The Boost Statechart Library

Tutorial

A Japanese translation of an earlier version of this tutorial can be found at http://prdownloads.sourceforge.jp/jyugem/7127/fsm-tutorial-jp.pdf. Kindly contributed by Mitsuo Fukasawa.

Contents

Introduction

The Boost Statechart library is a framework that allows you to quickly transform a UML statechart into executable C++ code, **without** needing to use a code generator. Thanks to support for almost all UML features the transformation is straight-forward and the resulting C++ code is a nearly redundancy-free textual description of the statechart.

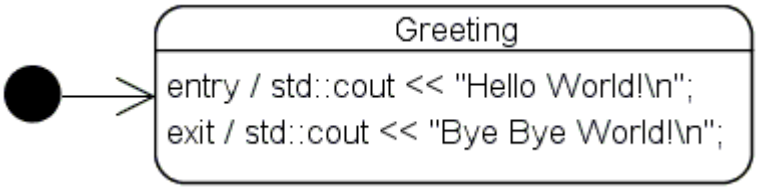
How to read this tutorial

This tutorial was designed to be read linearly. First time users should start reading right at the beginning and stop as soon as they know enough for the task at hand. Specifically:

- Small and simple machines with just a handful of states can be implemented reasonably well by using the features described under Basic topics: A stop watch
- For larger machines with up to roughly a dozen states the features described under Intermediate topics: A digital camera are often helpful
- Finally, users wanting to create even more complex machines and project architects evaluating Boost.Statechart should also read the Advanced topics section at the end. Moreover, reading the Limitations section in the Rationale is strongly suggested

Hello World!

We will use the simplest possible program to make our first steps. The statechart ...



... is implemented with the following code:

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

namespace sc = boost::statechart;

// We are declaring all types as structs only to avoid having to
// type public. If you don't mind doing so, you can just as well
// use class.

// We need to forward-declare the initial state because it can
// only be defined at a point where the state machine is
// defined.
struct Greeting;

// Boost.Statechart makes heavy use of the curiously recurring
// template pattern. The deriving class must always be passed as
// the first parameter to all base class templates.
//
// 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.
struct Machine : sc::state_machine< Machine, Greeting > {};

// For each state we need to define which state machine it
// belongs to and where it is located in the statechart. Both is
// specified with Context argument that is passed to
// simple_state<>. For a flat state machine as we have it here,
// the context is always the state machine. Consequently,
// Machine must be passed as the second template parameter to
// Greeting's base (the Context parameter is explained in more
// detail in the next example).
struct Greeting : sc::simple_state< Greeting, Machine >
```

```
{
  // Whenever the state machine enters a state, it creates an
  // object of the corresponding state class. The object is then
  // kept alive as long as the machine remains in the state.
  // Finally, the object is destroyed when the state machine
  // exits the state. Therefore, a state entry action can be
  // defined by adding a constructor and a state exit action can
  // be defined by adding a destructor.
  Greeting() { std::cout << "Hello World!\n"; } // entry
  ~Greeting() { std::cout << "Bye Bye World!\n"; } // exit
};

int main()
{
  Machine myMachine;
  // The machine is not yet running after construction. We start
  // it by calling initiate(). This triggers the construction of
  // the initial state Greeting
  myMachine.initiate();
  // When we leave main(), myMachine is destructed what leads to
  // the destruction of all currently active states.
  return 0;
}
```

This prints `Hello World!` and `Bye Bye World!` before exiting.

Basic topics: A stop watch

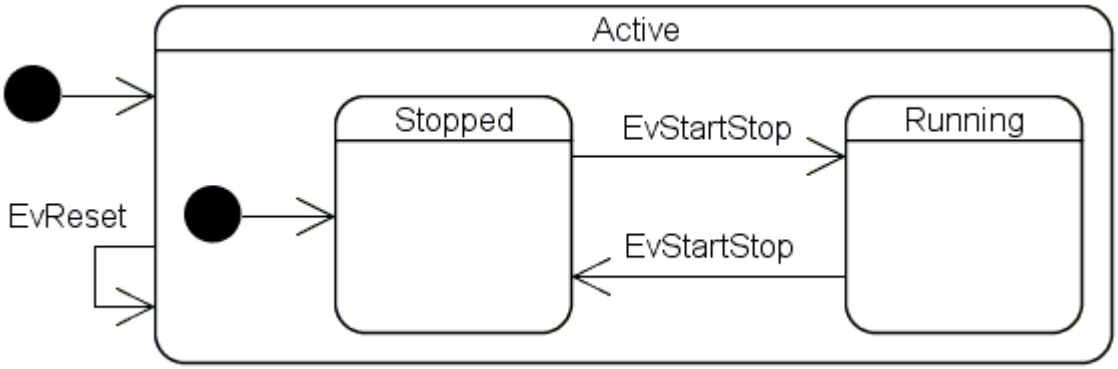
Next we will model a simple mechanical stop watch with a state machine. Such watches typically have two buttons:

- 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/statechart/event.hpp>
#include <boost/statechart/state_machine.hpp>
#include <boost/statechart/simple_state.hpp>

namespace sc = boost::statechart;

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

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

struct Stopped;

// The simple_state class template accepts up to four parameters
// - The third parameter specifies the inner initial state, if
//   there is one. Here, only Active has inner states, which is
//   why it needs to pass its inner initial state Stopped to its
//   base
// - The fourth parameter specifies whether and what kind of
//   history is kept

// Active is the outermost state and therefore needs to pass the
// state machine class it belongs to
struct Active : sc::simple_state<
  Active, StopWatch, Stopped > {};

// Stopped and Running both specify Active as their Context,
// which makes them nested inside Active
struct Running : sc::simple_state< Running, Active > {};
struct Stopped : sc::simple_state< Stopped, Active > {};

// 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 direct
// 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.

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

This compiles but doesn't do anything observable yet.

Adding reactions

For the moment we will use only one type of reaction: transitions. We **insert** the bold parts of the following code:

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

// ...

struct Stopped;
struct Active : sc::simple_state< Active, StopWatch, Stopped >
{
  typedef sc::transition< EvReset, Active > reactions;
};

struct Running : sc::simple_state< Running, Active >
{
  typedef sc::transition< EvStartStop, Stopped > reactions;
};

struct Stopped : sc::simple_state< Stopped, Active >
{
  typedef sc::transition< EvStartStop, Running > reactions;
};

// 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).

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

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 : sc::simple_state< Active, StopWatch, Stopped >
{
  public:
    typedef sc::transition< EvReset, Active > reactions;

    Active() : elapsedTime_( 0.0 ) {}
    double ElapsedTime() const { return elapsedTime_; }
    double & ElapsedTime() { return elapsedTime_; }
  private:
    double elapsedTime_;
};

struct Running : sc::simple_state< Running, Active >
{
  public:
    typedef sc::transition< EvStartStop, Stopped > reactions;

    Running() : startTime_( std::time( 0 ) ) {}
    ~Running()
    {
      // Similar to when a derived class object accesses its
      // base class portion, context<>() is used to gain
      // access to the direct or indirect context of a state.
      // This can either be a direct or indirect outer state
      // or the state machine itself
      // (e.g. here: context< StopWatch >()).
      context< Active >().ElapsedTime() +=
        std::difftime( std::time( 0 ), startTime_ );
    }
```

```
  private:
    std::time_t startTime_;
};

// ...
```

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

At this point, the advantages of state-local storage (which is still a relatively little-known feature) may not yet have become apparent. The FAQ item "What's so cool about state-local storage?" tries to explain them in more detail by comparing this StopWatch with one that does not make use of state-local storage.

