The boost::fsm library

Tutorial

Contents

Introduction
Hello World!
A stop watch
- Defining states and events
- Adding reactions
- State-local storage
- Getting state information out of the machine

A digital camera
- Spreading a state machine over multiple translation units
- Guards
- In-state reactions (aka inner transitions)
- Transition actions

Advanced topics
- Reaction function reference
- Reaction reference
- Specifying multiple reactions for a state
- Posting events
- Deferring events
- History
- Orthogonal states
- Exception handling
- Submachines & Parametrized States
- Asynchronous state machines

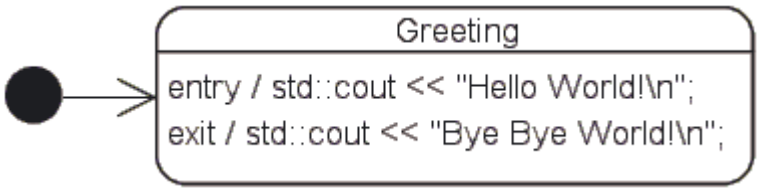
Introduction

The boost::fsm library is a framework that allows you to quickly transform a UML state chart into executable C++ code. This tutorial requires some familiarity with the state machine concept and UML state charts. A nice introduction to both can be found in http://www.objectmentor.com/resources/articles/umlfsm.pdf. David Harel, the inventor of state charts, presents an excellent tutorial-like but still thorough discussion in his original paper: http://www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/Statecharts.pdf. The UML specifications can be found at http://www.omg.org/cgi-bin/doc?formal/03-03-01 (see chapters 2.12 and 3.74).

All examples have been tested with MSVC7.1 and boost distribution 1.30.2.

Hello World!

We follow the tradition and use the simplest possible program to make our first steps. We will implement the following state chart:




```
#include <boost/fsm/state_machine.hpp>
#include <boost/fsm/simple_state.hpp>
#include <iostream>

namespace fsm = boost::fsm;

struct Greeting;
struct Machine : fsm::state_machine< Machine, Greeting > {};

struct Greeting : fsm::simple_state< Greeting, Machine >
{
  Greeting() { std::cout << "Hello World!\n"; } // entry
  ~Greeting() { std::cout << "Bye Bye World!\n"; } // exit
};

int main()
{
  Machine myMachine;
  myMachine.initiate();
  return 0;
}
```

This program prints `Hello World!` and `Bye Bye World!` before exiting. The first line is printed as a result of calling `initiate()`, which leads to the `Greeting` state begin entered. At the end of `main()`, the `myMachine` object is destroyed what automatically exits the `Greeting` state.

A few remarks:

- boost::fsm makes heavy use of the curiously recurring template pattern. The deriving class must always be passed as the first parameter to the base class template.
- The machine is not yet running after construction. We start it by calling `initiate()`.
- All states reside in a context. For the moment, this context is the state machine. That's why `Machine` is passed as the second template parameter of `Greeting`'s base.
- The state machine must be informed which state it has to enter when the machine is initiated. That's why `Greeting` is passed as the second template parameter of `Machine`'s base. We have to forward declare `Greeting` for this purpose.
- We are declaring all types as `struct`s only to avoid having to type `public`. If you don't mind doing so, you can just as well use `class`.

A stop watch

Next we will model a simple mechanical stop watch with a state machine. Such watches typically have two buttons:

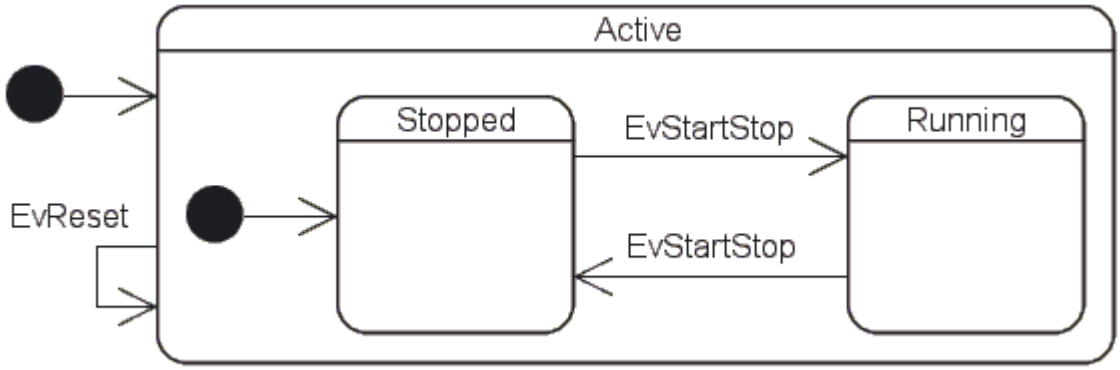
- Start/Stop

- Reset

And two states:

- Stopped: The hands reside in the position where they were last stopped.
  - Pressing the reset button moves the hands back to the 0 position. The watch remains in the Stopped state.
  - Pressing the start/stop button leads to a transition to the Running state.
- Running: The hands of the watch are in motion and continually show the elapsed time.
  - Pressing the reset button moves the hands back to the 0 position and leads to a transition to the Stopped state.
  - Pressing the start/stop button leads to a transition to the Stopped state.

Here is one way to specify this in UML:



Defining states and events

The two buttons are modeled by two events. Moreover, we also define the necessary states and the initial state. **The following code is our starting point, subsequent code snippets must be inserted**:

```
#include <boost/fsm/event.hpp>
#include <boost/fsm/state_machine.hpp>
#include <boost/fsm/simple_state.hpp>

namespace fsm = boost::fsm;

struct EvStartStop : fsm::event< EvStartStop > {};
struct EvReset : fsm::event< EvReset > {};

struct Active;
struct StopWatch : fsm::state_machine< StopWatch, Active > {};

struct Stopped;
struct Active : fsm::simple_state< Active, StopWatch,
  fsm::no_reactions, Stopped > {};
struct Running : fsm::simple_state< Running, Active > {};
struct Stopped : fsm::simple_state< Stopped, Active > {};

int main()
{
  StopWatch myWatch;
```

```
  myWatch.initiate();
  return 0;
}
```

This compiles but doesn't do anything observable yet. A few comments:

- The `simple_state` class template accepts up to four parameters.
  - The third parameter specifies reactions (explained in due course). Because there aren't any yet, we pass `fsm::no_reactions`, which is also the default.
  - The fourth parameter specifies the inner initial state, if there is one.
- A state is defined as an inner state simply by passing its outer state as its context (where outermost states pass the state machine).
- Because the context of a state must be a complete type (i.e. not forward declared), a machine must be defined from "outside to inside". That is, we always start with the state machine, followed by outermost states, followed by the inner states of outermost states and so on. We can do so in a breadth-first or depth-first way or employ a mixture of the two.
  Since the source and destination state of a transition often have the same nesting depth, the pure depth-first approach tends to require a lot of forward declarations for transition destinations while the pure breadth-first approach tends to minimize the number of necessary forward declarations.

Adding reactions

A reaction is anything that happens as the result of the processing of an event. For the moment we will use only one type of reaction: transitions. We **insert** the bold part of the following code:

```
#include <boost/fsm/transition.hpp>

// ...

struct Stopped;
struct Active : fsm::simple_state< Active, StopWatch,
  fsm::transition< EvReset, Active >, Stopped > {};
struct Running : fsm::simple_state< Running, Active,
  fsm::transition< EvStartStop, Stopped > > {};
struct Stopped : fsm::simple_state< Stopped, Active,
  fsm::transition< EvStartStop, Running > > {};

int main()
{
  StopWatch myWatch;
  myWatch.initiate();
  myWatch.process_event( EvStartStop() );
  myWatch.process_event( EvStartStop() );
  myWatch.process_event( EvStartStop() );
  myWatch.process_event( EvReset() );
  return 0;
}
```

A state can define an arbitrary number of reactions. That's why we have to put them into an `mpl::list<>` as soon as there is more than one of them (see Specifying multiple reactions for a state).
Now we have all the states and all the transitions in place and a number of events are also sent to

the stop watch. The machine dutifully makes the transitions we would expect, but no actions are executed yet.

