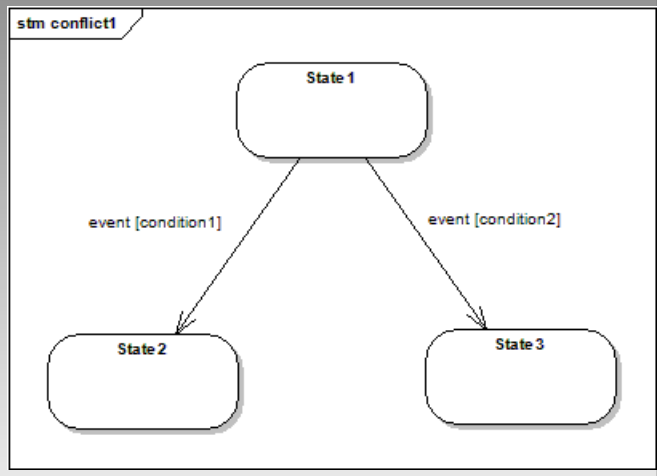
UML State Machine concepts: Internal transitions

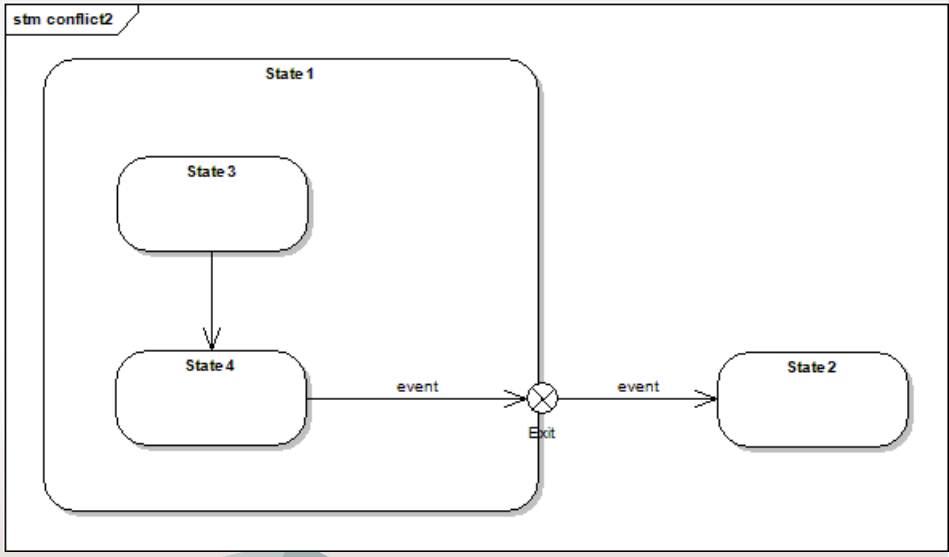
- Execute in the scope of a state
- Self-transition without executing entry/exit behaviors