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<>()` (StopWatch2.cpp shows the slightly more complex alternative). 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 double ElapsedTime() const = 0;
};

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

// ...

struct Running : IElapsedTime,
  sc::simple_state< Running, Active >
{
  public:
    typedef sc::transition< EvStartStop, Stopped > reactions;

    Running() : startTime_( std::time( 0 ) ) {}
```

```
    ~Running()
    {
      context< Active >().ElapsedTime() = ElapsedTime();
    }

    virtual double ElapsedTime() const
    {
      return context< Active >().ElapsedTime() +
        std::difftime( std::time( 0 ), startTime_ );
    }
  private:
    std::time_t startTime_;
};

struct Stopped : IElapsedTime,
  sc::simple_state< Stopped, Active >
{
  typedef sc::transition< EvStartStop, Running > reactions;

  virtual double 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.

Intermediate topics: 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

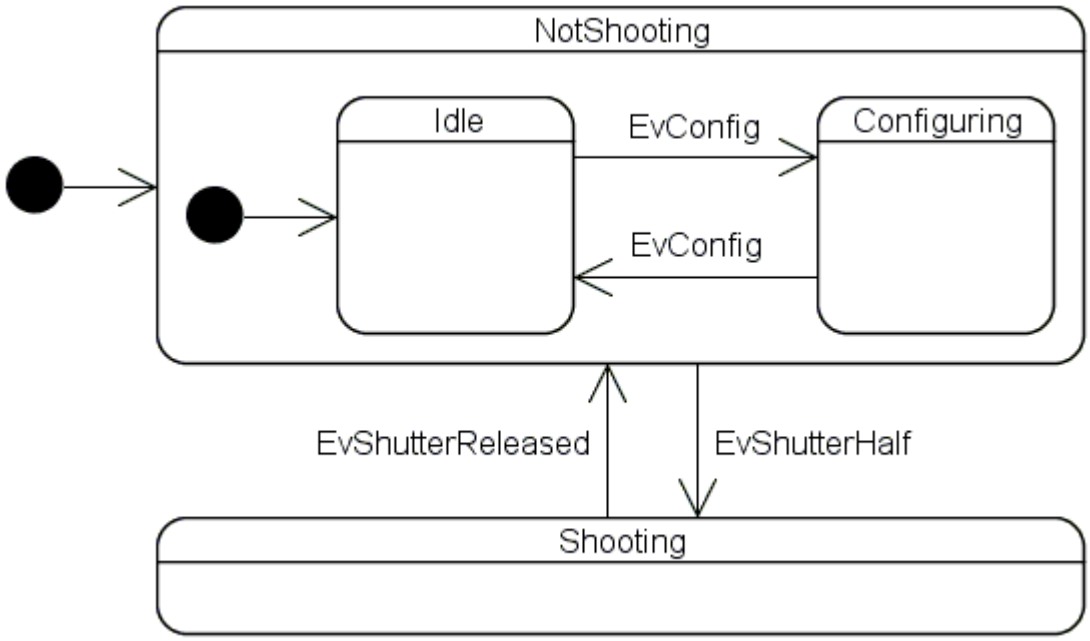
All these limitations can be overcome with custom reactions. **Warning: It is easy to abuse custom reactions up to the point of invoking undefined behavior. Please study the documentation before employing them!**

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 statechart 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_INCLUDED
#define CAMERA_HPP_INCLUDED

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

namespace sc = boost::statechart;

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

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

struct Idle;
struct NotShooting : sc::simple_state<
  NotShooting, Camera, Idle >
{
  // 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.
  typedef sc::custom_reaction< EvShutterHalf > reactions;

  // ...
  sc::result react( const EvShutterHalf & );
};

struct Idle : sc::simple_state< Idle, NotShooting >
{
  typedef sc::custom_reaction< EvConfig > reactions;

  // ...
  sc::result react( const EvConfig & );
```

```
};

#endif
```

Camera.cpp:

```
#include "Camera.hpp"

// The following includes are only made here but not in
// Camera.hpp
// The Shooting and Configuring states can themselves apply the
// same pattern to hide their inner implementation, which
// ensures that the two teams working on the Camera state
// machine will never need to disturb each other.
#include "Configuring.hpp"
#include "Shooting.hpp"

// ...

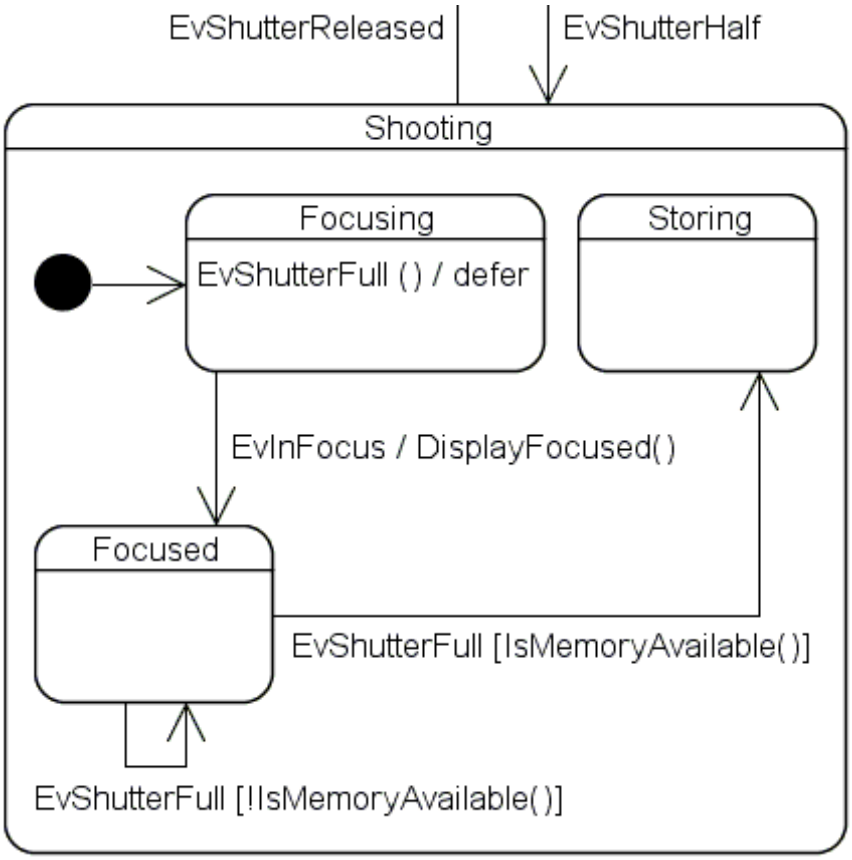
// not part of the Camera example
sc::result NotShooting::react( const EvShutterHalf & )
{
  return transit< Shooting >();
}

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

Caution: Any call to `simple_state<>::transit<>()` or `simple_state<>::terminate()` (see reference) will inevitably destruct the 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.