State-local storage

Next we'll make the stop watch actually measure time. Depending on the state the stop watch is in, we need different variables:

- Stopped: One variable holding the elapsed time
- Running: One variable holding the elapsed time **and** one variable storing the point in time at which the watch was last started.

We observe that the elapsed time variable is needed no matter what state the machine is in. Moreover, this variable should be reset to 0 when we send an `EvReset` event to the machine. The other variable is only needed while the machine is in the Running state. It should be set to the current time of the system clock whenever we enter the Running state. Upon exit we simply subtract the start time from the current system clock time and add the result to the elapsed time.

```
#include <ctime>

// ...

struct Stopped;
struct Active : fsm::simple_state< Active, StopWatch,
  fsm::transition< EvReset, Active >, Stopped >
{
  public:
    Active() : elapsedTime_( 0 ) {}
    std::clock_t ElapsedTime() const { return elapsedTime_; }
    std::clock_t & ElapsedTime() { return elapsedTime_; }
  private:
    std::clock_t elapsedTime_;
};

struct Running : fsm::simple_state< Running, Active,
  fsm::transition< EvStartStop, Stopped > >
{
  public:
    Running() : startTime_( std::clock() ) {}
    ~Running()
    {
      context< Active >().ElapsedTime() +=
        ( std::clock() - startTime_ );
    }
  private:
    std::clock_t startTime_;
};

// ...
```

Similar to when a derived class object accesses its base class portion, `context<>()` is used to gain access to a direct or indirect outer state object. The same function could be used to access the state machine (here `context< StopWatch >()`). The rest should be mostly self-explanatory.

The machine now measures the time, but we cannot yet retrieve it from the main program.

Getting state information out of the machine

To retrieve the measured time, we need a mechanism to get state information out of the machine. With our current machine design there are two ways to do that. For the sake of simplicity we use the less efficient one: `state_cast<>()`. As the name suggests, the semantics are very similar to the ones of `dynamic_cast`. For example, when we call `myWatch.state_cast< const Stopped & >()` **and** the machine is currently in the Stopped state, we get a reference to the `Stopped` state. Otherwise `std::bad_cast` is thrown. We can use this functionality to implement a `StopWatch` member function that returns the elapsed time. However, rather than ask the machine in which state it is and then switch to different calculations for the elapsed time, we put the calculation into the Stopped and Running states and use an interface to retrieve the elapsed time:

```
#include <iostream>

// ...

struct IElapsedTime
{
  virtual std::clock_t ElapsedTime() const = 0;
};

struct Active;
struct StopWatch : fsm::state_machine< StopWatch, Active >
{
  std::clock_t ElapsedTime() const
  {
    return state_cast< const IElapsedTime & >().ElapsedTime();
  }
};

// ...

struct Running : IElapsedTime, fsm::simple_state<
  Running, Active, fsm::transition< EvStartStop, Stopped > >
{
  public:
    Running() : startTime_( std::clock() ) {}
    ~Running()
    {
      context< Active >().ElapsedTime() = ElapsedTime();
    }

    virtual std::clock_t ElapsedTime() const
    {
      return context< Active >().ElapsedTime() +
        std::clock() - startTime_;
    }
  private:
    std::clock_t startTime_;
};
```

```
struct Stopped : IElapsedTime, fsm::simple_state<
  Stopped, Active, fsm::transition< EvStartStop, Running > >
{
  virtual std::clock_t ElapsedTime() const
  {
    return context< Active >().ElapsedTime();
  }
};

int main()
{
  StopWatch myWatch;
  myWatch.initiate();
  std::cout << myWatch.ElapsedTime() << "\n";
  myWatch.process_event( EvStartStop() );
  std::cout << myWatch.ElapsedTime() << "\n";
  myWatch.process_event( EvStartStop() );
  std::cout << myWatch.ElapsedTime() << "\n";
  myWatch.process_event( EvStartStop() );
  std::cout << myWatch.ElapsedTime() << "\n";
  myWatch.process_event( EvReset() );
  std::cout << myWatch.ElapsedTime() << "\n";
  return 0;
}
```

To actually see time being measured, you might want to single-step through the statements in `main()`. The StopWatch example extends this program to an interactive console application.

A digital camera

So far so good. However, the approach presented above has a few limitations:

- Bad scalability: As soon as the compiler reaches the point where `state_machine::initiate()` is called, a number of template instantiations take place, which can only succeed if the full declaration of each and every state of the machine is known. That is, the whole layout of a state machine must be implemented in one single translation unit (actions can be compiled separately, but this is of no importance here). For bigger (and more real-world) state machines, this leads to the following limitations:
  - At some point compilers reach their internal template instantiation limits and give up. This can happen even for moderately-sized machines. For example, in debug mode one popular compiler refused to compile earlier versions of the BitMachine example for anything above 3 bits. This means that the compiler reached its limits somewhere between 8 states, 24 transitions and 16 states, 64 transitions.
  - Multiple programmers can hardly work on the same state machine simultaneously because every layout change will inevitably lead to a recompilation of the whole state machine.
- Maximum one reaction per event: According to UML a state can have multiple reactions triggered by the same event. This makes sense when all reactions have mutually exclusive guards. The interface we used above only allows for at most one unguarded reaction for each event. Moreover, the UML concepts junction and choice point are not directly supported.
- There is no way to specify in-state reactions (aka inner transitions).

All these limitations can be overcome with custom reactions. **Warning: It is easy to abuse custom**

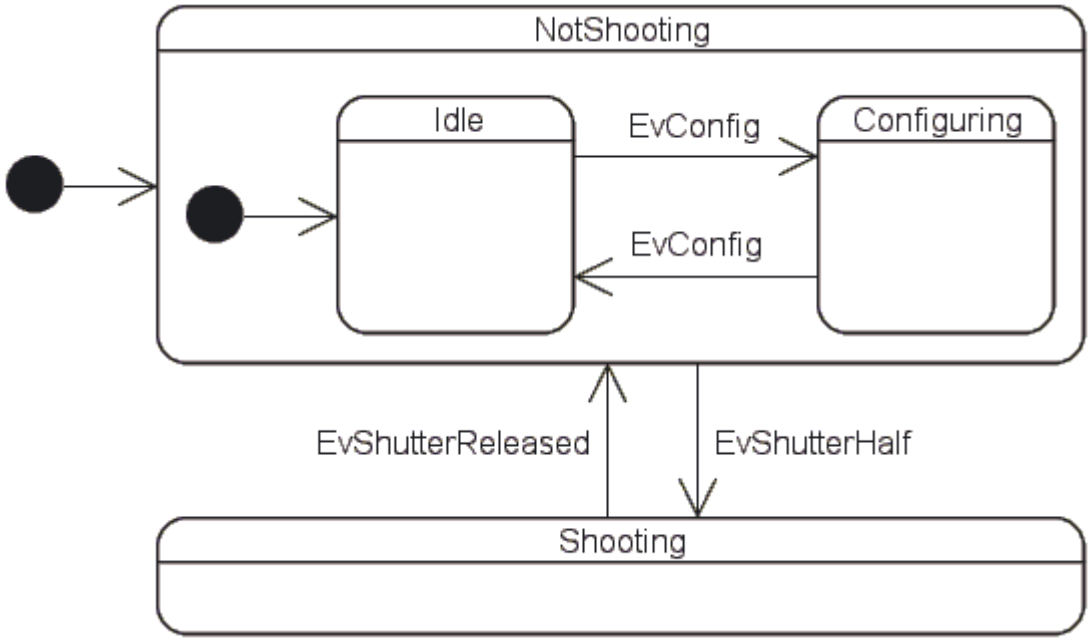
reactions up to the point of invoking undefined behavior. Please study the documentation before employing them!

Spreading a state machine over multiple translation units

Let's say your company would like to develop a digital camera. The camera has the following controls:

- Shutter button, which can be half-pressed and fully-pressed. The associated events are `EvShutterHalf`, `EvShutterFull` and `EvShutterReleased`
- Config button, represented by the `EvConfig` event
- A number of other buttons that are not of interest here

One use case for the camera says that the photographer can half-press the shutter **anywhere** in the configuration mode and the camera will immediately go into shooting mode. The following state chart is one way to achieve this behavior:



The Configuring and Shooting states will contain numerous nested states while the Idle state is relatively simple. It was therefore decided to build two teams. One will implement the shooting mode while the other will implement the configuration mode. The two teams have already agreed on the interface that the shooting team will use to retrieve the configuration settings. We would like to ensure that the two teams can work with the least possible interference. So, we put the two states in their own translation units so that machine layout changes within the Configuring state will never lead to a recompilation of the inner workings of the Shooting state and vice versa.

