# The Boost Statechart Library

## Reference

## Contents

[Concepts]
- Scheduler
- FifoWorker
- ExceptionTranslator
- StateBase
- SimpleState
- State
- Event

state_machine.hpp
- Class template `state_machine`

asynchronous_state_machine.hpp
- Class template `asynchronous_state_machine`

event_processor.hpp
- Class template `event_processor`

fifo_scheduler.hpp
- Class template `fifo_scheduler`

exception_translator.hpp
- Class template `exception_translator`

null_exception_translator.hpp
- Class `null_exception_translator`

simple_state.hpp
- Enum `history_mode`
- Class template `simple_state`

state.hpp
- Class template `state`

shallow_history.hpp
- Class template `shallow_history`

deep_history.hpp
- Class template `deep_history`

event_base.hpp
- Class `event_base`

event.hpp
- Class template `event`

transition.hpp
- Class template `transition`

in_state_reaction.hpp
- Class template in_state_reaction

termination.hpp
- Class template `termination`

deferral.hpp
    Class template `deferral`
custom_reaction.hpp
    Class template `custom_reaction`
result.hpp
    Class `result`

---

# Concepts

## Scheduler concept

A Scheduler type defines the following:

- What is passed to the constructors of `event_processor<>` subtypes and how the lifetime of such objects is managed
- Whether or not multiple `event_processor<>` subtype objects can share the same queue and scheduler thread
- How events are added to the schedulers' queue
- Whether and how to wait for new events when the schedulers' queue runs empty
- Whether and what type of locking is used to ensure thread-safety
- Whether it is possible to queue events for no longer existing `event_processor<>` subtype objects and what happens when such an event is processed
- What happens when one of the serviced `event_processor<>` subtype objects propagates an exception

For a Scheduler type `S` and an object `cpc` of type `const S::processor_context` the following expressions must be well-formed and have the indicated results:

| **Expression** | **Type** | **Result** |
|---|---|---|
| `cpc.my_scheduler()` | `S &` | A reference to the scheduler |
| `cpc.my_handle()` | `S::processor_handle` | The handle identifying the `event_processor<>` subtype object |

To protect against abuse, all members of `S::processor_context` should be declared private. As a result, `event_processor<>` must be a friend of `S::processor_context`.

## FifoWorker concept

A FifoWorker type defines the following:

- Whether and how to wait for new work items when the internal work queue runs empty
- Whether and what type of locking is used to ensure thread-safety

For a FifoWorker type `F`, an object `f` of that type, a `const` object `cf` of that type, a parameterless function object `w` of arbitrary type and an `unsigned long` value `n` the following expressions/statements must be well-formed and have the indicated results:

| Expression/Statement | Type | Effects/Result |
|---|---|---|
| `F::work_item` | `boost::function0< void >` | |
| `F()` or `F( false )` | `F` | Constructs a **non-blocking** (see below) object of the FifoWorker type. In single-threaded builds the second expression is not well-formed |
| `F( true )` | `F` | Constructs a **blocking** (see below) object of the FifoWorker type. Not well-formed in single-threaded builds |
| `f.queue_work_item( w );` | | Constructs and queues an object of type `F::work_item`, passing `w` as the only argument |
| `f.terminate();` | | Creates and queues an object of type `F::work_item` that, when later executed in `operator()()`, leads to a modification of internal state so that `terminated()` henceforth returns `true` |
| `cf.terminated();` | `bool` | `true` if `terminate()` has been called and the resulting work item has been executed in `operator()()`. Returns `false` otherwise<br><br>**Must only be called from the thread that also calls `operator()()`** |
| `f( n );` | `unsigned long` | Enters a loop that, with each cycle, dequeues and calls `operator()()` on the oldest work item in the queue.<br><br>The loop is left and the number of executed work items returned if one or more of the following conditions are met:<br><br>• `f.terminated() == true`<br>• The application is single-threaded and the internal queue is empty<br>• The application is multi-threaded and the internal queue is empty and the worker was created as non-blocking<br>• `n != 0` and the number of work items that have been processed since `operator()()` was called equals `n`<br><br>If the queue is empty and none of the above conditions are met then the thread calling `operator()()` is put into a wait state until `f.queue_work_item()` is called from another thread.<br><br>**Must only be called from exactly one thread** |

| | | |
|---|---|---|
| `f();` | `unsigned long` | Has exactly the same semantics as `f( n );` with `n == 0` (see above) |

## ExceptionTranslator concept

An ExceptionTranslator type defines how C++ exceptions occurring during state machine operation are translated to exception events.