Deferring events

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 : sc::state< Focusing, Shooting >
{
  typedef mpl::list<
    sc::custom_reaction< EvInFocus >,
    sc::deferral< EvShutterFull >
  > reactions;

  Focusing( my_context ctx );
  sc::result react( const EvInFocus & );
};
```

Guards

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
sc::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:

```
// ...
sc::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

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:

```
// ...
sc::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
      // and the machine remains in the Focused state.
      return discard_event();
    }
  }
  // ...
```

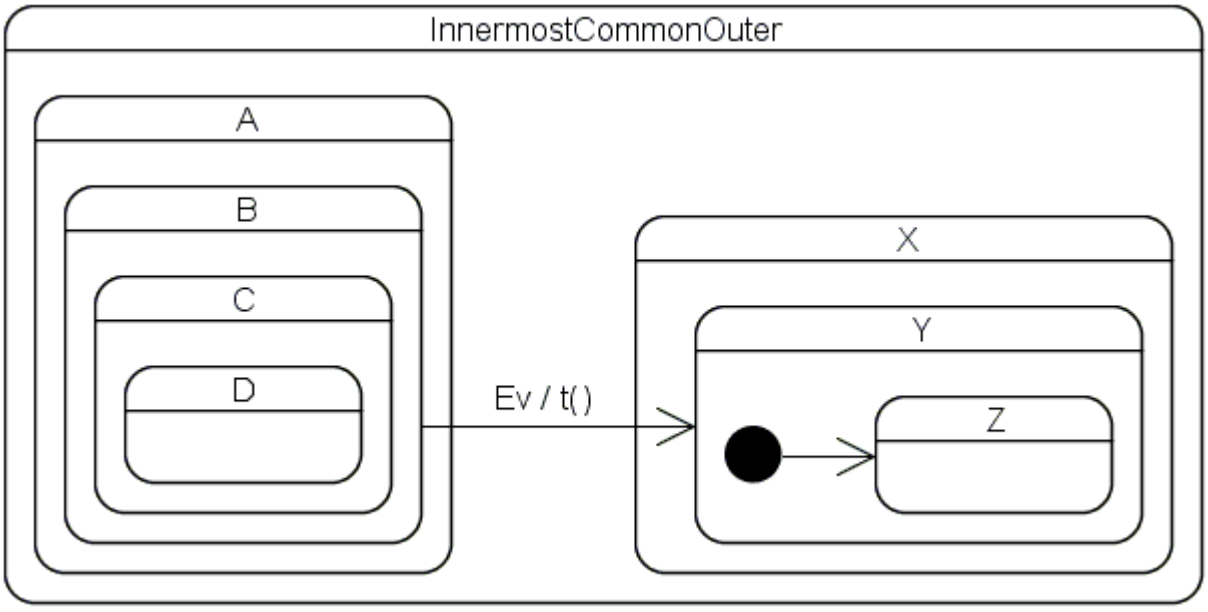
Because the in-state reaction is guarded, we need to employ a `custom_reaction<>` here. For unguarded in-state reactions `in_state_reaction<>` should be used for better code-readability.

Transition actions

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

1. Starting from the innermost active state, all exit actions up to but excluding the innermost common context
2. The transition action (if present)
3. Starting from the innermost common context, 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.Statechart, 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 : sc::simple_state< Shooting, Camera, Focusing >
{
  typedef sc::transition<
    EvShutterRelease, NotShooting > reactions;

  // ...
  void DisplayFocused( const EvInFocus & );
};

// ...

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

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

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

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

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

Shooting.cpp:

```
// ...
sc::result Focusing::react( const EvInFocus & evt )
{
  // We have to manually forward evt
  return transit< Focused >( &Shooting::DisplayFocused, evt );
}
```

Advanced topics

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 : sc::simple_state< Playing, Mp3Player >
{
  typdef mpl::list<
    sc::custom_reaction< EvFastForward >,
    sc::transition< EvStop, Stopped > > reactions;

  /* ... */
};
```

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( EvPumpingFinished() );
}
```

The event is pushed into the main queue. 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 : sc::state< Pumping, Purifier >
{
  Pumping( my_context ctx ) : my_base( ctx )
  {
    post_event( EvPumpingStarted() );
  }
  // ...
};
```

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 `state<>` rather than from `simple_state<>` and must implement a forwarding constructor as outlined above (apart from the constructor, `state<>` offers the same interface as `simple_state<>`). Hence, this must be done whenever an entry action makes one or more calls to the following functions:

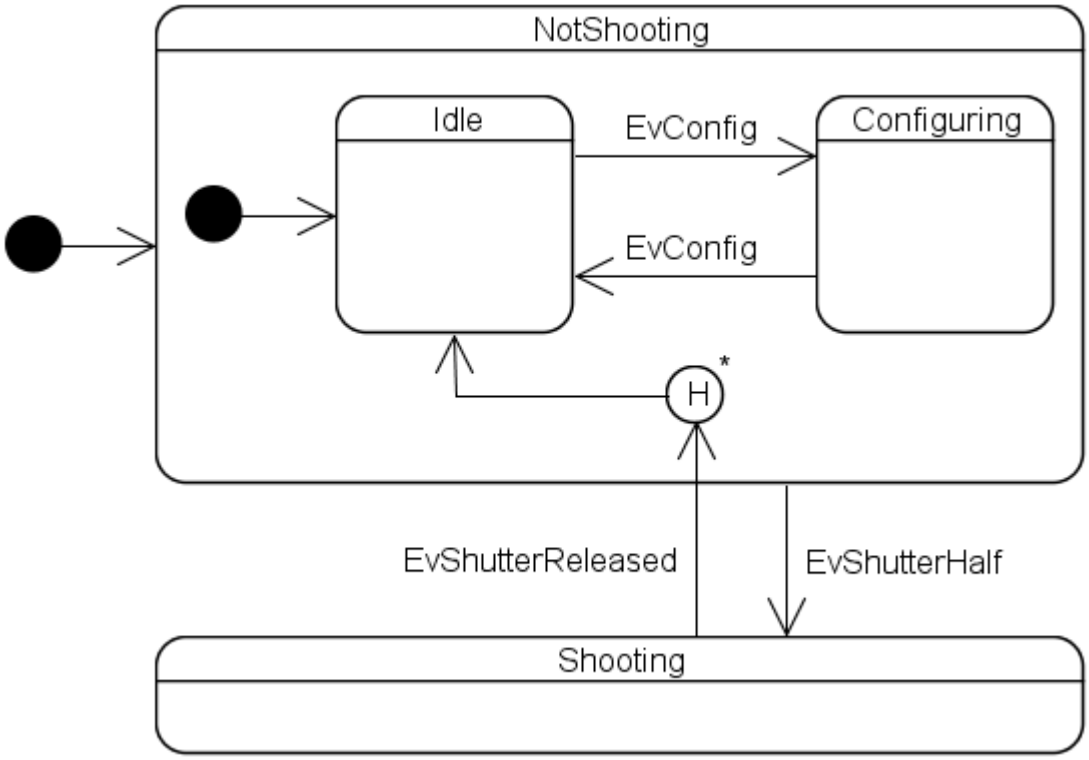
- `simple_state<>::post_event()`
- `simple_state<>::clear_shallow_history<>()`
- `simple_state<>::clear_deep_history<>()`
- `simple_state<>::outermost_context()`
- `simple_state<>::context<>()`
- `simple_state<>::state_cast<>()`

- `simple_state<>::state_downcast<>()`
- `simple_state<>::state_begin()`
- `simple_state<>::state_end()`

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

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 : sc::simple_state<
  NotShooting, Camera, Idle, sc::has_deep_history >
{
  // ...
};

// ...
```