Unlike in the previous example, the excerpts presented here often outline different options to achieve the same effect. That's why the code is often not equal to the Camera example code. Comments mark the parts where this is the case.

Camera.hpp:

```
#ifndef CAMERA_HPP
#define CAMERA_HPP
```

```
#include <boost/fsm/event.hpp>
#include <boost/fsm/state_machine.hpp>
#include <boost/fsm/simple_state.hpp>
#include <boost/fsm/custom_reaction.hpp>

namespace fsm = boost::fsm;

struct EvShutterHalf : fsm::event< EvShutterHalf > {};
struct EvShutterFull : fsm::event< EvShutterFull > {};
struct EvShutterRelease : fsm::event< EvShutterRelease > {};
struct EvConfig : fsm::event< EvConfig > {};

struct NotShooting;
struct Camera : fsm::state_machine< Camera, NotShooting >
{
    bool IsMemoryAvailable() const { return true; }
    bool IsBatteryLow() const { return false; }
};

struct Idle;
struct NotShooting : fsm::simple_state< NotShooting, Camera,
  fsm::custom_reaction< EvShutterHalf >, Idle >
{
  // ...
  fsm::result react( const EvShutterHalf & );
};

struct Idle : fsm::simple_state< Idle, NotShooting,
  fsm::custom_reaction< EvConfig > >
{
  // ...
  fsm::result react( const EvConfig & );
};

#endif
```

Please note the bold parts in the code. With a custom reaction we only specify that we **might** do something with a particular event, but the actual reaction is defined in the `react` member function, which can be implemented in the .cpp file.

Camera.cpp:

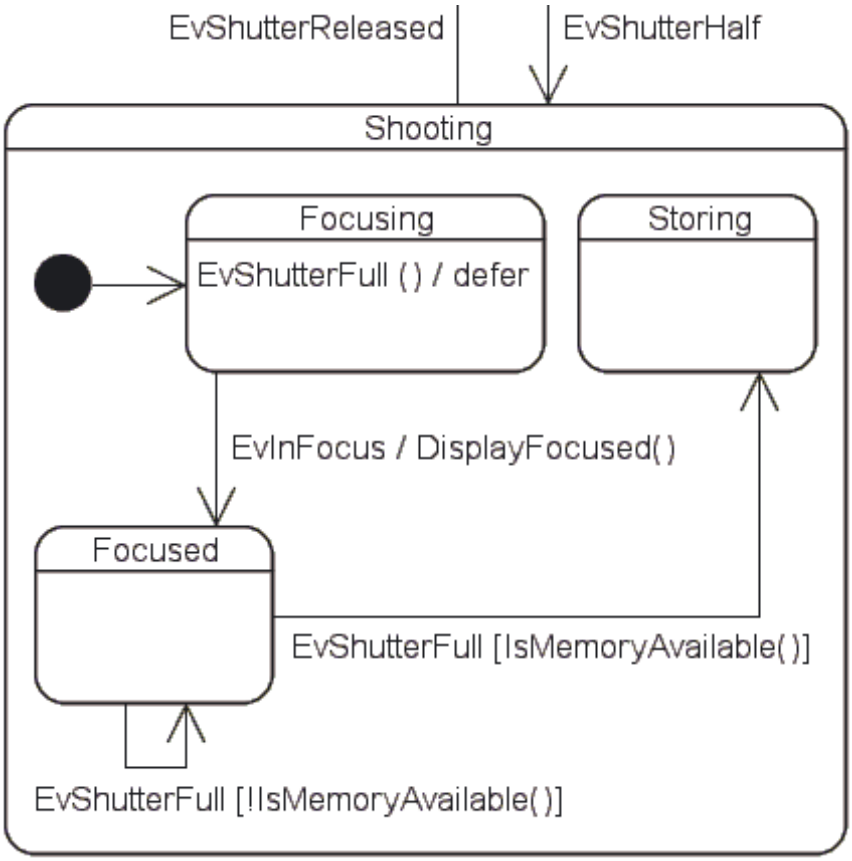
```
#include "Camera.hpp"
#include "Configuring.hpp"
#include "Shooting.hpp"

// ...

// not part of the Camera example
fsm::result NotShooting::react( const EvShutterHalf & )
{
  return transit< Shooting >();
}
```

```
fsm::result Idle::react( const EvConfig & )
{
  return transit< Configuring >();
}
```

Caution: Any call to the `simple_state::transit<>()` or `simple_state::terminate()` (see Reaction function reference) member functions will inevitably destruct the current state object (similar to `delete this;`)! That is, code executed after any of these calls may invoke undefined behavior! That's why these functions should only be called as part of a return statement.

Guards

The inner workings of the Shooting state could look as follows:



When the user half-presses the shutter, Shooting and its inner initial state Focusing are entered. In the Focusing entry action the camera instructs the focusing circuit to bring the subject into focus. The focusing circuit then moves the lenses accordingly and sends the EvInFocus event as soon as it is done. Of course, the user can fully-press the shutter while the lenses are still in motion. Without any precautions, the resulting EvShutterFull event would simply be lost because the Focusing state does not define a reaction for this event. As a result, the user would have to fully-press the shutter again after the camera has finished focusing. To prevent this, the EvShutterFull event is deferred inside the Focusing state. This means that all events of this type are stored in a separate queue, which is emptied into the main queue when the Focusing state is exited.

```
struct Focusing : fsm::state< Focusing, Shooting, mpl::list<
  fsm::custom_reaction< EvInFocus >,
```

```
    fsm::deferral< EvShutterFull > > >
{
  Focusing( my_context ctx );
  fsm::result react( const EvInFocus & );
};
```

Both transitions originating at the Focused state are triggered by the same event but they have mutually exclusive guards. Here is an appropriate custom reaction:

```
// not part of the Camera example
fsm::result Focused::react( const EvShutterFull & )
{
  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 >();
  }
}
```

Custom reactions can of course also be implemented directly in the state declaration, which is often preferable for easier browsing.

Next we will use a guard to prevent a transition and let outer states react to the event if the battery is low:

Camera.cpp:

```
// ...
fsm::result NotShooting::react( const EvShutterHalf & )
{
  if ( context< Camera >().IsBatteryLow() )
  {
    // We cannot react to the event ourselves, so we forward it
    // to our outer state (this is also the default if a state
    // defines no reaction for a given event).
    return forward_event();
  }
  else
  {
    return transit< Shooting >();
  }
}
// ...
```

In-state reactions (aka inner transitions)

The self-transition of the Focused state could also be implemented as an in-state reaction, which has the same effect as long as Focused does not have any entry or exit actions:

Shooting.cpp:

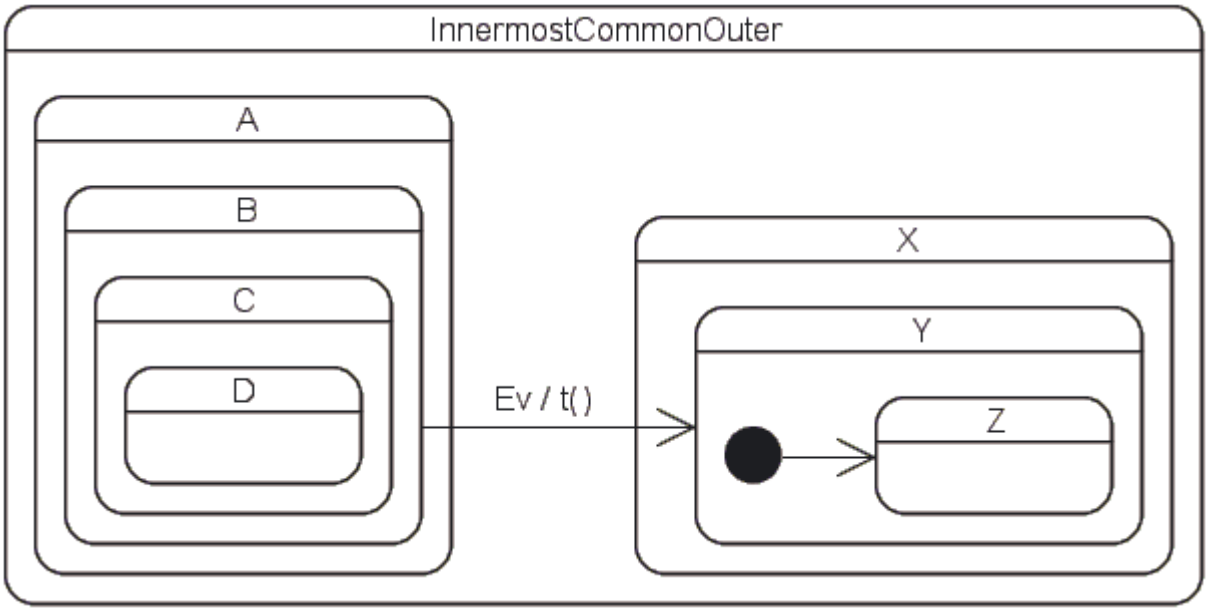
```
// ...
fsm::result Focused::react( const EvShutterFull & )
{
  if ( context< Camera >().IsMemoryAvailable() )
  {
    return transit< Storing >();
  }
  else
  {
    std::cout << "Cache memory full. Please wait...\n";
    // Indicate that the event can be discarded. So, the
    // dispatch algorithm will stop looking for a reaction.
    return discard_event();
  }
}
// ...
```

Transition actions

As an effect of every transition, actions are executed in the following order:

1. Starting from the innermost current state, all exit actions up to but excluding the innermost common outer state (aka LCA, least common ancestor).
2. The transition action (if present).
3. Starting from the innermost common outer state, all entry actions down to the target state followed by the entry actions of the initial states.

Example:



Here the order is as follows: ~D(), ~C(), ~B(), ~A(), t(), X(), Y(), Z(). The transition action t() is

therefore executed in the context of the InnermostCommonOuter state because the source state has already been left (destructed) and the target state has not yet been entered (constructed).

With boost::fsm, a transition action can be a member of **any** common outer context. That is, the transition between Focusing and Focused could be implemented as follows:

Shooting.hpp:

```
// ...
struct Focusing;
struct Shooting : fsm::simple_state< Shooting, Camera,
  fsm::transition< EvShutterRelease, NotShooting >, Focusing >
{
  // ...
  void DisplayFocused( const EvInFocus & );
};

// ...

// not part of the Camera example
struct Focusing : fsm::simple_state< Focusing, Shooting,
  fsm::transition< EvInFocus, Focused,
    Shooting, &Shooting::DisplayFocused > > {};
```

Or, the following is also possible (here the state machine itself serves as the outermost context):

```
// not part of the Camera example
struct Camera : fsm::state_machine< Camera, NotShooting >
{
  void DisplayFocused( const EvInFocus & );
};

// not part of the Camera example
struct Focusing : fsm::simple_state< Focusing, Shooting,
  fsm::transition< EvInFocus, Focused,
    Camera, &Camera::DisplayFocused > > {};
```

Naturally, transition actions can also be invoked from custom reactions:

Shooting.cpp:

```
// ...
fsm::result Focusing::react( const EvInFocus & evt )
{
  return transit< Focused >( &Shooting::DisplayFocused, evt );
}
```

Please note that we have to manually forward the event.

Advanced topics

Reaction function reference

The following functions can only be called from within `react` member functions, which must return by calling **exactly one** function (e.g. `return terminate();`):

- `simple_state::forward_event()`: The dispatch algorithm keeps searching for a reaction for the current event. The search always continues with the immediate outer state. If there is none it continues with the next orthogonal leaf state. This process is repeated until one of the visited states returns by calling any of the other 5 reaction functions. The event is silently discarded if no reaction can be found. Useful to implement guards. `forward_event()` is also the default for all states that do not define a reaction for the event.
- `simple_state::discard_event()`: The dispatch algorithm stops searching for a reaction and the current event is discarded. Useful to implement in-state reactions.
- `simple_state::defer_event()`: The current event is pushed into a separate queue and the dispatch algorithm stops searching for a reaction. When the state is exited later, the separate queue is emptied into the main queue, which is afterwards processed as usual. Please see Deferring events!
- `simple_state::transit< DestinationState >()`: Makes a transition to the specified destination state and discards the current event.
- `simple_state::transit< DestinationState >( void ( TransitionContext::* )( const Event & ), const Event & )`: Makes a transition to the specified destination state during which the passed transition action is called and discards the current event.
- `simple_state::terminate()`: Terminates the state and discards the current event.

Reaction reference

Reactions other than `custom_reaction` are nothing but syntactic sugar so that users don't have to write `react` member functions for common cases. Here's a list of the currently supplied reactions:

- `transition< Event, DestinationState >`: returns `simple_state::transit< DestinationState >();`
- `transition< Event, DestinationState, TransitionContext, void ( TransitionContext::*pTransitionAction )( const Event & ) >`: returns `simple_state::transit< DestinationState > ( pTransitionAction, evt );`
- `termination< Event >`: returns `simple_state::terminate();`
- `deferral< Event >`: returns `simple_state::defer_event();`. Please see Deferring events!
- `custom_reaction< Event >`: returns `react( evt );` (the user-supplied member function). The `react` member function must return by calling one of the reaction functions.

Should a user find herself implementing similar `react` member functions very often, she can easily define her own reaction and use it just like the ones that come with boost::fsm.

Specifying multiple reactions for a state

Often a state must define reactions for more than one event. In this case, an `mpl::list` must be used as outlined below:

```
// ...
```

```
#include <boost/mpl/list.hpp>

namespace mpl = boost::mpl;

// ...

struct Playing : fsm::simple_state< Playing, Mp3Player,
  mpl::list<
    fsm::custom_reaction< EvFastForward >,
    fsm::transition< EvStop, Stopped > > { /* ... */ };
```

Posting events

Non-trivial state machines often need to post internal events. Here's an example of how to do this:

```
Pumping::~Pumping()
{
  post_event( boost::intrusive_ptr< EvPumpingFinished >(
    new EvPumpingFinished() ) );
}
```

The event is pushed into the main queue, which is why it must be allocated with `new`. The events in the queue are processed as soon as the current reaction is completed. Events can be posted from inside `react` functions, entry-, exit- and transition actions. However, posting from inside entry actions is a bit more complicated (see e.g. `Focusing::Focusing` in `Shooting.cpp` in the Camera example):

```
struct Pumping : fsm::state< Pumping, Purifier >
{
  Pumping( my_context ctx ) : my_base( ctx )
  {
    post_event( boost::intrusive_ptr< EvPumpingStarted >(
      new EvPumpingStarted() ) );
  }
  // ...
};
```

Please note the bold parts. As soon as an entry action of a state needs to contact the "outside world" (here: the event queue in the state machine), the state must derive from `fsm::state` rather than from `fsm::simple_state` and must implement a forwarding constructor as outlined above (apart from the constructor, `fsm::state` offers the same interface as `fsm::simple_state`). Hence, this must be done whenever an entry action makes one or more calls to the following functions:

- `simple_state::context<>()`
- `simple_state::post_event()`
- `simple_state::state_cast<>()`
- `simple_state::state_downcast<>()`

In my experience, these functions are needed only rarely in entry actions so this workaround should not uglify user code too much.