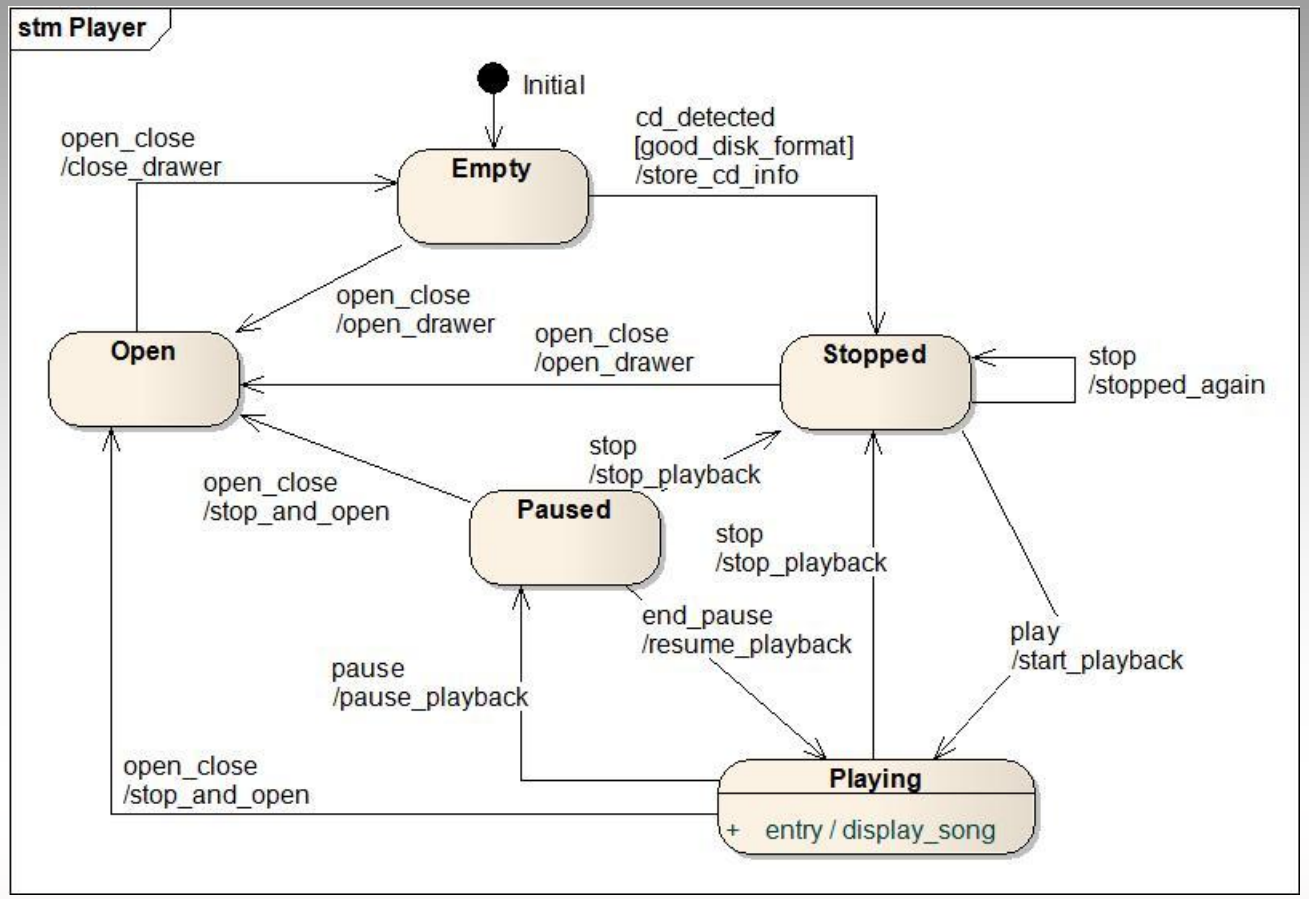
UML State Machine concepts: conflicting transitions

- Happens if for a given event, several transitions are enabled
- Conflict can be between an internal and an external transition
- First case solved by guards
- In the second case, the internal is tried first





A simple state machine



Implementation with the functor front-end: (1)

- *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;
```

Implementation with the functor front-end: (2)

- *State machine front-end definition:*

```
struct player_ : public
  msm::front::state_machine_def<player_>{…
  };
```

- *Initial state:*

```
typedef Empty initial_state;
```

- *Configuration:*

```
typedef int no_message_queue;
```

Implementation with the functor front-end: (3)

- *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*

Implementation with the functor front-end: (4)

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*

Implementation with the functor front-end: (5)

- *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> >,
…
> {};
 struct close_drawer{
 template<class Evt,class Fsm,class Source,class Target>
 void operator()(Evt const&,Fsm&,Source&,Target&){…}
 };
```

Implementation with the functor front-end: (6)

- *Events: any type:* `struct some_event{};`
- *States:*
  - *inherit from* `msm::front::state<>`
  - *Can have entry and exit behaviors, on_entry/on_exit:*

```
template <class Event,class Fsm>
void on_entry(Event const&,Fsm& ){…}
```

  - *Can defer events:*

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

  - *Can have internal transitions as… a transition table:*

```
struct internal_transition_table : mpl::vector<
Internal< some_event, some_action, some_guard>
>{};
```

Implementation with the functor front-end: (7)

- *Submachines: a state appearing inside the transition table and being a* `msm::back::state_machine<…>`
- *Regions: more initial states:*

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

- *History: made a back-end policy:*
  - *NoHistory(default): initial state is activated with the submachine*
  - *ShallowHistory: (almost)UML way. Activated by a list of events*

```
typedef msm::back::state_machine<Playing_,
   msm::back::ShallowHistory<mpl::vector<some_event> > >
```

  - *AlwaysHistory: last active state always activated with submachine*

```
typedef msm::back::state_machine<Playing_,
   msm::back::AlwaysHistory>
```

Implementation with the functor front-end: (8)

- *Terminate state: inherit from* `front::terminate_state<>`
- *Interrupt state: inherit from* `front::interrupt_state<event_name>`

- *Entry pseudo-state:* `front::entry_pseudo_state<int region>`
- *Exit pseudo-state:* `front::exit_pseudo_state<forwarded_event>`
- *Explicit entry state:* `front::explicit_entry<int region>`
- *To reference these states inside the containing fsm, use entry_pt, exit_pt, direct:*

```
Row<State1,event2,SubFsm2::direct<SubState2> >
```

Or for a fork:

```
Row<State1,event2,mpl::vector<SubFsm2::direct<Sub2>,
                              SubFsm2::direct<Sub2b> >
```

Implementation with eUML: (1)