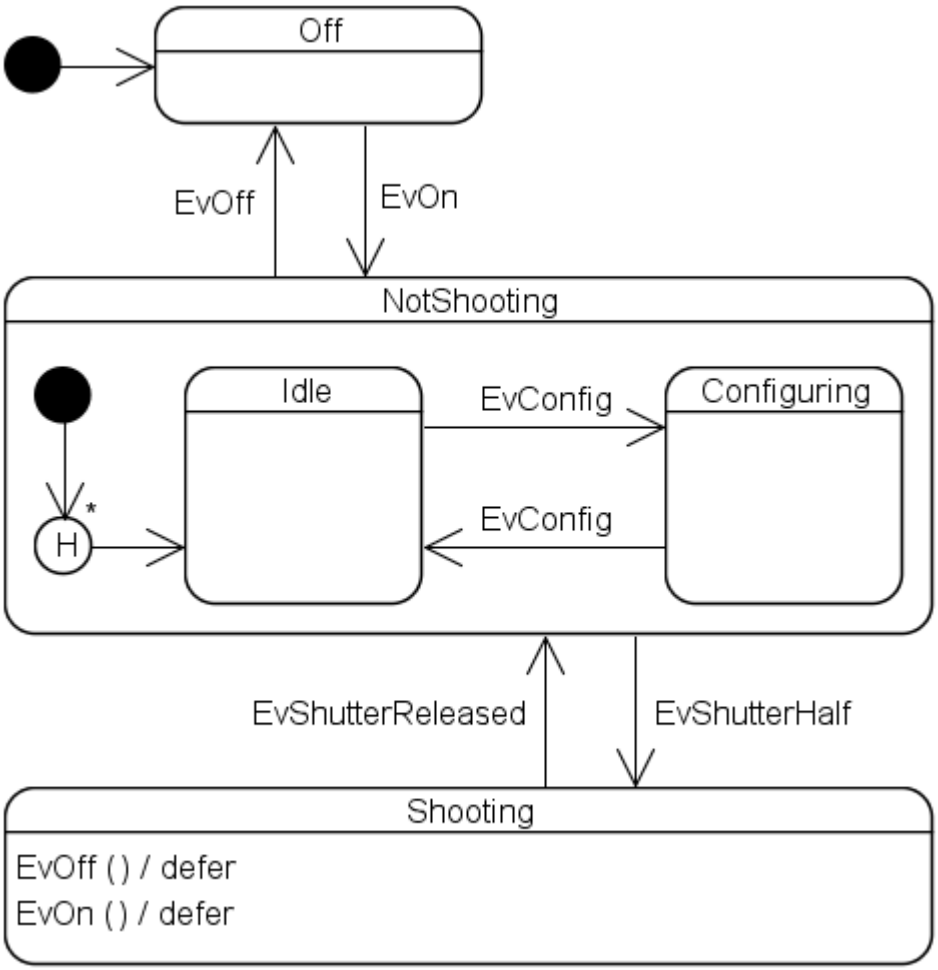
```
struct Shooting : sc::simple_state< Shooting, Camera, Focusing >
{
  typedef sc::transition<
    EvShutterRelease, sc::deep_history< Idle > > reactions;

  // ...
};
```

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 `has_shallow_history`, `has_deep_history` or `has_full_history` (which combines shallow and deep history) as the last parameter to the `simple_state` and `state` class templates. The latter is expressed by specifying either `shallow_history<>` or `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 class 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 `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:



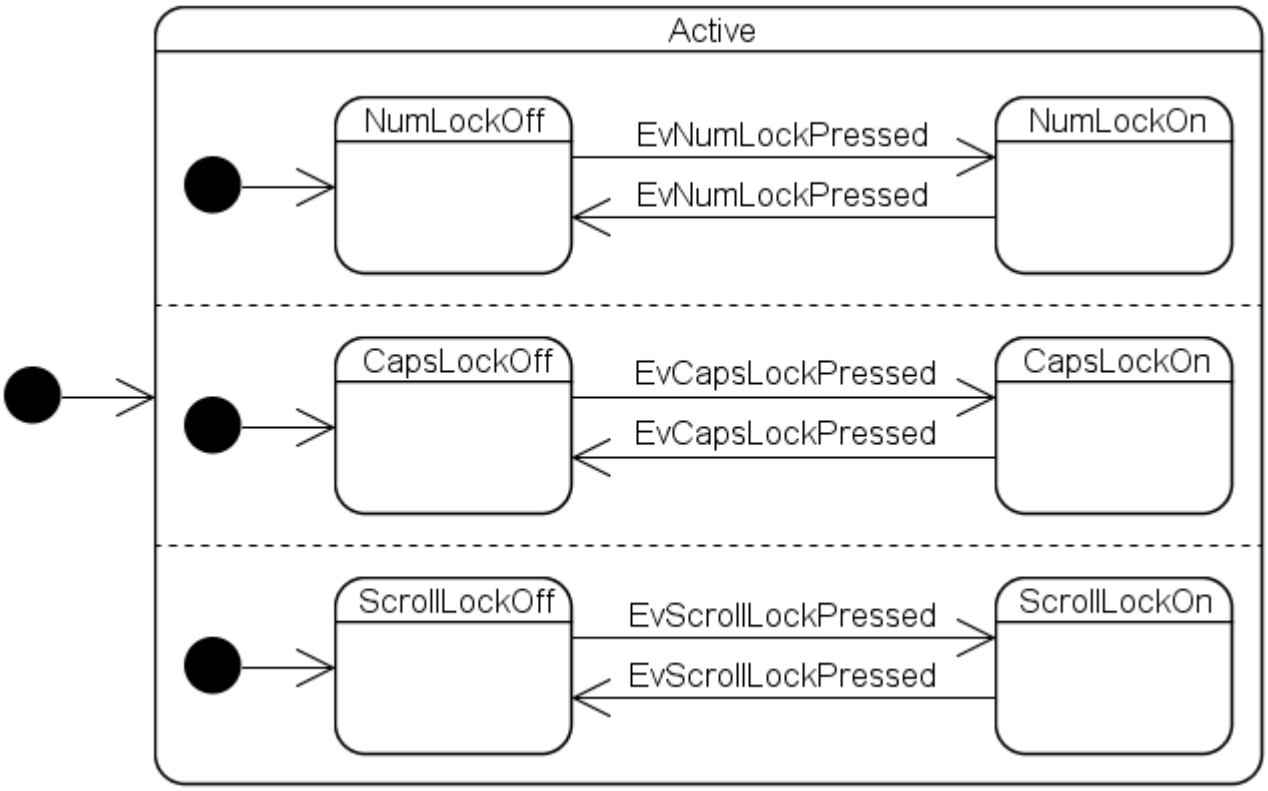
```
// ...

// not part of the Camera example
struct NotShooting : sc::simple_state< NotShooting, Camera,
  mpl::list< sc::deep_history< Idle > >,
  sc::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.

Orthogonal states



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

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

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

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

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

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

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

struct CapsLockOn : sc::simple_state<
  CapsLockOn, Active::orthogonal< 1 > >
{
  typedef sc::transition<
    EvCapsLockPressed, CapsLockOff > reactions;
};

struct CapsLockOff : sc::simple_state<
  CapsLockOff, Active::orthogonal< 1 > >
{
  typedef sc::transition<
    EvCapsLockPressed, CapsLockOn > reactions;
};

struct ScrollLockOn : sc::simple_state<
  ScrollLockOn, Active::orthogonal< 2 > >
{
  typedef sc::transition<
    EvScrollLockPressed, ScrollLockOff > reactions;
};

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

`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: sc::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 : sc::simple_state<
  NumLockOff, Active::orthogonal< 1 > > {};

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

struct CapsLockOff : sc::simple_state<
  CapsLockOff, Active::orthogonal< 2 > >
{
  typedef sc::transition<
    EvCapsLockPressed, CapsLockOn > reactions;
};
```

State queries

Often reactions in a state machine depend on the active 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 `state_cast<>` function introduced under Getting state information out of the machine is also available within states.

As a somewhat far-fetched example, let's assume that our keyboard 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 Keyboard state machine as follows:

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

struct NumLockOff;
struct CapsLockOff;
struct ScrollLockOff;
struct Active: sc::simple_state< Active, Keyboard,
  mpl::list< NumLockOff, CapsLockOff, ScrollLockOff > >
{
  typedef sc::custom_reaction< EvRequestShutdown > reactions;

  sc::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();
      }
    }
  };
```

Passing a pointer type instead of reference type results in 0 pointers being returned instead of `std::bad_cast` being thrown 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.

Custom state queries

It is often desirable to find out exactly which state(s) a machine currently resides in. To some extent this is already possible with `state_cast<>()` and `state_downcast<>()` but their utility is rather limited because both only return a yes/no answer to the question "Are you in state X?". It is possible to ask more sophisticated questions when you pass an additional base class rather than a state class to `state_cast<>()` but this involves more work (all states need to derive from and implement the additional base), is slow (under the hood `state_cast<>()` uses `dynamic_cast`), forces projects to compile with C++ RTTI turned on and has a negative impact on state entry/exit speed.

Especially for debugging it would be so much more useful being able to ask "In which state(s) are you?". For this purpose it is possible to iterate over all active **innermost** states with `state_machine<>::state_begin()` and `state_machine<>::state_end()`. Dereferencing the returned iterator returns a reference to `const state_machine<>::state_base_type`, the common base of all states. We can thus print the currently active state configuration as follows (see the Keyboard example for the complete code):

```
void DisplayStateConfiguration( const Keyboard & kbd )
{
  char region = 'a';

  for (
    Keyboard::state_iterator pLeafState = kbd.state_begin();
    pLeafState != kbd.state_end(); ++pLeafState )
  {
    std::cout << "Orthogonal region " << region << ": ";
    // The following use of typeid assumes that
    // BOOST_STATECHART_USE_NATIVE_RTTI is defined
    std::cout << typeid( *pLeafState ).name() << "\n";
    ++region;
  }
}
```

If necessary, the outer states can be accessed with `state_machine<>::state_base_type::outer_state_ptr()`, which returns a pointer to `const state_machine<>::state_base_type`. When called on an outermost state this function simply returns 0.

State type information

To cut down on executable size some applications must be compiled with C++ RTTI turned off. This would render the ability to iterate over all active states pretty much useless if it weren't for the following two functions:

- `static` *`unspecified_type`* `simple_state<>::static_type()`
- *`unspecified_type`*
  `state_machine<>::state_base_type::dynamic_type() const`

Both return a value that is comparable via `operator==()` and `std::less<>`. This alone would be enough to implement the `DisplayStateConfiguration` function above without the help of `typeid` but it is still somewhat cumbersome as a map must be used to associate the type information values with the state names.

Custom state type information

That's why the following functions are also provided (only available when BOOST_STATECHART_USE_NATIVE_RTTI is **not** defined):

- ```
  template< class T >
  static void simple_state<>::custom_static_type_ptr( const T
  * );
  ```
- ```
  template< class T >
  static const T * simple_state<>::custom_static_type_ptr();
  ```
- ```
  template< class T >
  const T * state_machine<>::
    state_base_type::custom_dynamic_type_ptr() const;
  ```

These allow us to directly associate arbitrary state type information with each state ...