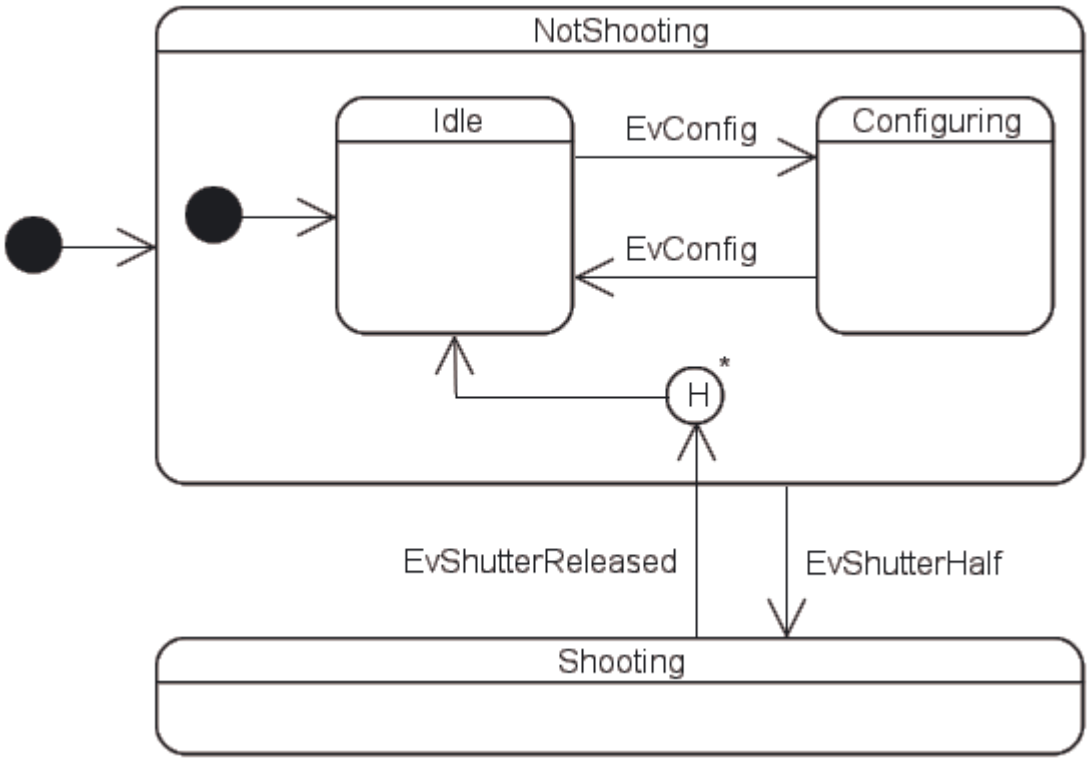
Deferring events

To avoid a number of overheads, event deferral has one limitation: Only events allocated with `new` **and** pointed to by a `boost::intrusive_ptr<>` can be deferred. Any attempt to defer a differently allocated event will result in a failing runtime assert. Example:

```
struct Event : fsm::event< Event > {};
struct Initial;
struct Machine : fsm::state_machine<
  Machine, Initial > {};
struct Initial : fsm::simple_state< Initial, Machine,
  fsm::deferral< Event > > {};

int main()
{
  Machine myMachine;
  myMachine.initiate();
  myMachine.process_event( Event() ); // error
  myMachine.process_event(
    *boost::shared_ptr< Event >( new Event() ) ); // error
  myMachine.process_event(
    *boost::intrusive_ptr< Event >( new Event() ) ); // fine
  return 0;
}
```

History

Photographers testing beta versions of our digital camera said that they really liked that half-pressing the shutter anytime (even while the camera is being configured) immediately readies the camera for picture-taking. However, most of them found it unintuitive that the camera always goes into the idle mode after releasing the shutter. They would rather see the camera go back into the state it had before half-pressing the shutter. This way they can easily test the influence of a configuration setting by modifying it, half- and then fully-pressing the shutter to take a picture. Finally, releasing the shutter will bring them back to the screen where they have modified the setting. To implement this behavior we'd change the state chart as follows:



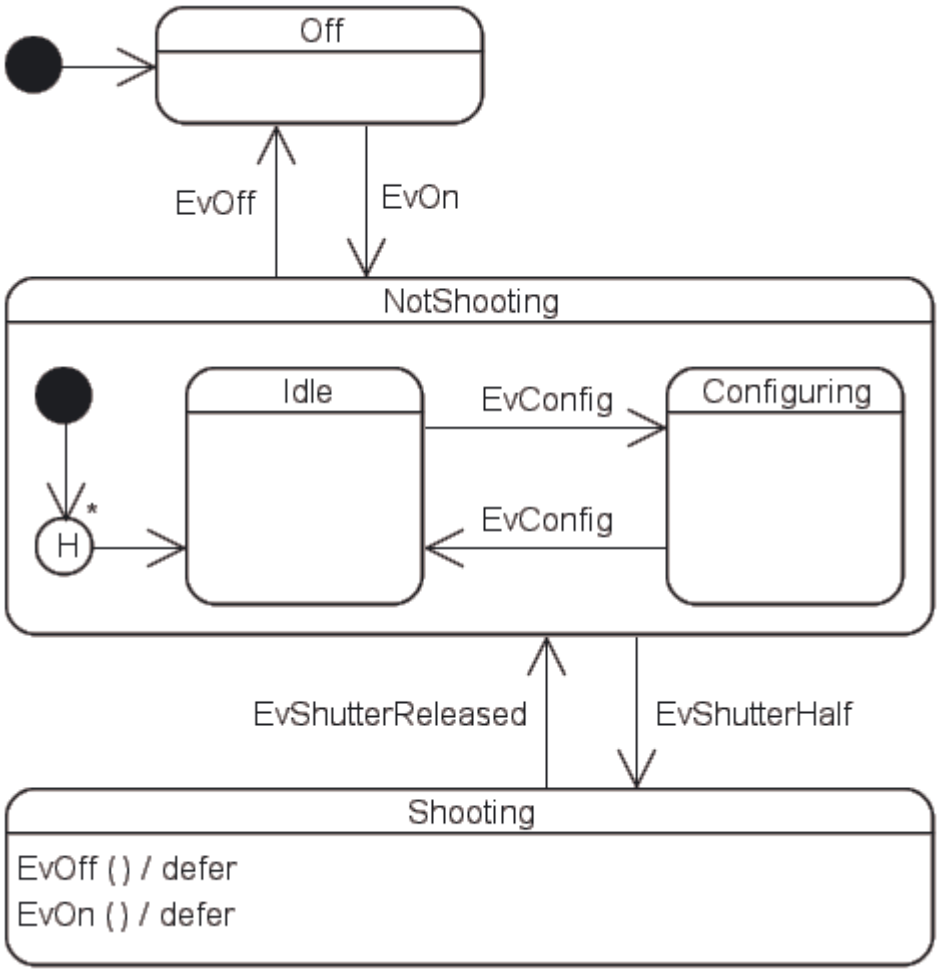
As mentioned earlier, the Configuring state contains a fairly complex and deeply nested inner machine. Naturally, we'd like to restore the previous state down to the innermost state(s) in Configuring, that's why we use a deep history pseudo state. The associated code looks as follows:

```
// not part of the Camera example
struct NotShooting : fsm::simple_state< NotShooting, Camera,
  /* ... */, Idle, fsm::has_deep_history > //
{
  // ...
};

// ...

struct Shooting : fsm::simple_state< Shooting, Camera,
  fsm::transition< EvShutterRelease,
    fsm::deep_history< Idle >, Focusing >
{
  // ...
};
```

History has two phases: Firstly, when the state containing the history pseudo state is exited, information about the previously active inner state hierarchy must be saved. Secondly, when a transition to the history pseudo state is made later, the saved state hierarchy information must be retrieved and the appropriate states entered. The former is expressed by passing either `fsm::has_shallow_history`, `fsm::has_deep_history` or `fsm::has_full_history` (which combines shallow and deep history) as the last parameter to the `simple_state` and `state` templates. The latter is expressed by specifying either `fsm::shallow_history<>` or `fsm::deep_history<>` as a transition destination or, as we'll see in an instant, as an inner initial state. Because it is possible that a state containing a history pseudo state has never been entered before a transition to history is made, both templates demand a parameter specifying the default state to enter in such situations.

The redundancy necessary for using history is checked for consistency at compile time. That is, the state machine wouldn't have compiled had we forgotten to pass `fsm::has_deep_history` to the base of `NotShooting`.