- *Define a state machine using the eUML front-end*
- *Same back-end:*

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

Implementation with eUML: (2)

- *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*

Implementation with eUML: (3)

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*
- `(explicit_(Sub,target1), explicit_(Sub,target2))` => *fork*

Implementation with eUML: (4)

- *More powerful constructs:*

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

 BOOST_MSM_EUML_ACTION(close_drawer)
 {
 template<class Evt,class Fsm,class Source,class Target>
 void operator()(Evt const&,Fsm&,Source&,Target&){…}
 };
```

Implementation with eUML: (5)

- *Events:* `BOOST_MSM_EUML_EVENT(some_event)`
- *States:*
  - `BOOST_MSM_EUML_STATE((/* state expression */),some_state)`
  - *Can have entry and exit behaviors:*

```
BOOST_MSM_EUML_STATE((my_entry, my_exit),some_state)


BOOST_MSM_EUML_ACTION(my_entry)
{
template<class Evt, class Fsm, class State>
void operator()(Evt const&,Fsm&,State&){…}
};
```

Implementation with eUML: (6)

- *Can defer events:*

```
BOOST_MSM_EUML_STATE((my_entry, my_exit,
    attributes_ << no_attributes_, configure_ << some_event),
 some_state)
```

- *Can have internal transitions:*

```
BOOST_MSM_EUML_STATE((my_entry, my_exit),state_def)
struct state_impl : state_def
{
  BOOST_MSM_EUML_DECLARE_INTERNAL_TRANSITION_TABLE((
    some_event [some_guard] / some_action,…
  ))
};
state_impl const some_state;
```

Implementation with eUML: (7)

We need a way to define attributes

- *Declare a name and type*

```
BOOST_MSM_EUML_DECLARE_ATTRIBUTE(int,m_attribute)
```

- *Add to a state (events have another macro)*

```
BOOST_MSM_EUML_STATE((my_entry, my_exit,
      attributes_ << m_attribute),
    some_state)
```

- *Use it:*

```
++source_(m_attribute)
source_(m_attribute)=target_(m_attribute)+event_(m_attribute)
```

Are valid actions.

Implementation with eUML: (8)

- *State machines:*

```
BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
  (stt,init,entry,exit,attributes,configuration,
   no_transition),my_fsm_frontend)
```

- *Regions: more initial states:* `init_ << state1 << state2`

Implementation with eUML: (9)

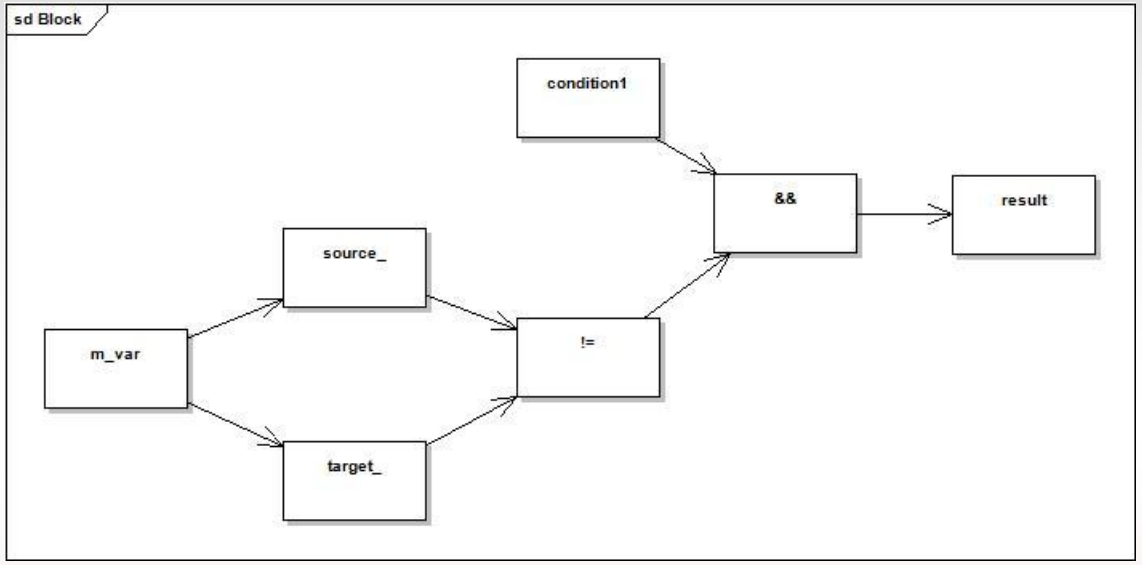
- *Built-in support for STL functions / algorithms:*