For an ExceptionTranslator object `et`, a parameterless function object `a` of arbitrary type returning `result` and a function object `eh` of arbitrary type taking a `const event_base &` parameter and returning `result` the following expression must be well-formed and have the indicated results:

| Expression | Type | Effects/Result |
|---|---|---|
| `et( a, eh );` | `result` | 1. Attempts to execute `return a();`<br>2. If `a()` propagates an exception, the exception is caught<br>3. Inside the catch block calls `eh`, passing a suitable stack-allocated model of the Event concept<br>4. Returns the result returned by `eh` |

## StateBase concept

A StateBase type is the common base of all states of a given state machine type. `state_machine<>::state_base_type` is a model of the StateBase concept.

For a StateBase type `S` and a `const` object `cs` of that type the following expressions must be well-formed and have the indicated results:

| Expression | Type | Result |
|---|---|---|
| `cs.outer_state_ptr()` | `const S *` | `0` if `cs` is an outermost state, a pointer to the direct outer state of `cs` otherwise |
| `cs.dynamic_type()` | `S::id_type` | A value unambiguously identifying the most-derived type of `cs`. `S::id_type` values are comparable with `operator==()` and `operator!=()`. An unspecified collating order can be established with `std::less< S::id_type >` |
| `cs.custom_dynamic_type_ptr< Type >()` | `const Type *` | A pointer to the custom type identifier or `0`. If `!= 0`, `Type` must match the type of the previously set pointer. This function is only available if BOOST_STATECHART_USE_NATIVE_RTTI is not defined |
| `typeid( cs )` | `const std::type_info &` | A value unambiguously identifying the most-derived type of `cs`, **if** BOOST_STATECHART_USE_NATIVE_RTTI is defined. Otherwise, a value identifying the StateBase type is returned |

## SimpleState concept

A SimpleState type defines one state of a particular state machine.

For a SimpleState type `S` and a pointer `pS` pointing to an object of type `S` allocated with `new` the following expressions/statements must be well-formed and have the indicated effects/results:

| Expression/Statement | Type | Effects/Result/Notes |
|---|---|---|
| `simple_state< S, C, I, h > * pB = pS;` | | `simple_state< S, C, I, h >` must be an unambiguous public base of `S`. See `simple_state<>` documentation for the requirements and semantics of `C`, `I` and `h` |
| `new S()` | `S *` | Enters the state `S`. Certain functions must not be called from `S::S()`, see `simple_state<>` documentation for more information |
| `pS->exit();` | | Exits the state `S` (first stage). The definition of an `exit` member function within models of the SimpleState concept is optional since `simple_state<>` already defines the following public member: `void exit() {}`. `exit()` is not called when a state is exited while an exception is pending, see `simple_state<>::terminate()` for more information |
| `delete pS;` | | Exits the state `S` (second stage) |
| `S::reactions` | An `mpl::list<>` that is either empty or contains instantiations of the `custom_reaction`, `in_state_reaction`, `deferral`, `termination` or `transition` class templates. If there is only a single reaction then it can also be `typedef`ed directly, without wrapping it into an `mpl::list<>` | The declaration of a `reactions` member `typedef` within models of the SimpleState concept is optional since `simple_state<>` already defines the following public member: `typedef mpl::list<> reactions;` |

## State concept

A State is a **refinement** of SimpleState (that is, except for the default constructor a State type must also satisfy SimpleState requirements). For a State type `S`, a pointer `pS` of type `S *` pointing to an object of type `S` allocated with `new`, and an object `mc` of type `state< S, C, I, h >::my_context` the following expressions/statements must be well-formed:

| Expression/Statement | Type | Effects/Result/Notes |
|---|---|---|
| `state< S, C, I, h > * pB = pS;` | | `state< S, C, I, h >` must be an unambiguous public base of `S`. See |

| | | |
|---|---|---|
| | | `state<>` documentation for the requirements and semantics of `C`, `I` and `h` |
| `new S( mc )` | `S *` | Enters the state `S`. No restrictions exist regarding the functions that can be called from `S::S()` (in contrast to the constructors of models of the SimpleState concept). `mc` must be forwarded to `state< S, C, I, h >::state()` |

## Event concept

A Event type defines an event for which state machines can define reactions.