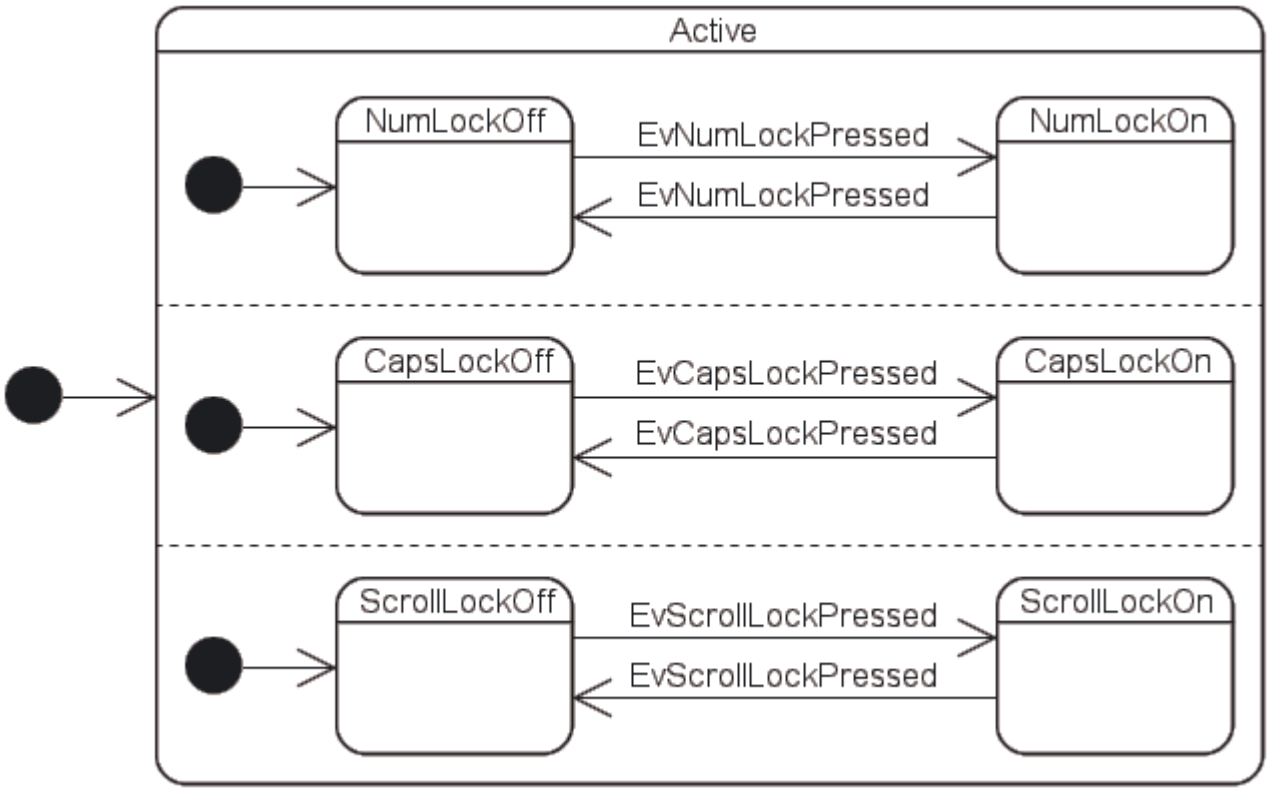
Another change request filed by a few beta testers says that they would like to see the camera go back into the state it had before turning it off when they turn it back on. Here's the implementation:




```
// ...

// not part of the Camera example
struct NotShooting : fsm::simple_state< NotShooting, Camera,
  /* ... */, mpl::list< fsm::deep_history< Idle > >,
  fsm::has_deep_history >
{
  // ...
};

// ...
```

Unfortunately, there is a small inconvenience due to some template-related implementation details. When the inner initial state is a class template instantiation we always have to put it into an `mpl::list<>`, although there is only one inner initial state. Moreover, the current deep history implementation has some limitations. Please have a look at the Limitations chapter in the Rationale.

Orthogonal states



To implement this state chart you simply specify more than one inner initial state (see the Keyboard example):

```
struct Active;
struct Keyboard : fsm::state_machine< Keyboard, Active > {};

struct NumLockOff;
struct CapsLockOff;
struct ScrollLockOff;
struct Active: fsm::simple_state<
  Active, Keyboard, fsm::no_reactions,
  mpl::list< NumLockOff, CapsLockOff, ScrollLockOff > > {};
```

Active's inner states must declare which orthogonal region they belong to:

```
struct EvNumLockPressed : fsm::event< EvNumLockPressed > {};
struct EvCapsLockPressed : fsm::event< EvCapsLockPressed > {};
struct EvScrollLockPressed :
  fsm::event< EvScrollLockPressed > {};

struct NumLockOn : fsm::simple_state<
  NumLockOn, Active::orthogonal< 0 >,
  fsm::transition< EvNumLockPressed, NumLockOff > > {};
struct NumLockOff : fsm::simple_state<
  NumLockOff, Active::orthogonal< 0 >,
  fsm::transition< EvNumLockPressed, NumLockOn > > {};

struct CapsLockOn : fsm::simple_state<
  CapsLockOn, Active::orthogonal< 1 >,
  fsm::transition< EvCapsLockPressed, CapsLockOff > > {};
```

```
struct CapsLockOff : fsm::simple_state<
  CapsLockOff, Active::orthogonal< 1 >,
  fsm::transition< EvCapsLockPressed, CapsLockOn > > {};

struct ScrollLockOn : fsm::simple_state<
  ScrollLockOn, Active::orthogonal< 2 >,
  fsm::transition< EvScrollLockPressed, ScrollLockOff > > {};
struct ScrollLockOff : fsm::simple_state<
  ScrollLockOff, Active::orthogonal< 2 >,
  fsm::transition< EvScrollLockPressed, ScrollLockOn > > {};
```

`orthogonal< 0 >` is the default, so `NumLockOn` and `NumLockOff` could just as well pass `Active` instead of `Active::orthogonal< 0 >` to specify their context. The numbers passed to the `orthogonal` member template must correspond to the list position in the outer state. Moreover, the orthogonal position of the source state of a transition must correspond to the orthogonal position of the target state. Any violations of these rules lead to compile time errors. Examples:

```
// Example 1: does not compile because Active specifies
// only 3 orthogonal regions
struct WhateverLockOn: fsm::simple_state<
  WhateverLockOn, Active::orthogonal< 3 > > {};

// Example 2: does not compile because Active specifies
// that NumLockOff is part of the "0th" orthogonal region
struct NumLockOff : fsm::simple_state<
  NumLockOff, Active::orthogonal< 1 > > {};

// Example 3: does not compile because a transition between
// different orthogonal regions is not permitted
struct CapsLockOn : fsm::simple_state<
  CapsLockOn, Active::orthogonal< 1 >,
  fsm::transition< EvCapsLockPressed, CapsLockOff > > {};
struct CapsLockOff : fsm::simple_state<
  CapsLockOff, Active::orthogonal< 2 >,
  fsm::transition< EvCapsLockPressed, CapsLockOn > > {};
```

State queries

Often reactions in a state machine depend on the current state in one or more orthogonal regions. This is because orthogonal regions are not completely orthogonal or a certain reaction in an outer state can only take place if the inner orthogonal regions are in particular states. For this purpose, the previously introduced `state_cast<>()` function is also available within states.

As a somewhat far-fetched example, let's assume that our keyboard above also accepts `EvRequestShutdown` events, the reception of which makes the keyboard terminate only if all lock keys are in the off state. We would then modify the `Active` state as follows:

```
struct EvRequestShutdown : fsm::event< EvRequestShutdown > {};

struct NumLockOff;
struct CapsLockOff;
struct ScrollLockOff;
```

```
struct Active: fsm::simple_state<
  Active, Keyboard, fsm::custom_reaction< EvRequestShutdown >,
  mpl::list< NumLockOff, CapsLockOff, ScrollLockOff > >
{
  fsm::result react( const EvRequestShutdown & )
  {
    if ( ( state_downcast< const NumLockOff * >() != 0 ) &&
         ( state_downcast< const CapsLockOff * >() != 0 ) &&
         ( state_downcast< const ScrollLockOff * >() != 0 ) )
    {
      return terminate();
    }
    else
    {
      return discard_event();
    }
  }
};
```

Just like `dynamic_cast`, passing a pointer type instead of reference type results in 0 pointers being returned when the cast fails. Note also the use of `state_downcast` instead of `state_cast`. Similar to the differences between `boost::polymorphic_downcast` and `dynamic_cast`, `state_downcast` is a much faster variant of `state_cast` and can only be used when the passed type is a most-derived type. `state_cast` should only be used if you want to query an additional base, as under Getting state information out of the machine.