```
[first_(
        mismatch_(begin_(fsm_(m_vec1)),end_(fsm_(m_vec1)),
                  begin_(fsm_(m_vec2)))
       ) == end_(fsm_(m_vec1))]
```

Implementation with eUML: (10)

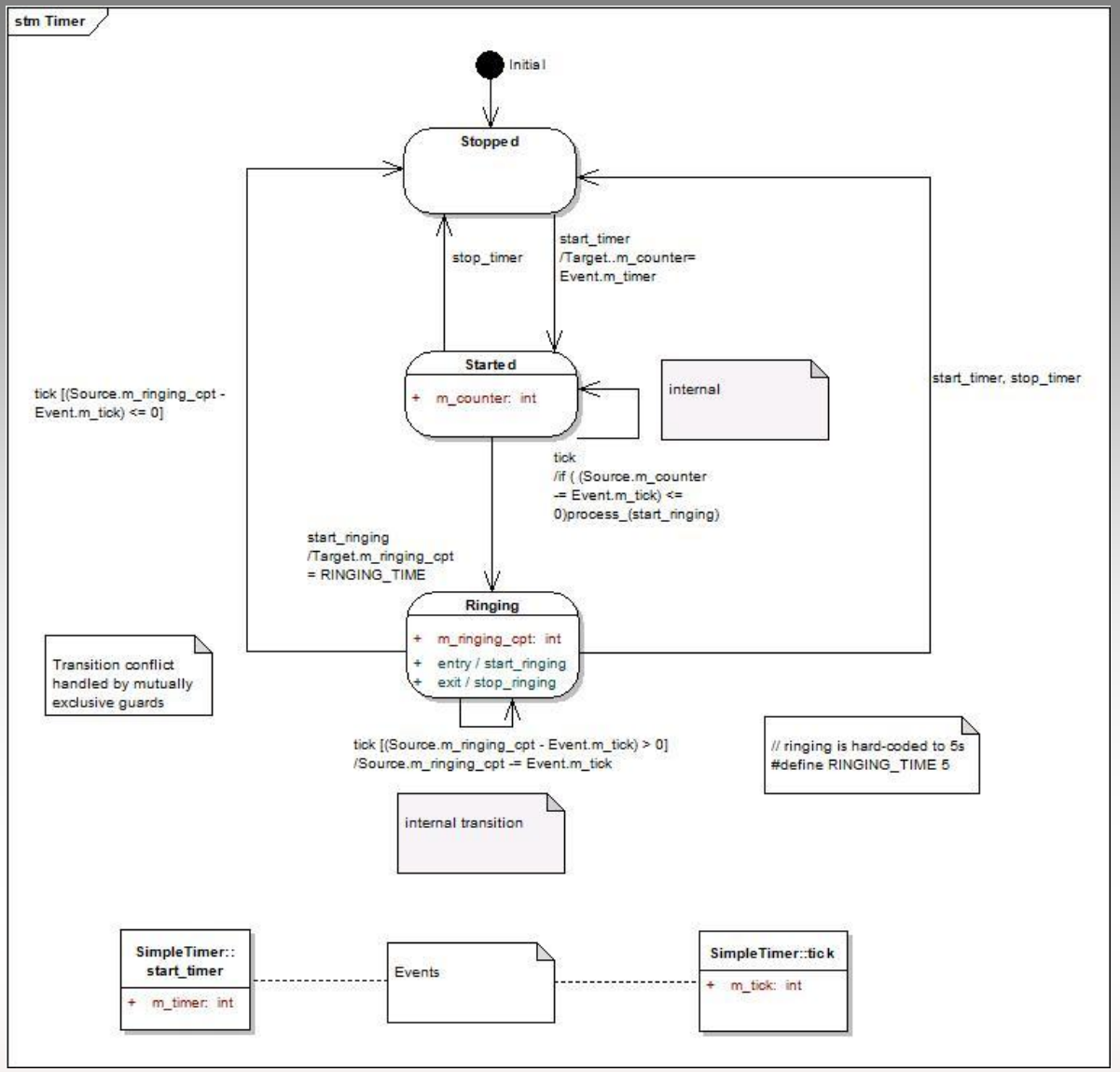
Easy to model

[condition1 && (source_(m_var) != target_(m_var))]



Example: a simple timer

- *Timer started by a start event containing the timer value*
- *Timer stopped by a stop button*
- *Hardware supports tick events coming at imprecise intervals containing time elapsed since last tick*
- *Ringing time hard-coded*



Example: parsing

Regex (from Xpressive doc):

```
[[:digit:]]{4}[- ]){3}[[:digit:]]{3,4}
```

Parsed string:

```
1234-5678-1234-456
```

Waiting→Digit1→Digit2→Digit3→Digit4→MinusChar1→…→Digit15→Parsed.

- ⇒ Fast in some cases, slower in others
- ⇒ Could be greatly improved
- ⇒ Needs a special front-end (for example Xpressive or Ragel)

“Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.”

Winston Churchill