```
// ...

int main()
{
  NumLockOn::custom_static_type_ptr( "NumLockOn" );
  NumLockOff::custom_static_type_ptr( "NumLockOff" );
  CapsLockOn::custom_static_type_ptr( "CapsLockOn" );
  CapsLockOff::custom_static_type_ptr( "CapsLockOff" );
  ScrollLockOn::custom_static_type_ptr( "ScrollLockOn" );
  ScrollLockOff::custom_static_type_ptr( "ScrollLockOff" );

  // ...
}
```

... and rewrite the display function as follows:
```

```
void DisplayStateConfiguration( const Keyboard & kbd )
{
  char region = 'a';

  for (
    Keyboard::state_iterator pLeafState = kbd.state_begin();
    pLeafState != kbd.state_end(); ++pLeafState )
  {
    std::cout << "Orthogonal region " << region << ": ";
    std::cout <<
      pLeafState->custom_dynamic_type_ptr< char >() << "\n";
    ++region;
  }
}
```

Exception handling

Exceptions can be propagated from all user code except from state destructors. Out of the box, the state machine framework is configured for simple exception handling and does not catch any of these exceptions, so they are immediately propagated to the state machine client. A scope guard inside the `state_machine<>` ensures that all state objects are destructed before the exception is caught by the client. The scope guard does not attempt to call any `exit` functions (see Two stage exit below) that states might define as these could themselves throw other exceptions which would mask the original exception. Consequently, if a state machine should do something more sensible when exceptions are thrown, it has to catch them before they are propagated into the Boost.Statechart framework. This exception handling scheme is often appropriate but it can lead to considerable code duplication in state machines where many actions can trigger exceptions that need to be handled inside the state machine (see Error handling in the Rationale).
That's why exception handling can be customized through the `ExceptionTranslator` parameter of the `state_machine` class template. Since the out-of-the box behavior is to **not** translate any exceptions, the default argument for this parameter is `null_exception_translator`. A `state_machine<>` subtype can be configured for advanced exception handling by specifying the library-supplied `exception_translator<>` instead. This way, the following happens when an exception is propagated from user code:

1. The exception is caught inside the framework
2. In the catch block, an `exception_thrown` event is allocated on the stack
3. Also in the catch block, an **immediate** dispatch of the `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

On platforms with buggy exception handling implementations users would probably want to implement their own model of the ExceptionTranslator concept (see also Discriminating exceptions).

Successful exception handling

An exception is considered handled successfully, if:

- an appropriate reaction for the `exception_thrown` event has been found, **and**
- the state machine is in a stable state after the reaction has completed.

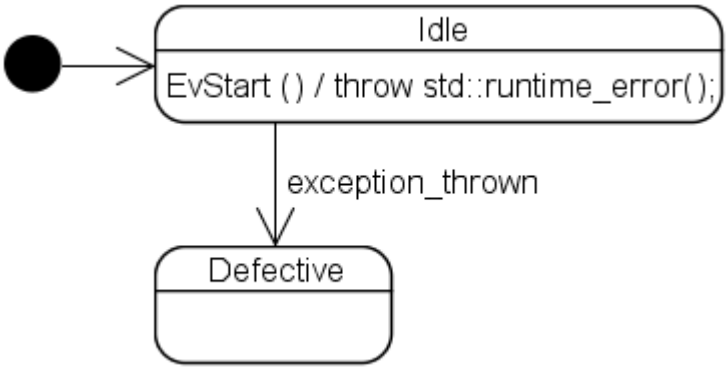
The second condition is important for scenarios 2 and 3 in the next section. In these scenarios, the state machine is in the middle of a transition when the exception is handled. The machine would be left in an invalid state, should the reaction simply discard the event without doing anything else. `exception_translator<>` simply rethrows the original exception if the exception handling was unsuccessful. Just as with simple exception handling, in this case a scope guard inside the `state_machine<>` ensures that all state objects are destructed before the exception is caught by the client.

Which states can react to an `exception_thrown` event?

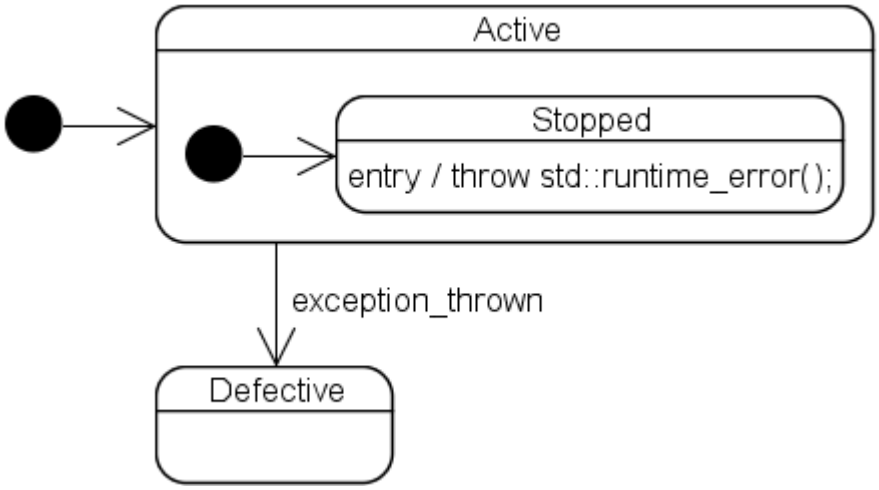
Short answer: If the state machine is stable when the exception is thrown, the state that caused the exception is first tried for a reaction. Otherwise the outermost unstable state is first tried for a reaction.

Longer answer: There are three scenarios:

1. A `react` member function propagates an exception **before** calling any of the reaction functions or the action executed during an in-state reaction propagates an exception. 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:

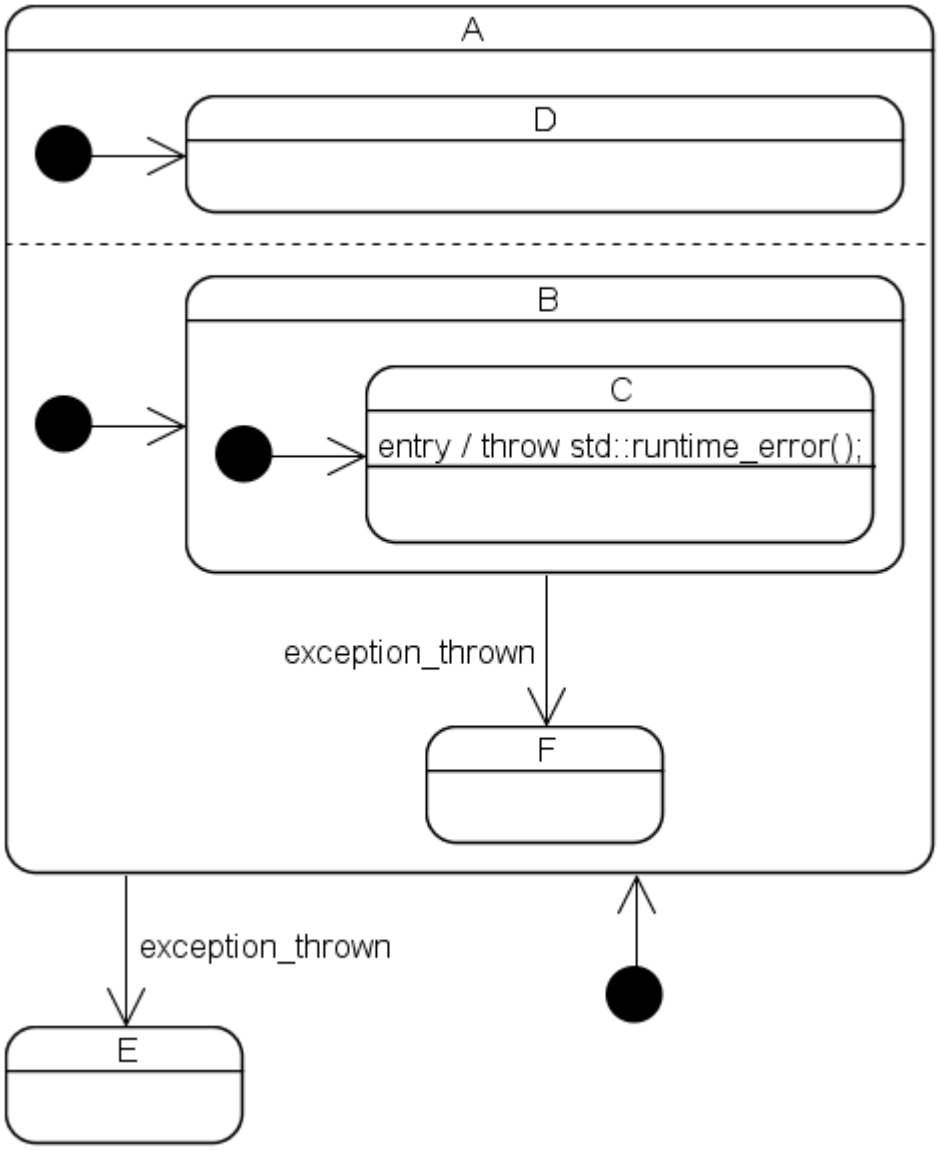


2. A state entry action (constructor) propagates an exception:
   - If there are no orthogonal regions, the direct outer state of the state that caused the exception is first tried for a reaction, so the following machine will transit to Defective after trying to enter Stopped:



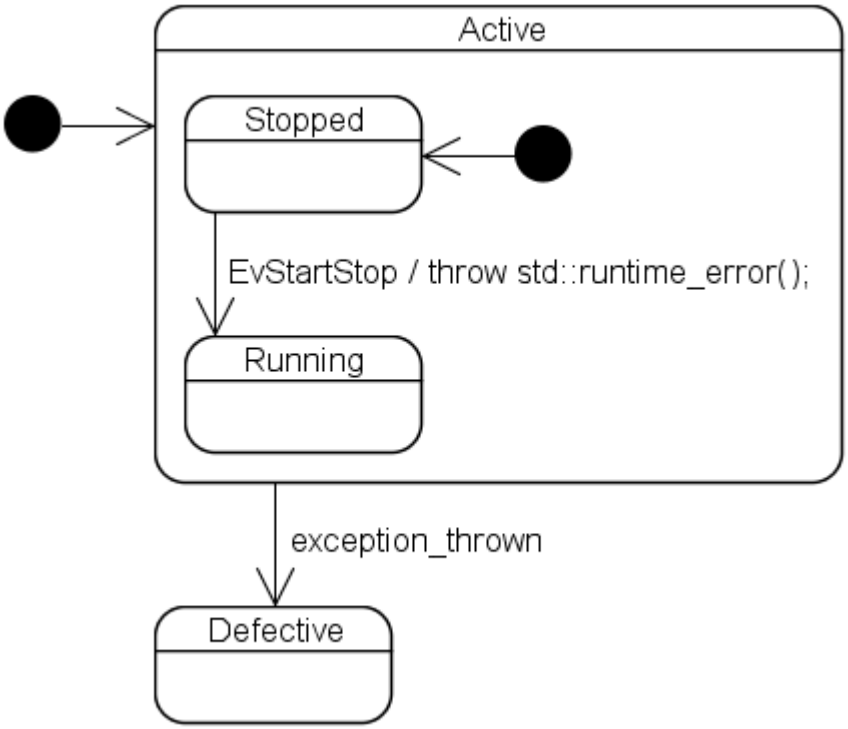
   - If there are orthogonal regions, the outermost unstable state is first tried for a reaction. The outermost unstable state is found by first selecting the direct outer state of the state that caused the exception and then moving outward until a state is found that is unstable

but has no direct or indirect outer states that are unstable. This more complex rule is necessary because only reactions associated with the outermost unstable state (or any of its direct or indirect outer states) are able to bring the machine back into a stable state. Consider the following statechart:

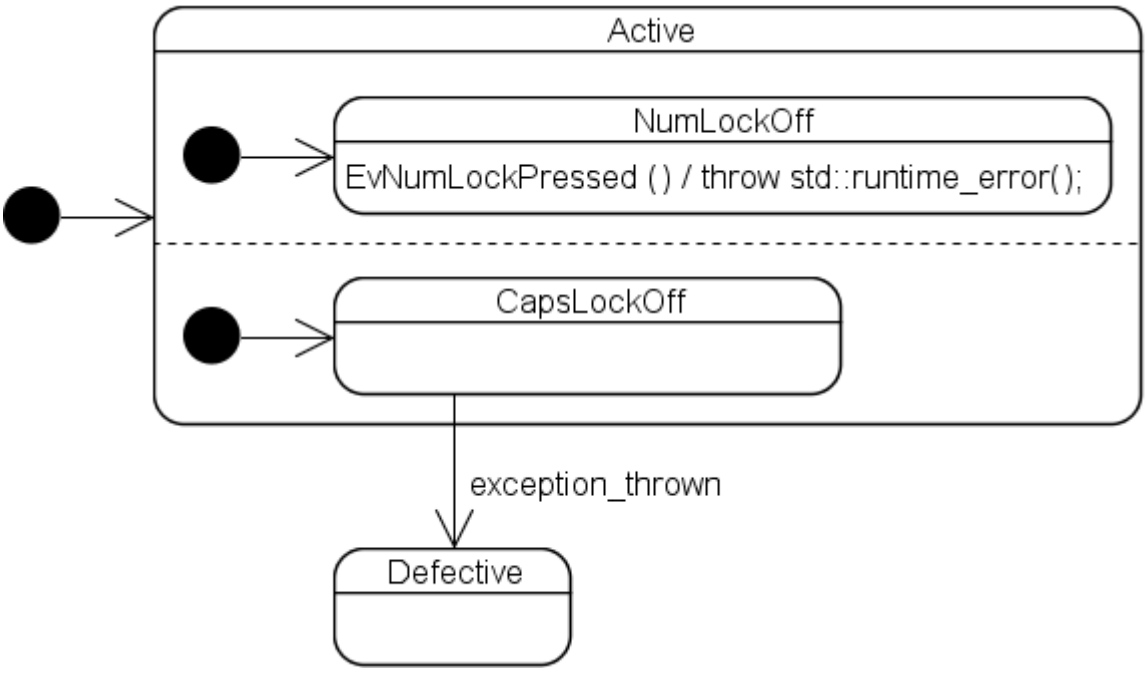


Whether this state machine will ultimately transition to E or F after initiation depends on which of the two orthogonal regions is initiated first. If the upper orthogonal region is initiated first, the entry sequence is as follows: A, D, B, (exception is thrown). Both D and B were successfully entered, so B is the outermost unstable state when the exception is thrown and the machine will therefore transition to F. However, if the lower orthogonal region is initiated first, the sequence is as follows: A, B, (exception is thrown). D was never entered so A is the outermost unstable state when the exception is thrown and the machine will therefore transition to E.
In practice these differences rarely matter as top-level error recovery is adequate for most state machines. However, since the sequence of initiation is clearly defined (orthogonal region 0 is always initiated first, then region 1 and so forth), users **can** accurately control when and where they want to handle exceptions

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.

Discriminating exceptions

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

```
struct Defective : sc::simple_state<
```

```
    Defective, Purifier > {};

  // Pretend this is a state deeply nested in the Purifier
  // state machine
  struct Idle : sc::simple_state< Idle, Purifier >
  {
    typedef mpl::list<
      sc::custom_reaction< EvStart >,
      sc::custom_reaction< sc::exception_thrown >
    > reactions;

    sc::result react( const EvStart & )
    {
      throw std::runtime_error( "" );
    }

    sc::result react( const sc::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.

Two stage exit

If a `simple_state<>` or `state<>` subtype declares a public member function with the signature `void exit()` then this function is called just before the state object is destructed. As explained under Error handling in the Rationale, this is useful for two things that would otherwise be difficult

or cumbersome to achieve with destructors only:

1. To signal a failure in an exit action
2. To execute certain exit actions **only** during a transition or a termination but not when the state machine object is destructed

A few points to consider before employing `exit()`:

- There is no guarantee that `exit()` will be called:
  - If the client destructs the state machine object without calling `terminate()` beforehand then the currently active states are destructed without calling `exit()`. This is necessary because an exception that is possibly thrown from `exit()` could not be propagated on to the state machine client
  - `exit()` is not called when a previously executed action propagated an exception and that exception has not (yet) been handled successfully. This is because a new exception that could possibly be thrown from `exit()` would mask the original exception
- A state is considered exited, even if its `exit` function propagated an exception. That is, the state object is inevitably destructed right after calling `exit()`, regardless of whether `exit()` propagated an exception or not. A state machine configured for advanced exception handling is therefore always unstable while handling an exception propagated from an `exit` function
- In a state machine configured for advanced exception handling the processing rules for an exception event resulting from an exception propagated from `exit()` are analogous to the ones defined for exceptions propagated from state constructors. That is, the outermost unstable state is first tried for a reaction and the dispatcher then moves outward until an appropriate reaction is found

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.Statechart 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 : sc::simple_state<
  Active, Keyboard, mpl::list<
  Off< NUM_LOCK >, Off< CAPS_LOCK >, Off< SCROLL_LOCK > > > {};

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

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

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

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` function only returns after having executed all reactions (including the ones provoked by internal events that actions might have posted). This function is strictly non-reentrant (just like all other member functions, so `state_machine<>` is not thread-safe). This makes it difficult for two `state_machine<>` subtype objects 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()`). 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

The `asynchronous_state_machine` class template has none of the member functions the `state_machine` class template has. Moreover, `asynchronous_state_machine<>` subtype objects cannot even be created or destroyed directly. Instead, all these operations must be performed through the `Scheduler` object each asynchronous state machine is associated with. All these `Scheduler` member functions only push an appropriate item into the schedulers' queue and then return immediately. A dedicated thread will later pop the items out of the queue to have them processed.

Applications will usually first create a `fifo_scheduler<>` object and then call `fifo_scheduler<>::create_processor<>()` and `fifo_scheduler<>::initiate_processor()` to schedule the creation and initiation of one or more `asynchronous_state_machine<>` subtype objects. Finally, `fifo_scheduler<>::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`. Alternatively, the latter could also be done right after constructing the

`fifo_scheduler<>` object. In 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 :
  sc::asynchronous_state_machine< Player, Waiting >
{
  // ...
};

// ...

int main()
{
  // Create two schedulers that will wait for new events
  // when their event queue runs empty
  sc::fifo_scheduler<> scheduler1( true );
  sc::fifo_scheduler<> scheduler2( true );

  // Each player is serviced by its own scheduler
  sc::fifo_scheduler<>::processor_handle player1 =
    scheduler1.create_processor< Player >( /* ... */ );
  scheduler1.initiate_processor( player1 );
  sc::fifo_scheduler<>::processor_handle player2 =
    scheduler2.create_processor< Player >( /* ... */ );
  scheduler2.initiate_processor( player2 );

  // the initial event that will start the game
  boost::intrusive_ptr< BallReturned > pInitialBall =
    new BallReturned();

  // ...

  scheduler2.queue_event( player2, pInitialBall );

  // ...

  // Up until here no state machines exist yet. They
  // will be created when operator()() is called

  // Run first scheduler in a new thread
  boost::thread otherThread( boost::bind(
    &sc::fifo_scheduler<>::operator(), &scheduler1, 0 ) );
  scheduler2(); // Run second scheduler in this thread
  otherThread.join();

  return 0;
}
```

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

```
int main()
```

```
{
  // ...

  boost::thread thread1( boost::bind(
    &sc::fifo_scheduler<>::operator(), &scheduler1, 0 ) );
  boost::thread thread2( boost::bind(
    &sc::fifo_scheduler<>::operator(), &scheduler2, 0 ) );

  // do something else ...

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

  return 0;
}
```

Or, run both machines in the same thread:

```
int main()
{
  sc::fifo_scheduler<> scheduler1( true );

  sc::fifo_scheduler<>::processor_handle player1 =
    scheduler1.create_processor< Player >( /* ... */ );
  sc::fifo_scheduler<>::processor_handle player2 =
    scheduler1.create_processor< Player >( /* ... */ );

  // ...

  scheduler1();

  return 0;
}
```

In all the examples above, `fifo_scheduler<>::operator()()` waits on an empty event queue and will only return after a call to `fifo_scheduler<>::terminate()`. The `Player` state machine calls this function on its scheduler object right before terminating.

Revised 16 July, 2006