Exception handling

Exceptions can be propagated from all user code except from state exit actions (mapped to destructors and destructors should virtually never throw in C++). Out of the box, `state_machine` does the following:

1. The exception is caught.
2. In the catch block, an `fsm::exception_thrown` event is allocated on the stack.
3. Also in the catch block, an **immediate** dispatch of the `fsm::exception_thrown` event is attempted. That is, possibly remaining events in the queue are dispatched only after the exception has been handled successfully.
4. If the exception was handled successfully, the state machine returns to the client normally. If the exception could not be handled successfully, the original exception is rethrown so that the client of the state machine can handle the exception.

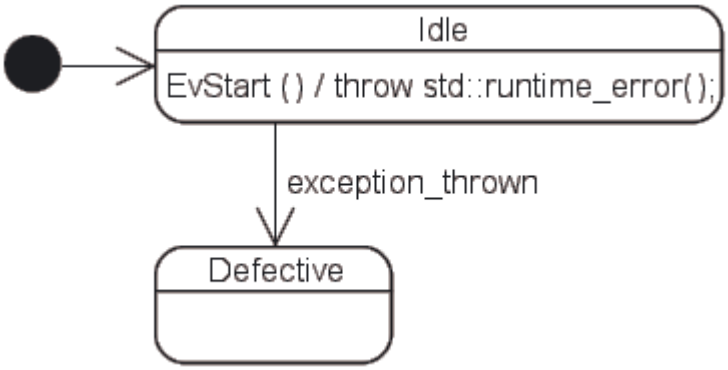
This behavior is implemented in the `exception_translator` class, which is the default for the `ExceptionTranslator` parameter of the `state_machine` class template. It was introduced because users would want to change this on some platforms to work around buggy exception handling implementations (see Discriminating exceptions).

boost::fsm can also be used in applications compiled with C++ exception handling turned off but doing so means losing **all** error handling support, making proper error handling much more cumbersome (see Error handling in the Rationale).

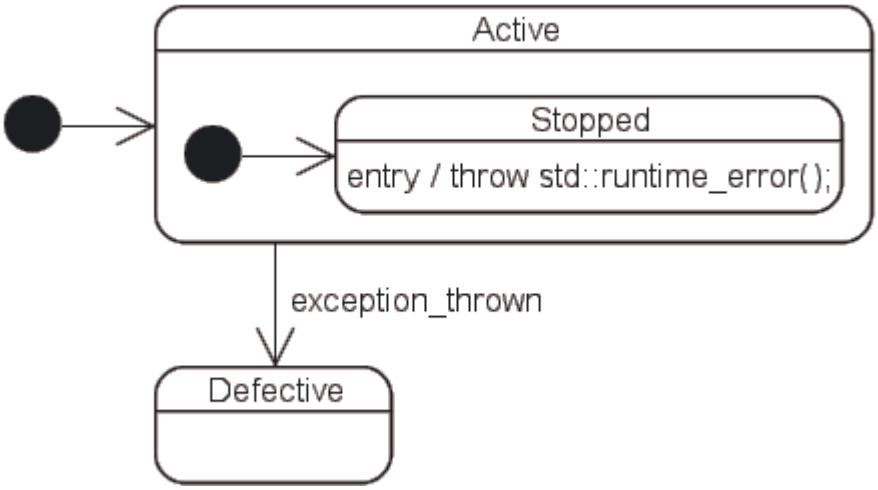
Which states can react to an `fsm::exception_thrown` event?

This depends on where the exception occurred. There are three scenarios:

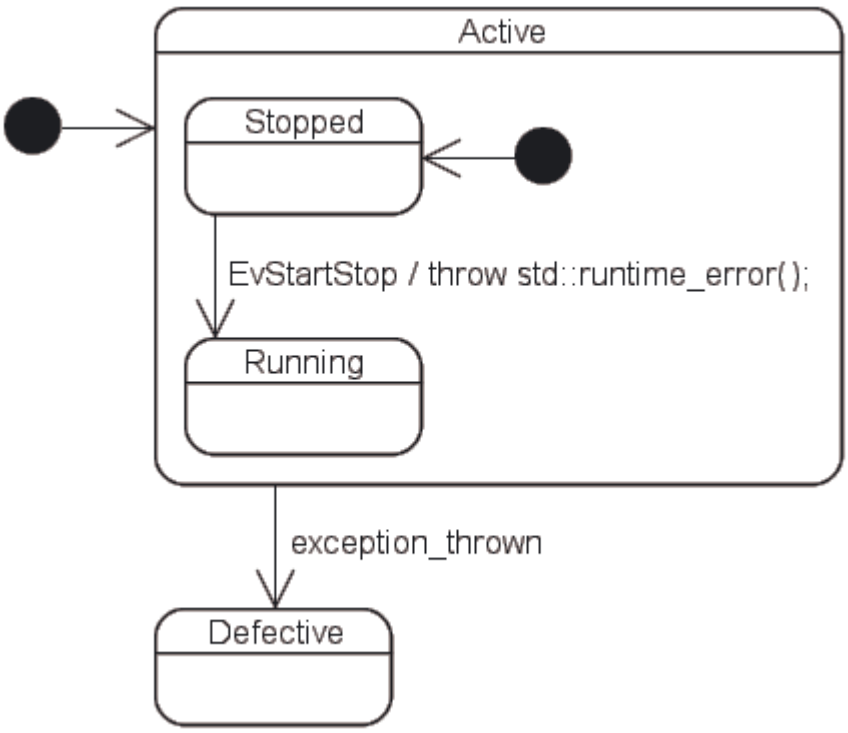
1. A `react` member function propagates an exception **before** calling any of the reaction functions. The state that caused the exception is first tried for a reaction, so the following machine will transit to Defective after receiving an EvStart event:



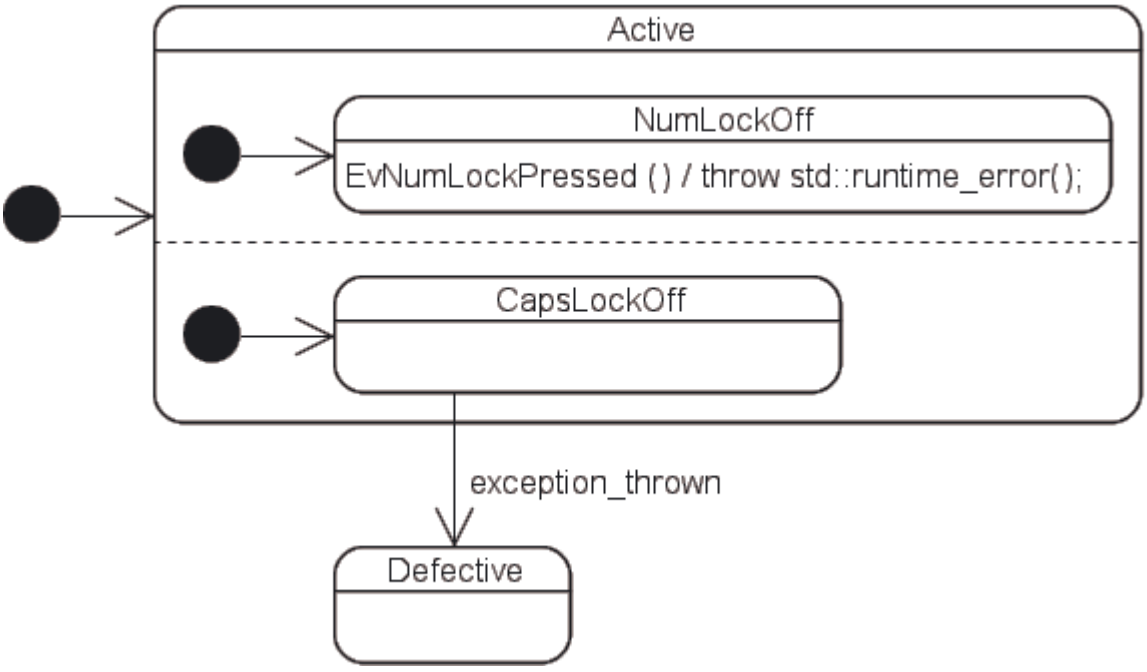
2. A state entry action (constructor) propagates an exception. The outer state of the state that caused the exception is first tried for a reaction, so the following machine will transit to Defective after trying to enter Stopped:



3. A transition action propagates an exception. The innermost common outer state of the source and the target state is first tried for a reaction, so the following machine will transit to Defective after receiving an EvStartStop event:



As with a normal event, the dispatch algorithm will move outward to find a reaction if the first tried state does not provide one (or if the reaction explicitly returned `forward_event();`). However, **in contrast to normal events, it will give up once it has unsuccessfully tried an outermost state**, so the following machine will **not** transit to Defective after receiving an EvNumLockPressed event:



Instead, the machine is terminated and the original exception rethrown.

Successful exception handling

An exception is considered handled successfully, if:

- an appropriate reaction for the `fsm::exception_thrown` event has been found, **and**

- the state machine is in a stable state after the reaction has completed.

The second condition is important for scenarios 2 and 3 in the last section. In these scenarios, the state machine is in the middle of a transition when the exception is handled. The machine would be left in an invalid state, should the reaction simply discard the event without doing anything else.

The out of the box behavior for unsuccessful exception handling is to rethrow the original exception. The state machine is terminated before the exception is propagated to the machine client.

Discriminating exceptions

Because the `fsm::exception_thrown` object is dispatched from within the catch block, we can rethrow and catch the exception in a custom reaction:

```
struct Defective : fsm::simple_state<
  Defective, Purifier > {};

// Pretend this is a state deeply nested in the Purifier
// state machine
struct Idle : fsm::simple_state< Idle, Purifier,
  mpl::list<
    fsm::custom_reaction< EvStart >,
    fsm::custom_reaction< fsm::exception_thrown > > >
{
  fsm::result react( const EvStart & )
  {
    throw std::runtime_error( "" );
  }

  fsm::result react( const fsm::exception_thrown & )
  {
    try
    {
      throw;
    }
    catch ( const std::runtime_error & )
    {
      // only std::runtime_errors will lead to a transition
      // to Defective ...
      return transit< Defective >();
    }
    catch ( ... )
    {
      // ... all other exceptions are forwarded to our outer
      // state(s). The state machine is terminated and the
      // exception rethrown if the outer state(s) can't
      // handle it either...
      return forward_event();
    }

    // Alternatively, if we want to terminate the machine
    // immediately, we can also either rethrow or throw
    // a different exception.
```

```
  }
};
```

Unfortunately, this idiom (using `throw;` inside a `try` block nested inside a `catch` block) does not work on at least one very popular compiler. If you have to use one of these platforms, you can pass a customized exception translator class to the `state_machine` class template. This will allow you to generate different events depending on the type of the exception.

Submachines & parameterized states

Submachines are to event-driven programming what functions are to procedural programming, reusable building blocks implementing often needed functionality. The associated UML notation is not entirely clear to me. It seems to be severely limited (e.g. the same submachine cannot appear in different orthogonal regions) and does not seem to account for obvious stuff like e.g. parameters.

boost::fsm is completely unaware of submachines but they can be implemented quite nicely with templates. Here, a submachine is used to improve the copy-paste implementation of the keyboard machine discussed under Orthogonal states:

```
enum LockType
{
  NUM_LOCK,
  CAPS_LOCK,
  SCROLL_LOCK
};

template< LockType lockType >
struct Off;
struct Active : fsm::simple_state<
  Active, Keyboard, fsm::no_reactions, mpl::list<
  Off< NUM_LOCK >, Off< CAPS_LOCK >, Off< SCROLL_LOCK > > > {};

template< LockType lockType >
struct EvPressed : fsm::event< EvPressed< lockType > > {};

template< LockType lockType >
struct On : fsm::simple_state<
  On< lockType >, Active::orthogonal< lockType >,
  fsm::transition< EvPressed< lockType >, Off< lockType > > > {}

template< LockType lockType >
struct Off : fsm::simple_state<
  Off< lockType >, Active::orthogonal< lockType >,
  fsm::transition< EvPressed< lockType >, On< lockType > > > {};
```

Asynchronous state machines

Why asynchronous state machines are necessary

As the name suggests, a synchronous state machine processes each event synchronously. This behavior is implemented by the `state_machine<>` class template, whose `process_event()` only returns after having executed all reactions (including the ones provoked by internal events that

actions might have posted). Moreover, this function is also strictly non-reentrant (just like all other member functions, so `state_machine<>` is not thread-safe). This makes it difficult for two `state_machine<>` subclasses to communicate via events in a bi-directional fashion correctly, **even in a single-threaded program**. For example, state machine `A` is in the middle of processing an external event. Inside an action, it decides to send a new event to state machine `B` (by calling `B::process_event` with an appropriate event). It then "waits" for B to send back an answer via a boost::function-like call-back, which references `A::process_event` and was passed as a data member of the event. However, while `A` is "waiting" for `B` to send back an event, `A::process_event` has not yet returned from processing the external event and as soon as `B` answers via the call-back, `A::process_event` is **unavoidably** reentered. This all really happens in a single thread, that's why "wait" is in quotes.

How it works

In contrast to `state_machine<>`, `asynchronous_state_machine<>` does not have a member function `process_event()`. Instead, there is only `queue_event()`, which returns immediately after pushing the event into a queue. A worker thread will later pop the event out of the queue to have it processed. For applications using the boost::thread library, the necessary locking, unlocking and waiting logic is readily available in class `worker<>`.

Applications will usually first create a `worker<>` object and then create one or more `asynchronous_state_machine<>` subclass objects, passing the worker object to the constructor(s). Finally, `worker<>::operator()()` is either called directly to let the machine (s) run in the current thread, or, a `boost::function` object referencing `operator()` is passed to a new `boost::thread`. I the following code, we are running one state machine in a new boost::thread and the other in the main thread (see the PingPong example for the full source code):

```
// ...

struct Waiting;
struct Player :
  fsm::asynchronous_state_machine< Player, Waiting >
{
  typedef fsm::asynchronous_state_machine< Player, Waiting >
    BaseType;

  Player( fsm::worker<> & myWorker ) :
    BaseType( myWorker ) // ...
  {
    // ...
  }

  // ...
};

// ...

int main()
{
  fsm::worker<> worker1;
  fsm::worker<> worker2;
```

```
  // each player runs in its own worker thread
  Player player1( worker1 );
  Player player2( worker2 );

  // ...

  // run first worker in a new thread
  boost::thread otherThread(
    boost::bind( &fsm::worker<>::operator(), &worker1 ) );

  worker2(); // run second worker in this thread
  otherThread.join();

  return 0;
}
```

We could just as well use two boost::threads:

```
int main()
{
  // ...

  boost::thread thread1(
    boost::bind( &fsm::worker<>::operator(), &worker1 ) );
  boost::thread thread2(
    boost::bind( &fsm::worker<>::operator(), &worker2 ) );

  // do something else ...

  thread1.join();
  thread2.join();

  return 0;
}
```

Or, run both machines in the same worker thread:

```
int main()
{
  fsm::worker<> worker1;

  Player player1( worker1 );
  Player player2( worker1 );

  // ...

  worker1();

  return 0;
}
```

`worker<>::operator()()` first initiates all machines and then waits for events. Whenever `queue_event` is called on one of the previously registered machines, the passed event is pushed

into the worker's queue and the worker thread is waked up to dispatch all queued events before waiting again. `worker<>::operator()()` returns as soon as all machines have terminated. `worker<>::operator()()` also throws any exceptions that machines fail to handle. In this case all machines are terminated before the exception is propagated.

Caution:

- **`asynchronous_state_machine<>` subclass objects must not be destructed before `worker::operator()()` returns. Moreover, the `worker<>` object may be destructed only after all of the registered state machines have been destructed. Violations of these rules will result in failing runtime asserts.**
- **The interface of `asynchronous_state_machine` consists of only the constructor and `queue_event()`. For technical reasons, other functions like `initiate()`, `process_event()`, etc. are nevertheless also publicly available, but it is not safe to call these functions from any other thread than the worker (over which most users have no control). `asynchronous_state_machine<>::queue_event()` is the only function than can safely be called simultaneously from multiple threads.**

Revised 12 October, 2003