For a Event type `E` and a pointer `pCE` of type `const E *` pointing to an object of type `E` allocated with `new` the following expressions/statements must be well-formed and have the indicated effects/results:

| Expression/Statement | Type | Effects/Result/Notes |
|---|---|---|
| `const event< E > * pCB = pCE;` | | `event< E >` must be an unambiguous public base of `E` |
| `new E( *pCE )` | `E *` | Makes a copy of `pE` |

# Header <boost/statechart/state_machine.hpp>

## Class template `state_machine`

This is the base class template of all synchronous state machines.

### Class template `state_machine` parameters

| Template parameter | Requirements | Semantics | Default |
|---|---|---|---|
| `MostDerived` | The most-derived subtype of this class template | | |
| `InitialState` | A model of the SimpleState or State concepts. The `Context` argument passed to the `simple_state<>` or `state<>` base of `InitialState` must be `MostDerived`. That is, `InitialState` | The state that is entered when `state_machine<>::initiate()` is called | |

| | | | |
|---|---|---|---|
| | must be an outermost state of this state machine | | |
| `Allocator` | A model of the standard Allocator concept | | `std::allocator< void >` |
| `ExceptionTranslator` | A model of the ExceptionTranslator concept | see ExceptionTranslator concept | `null_exception_translator` |

## Class template `state_machine` synopsis

```
namespace boost
{
namespace statechart
{
  template<
    class MostDerived,
    class InitialState,
    class Allocator = std::allocator< void >,
    class ExceptionTranslator = null_exception_translator >
  class state_machine : noncopyable
  {
    public:
      typedef MostDerived outermost_context_type;

      void initiate();
      void terminate();
      bool terminated() const;

      void process_event( const event_base & );

      template< class Target >
      Target state_cast() const;
      template< class Target >
      Target state_downcast() const;

      // a model of the StateBase concept
      typedef implementation-defined state_base_type;
      // a model of the standard Forward Iterator concept
      typedef implementation-defined state_iterator;

      state_iterator state_begin() const;
      state_iterator state_end() const;

      void unconsumed_event( const event_base & ) {}

    protected:
      state_machine();
      ~state_machine();
```

```
      void post_event(
        const intrusive_ptr< const event_base > & );
      void post_event( const event_base & );
  };
}
}
```

## Class template `state_machine` constructor and destructor

```
state_machine();
```

**Effects**: Constructs a non-running state machine

```
~state_machine();
```

**Effects**: Destructs the currently active outermost state and all its direct and indirect inner states. Innermost states are destructed first. Other states are destructed as soon as all their direct and indirect inner states have been destructed. The inner states of each state are destructed according to the number of their orthogonal region. The state in the orthogonal region with the highest number is always destructed first, then the state in the region with the second-highest number and so on
**Note**: Does not attempt to call any `exit` member functions

## Class template `state_machine` modifier functions

```
void initiate();
```

**Effects**:

1. Calls `terminate()`
2. Constructs a function object `action` with a parameter-less `operator()()` returning `result` that
   a. enters (constructs) the state specified with the `InitialState` template parameter
   b. enters the tree formed by the direct and indirect inner initial states of `InitialState` depth first. The inner states of each state are entered according to the number of their orthogonal region. The state in orthogonal region 0 is always entered first, then the state in region 1 and so on
3. Constructs a function object `exceptionEventHandler` with an `operator()()` returning `result` and accepting an exception event parameter that processes the passed exception event, with the following differences to the processing of normal events:
   - From the moment when the exception has been thrown until right after the execution of the exception event reaction, states that need to be exited are only destructed but no `exit` member functions are called
   - Reaction search always starts with the outermost unstable state
   - As for normal events, reaction search moves outward when the current state cannot handle the event. However, if there is no outer state (an outermost state has been reached) the reaction search is considered unsuccessful. That is, exception events will never be dispatched to orthogonal regions other than the one that caused the exception event
   - Should an exception be thrown during exception event reaction search or reaction execution then the exception is propagated out of the `exceptionEventHandler` function object (that is, `ExceptionTranslator` is **not** used to translate exceptions thrown while processing an exception event)

- If no reaction could be found for the exception event or if the state machine is not stable after processing the exception event, the original exception is rethrown. Otherwise, a `result` object is returned equal to the one returned by `simple_state<>::discard_event()`

4. Passes `action` and `exceptionEventHandler` to `ExceptionTranslator::operator()()`. If `ExceptionTranslator::operator()()` throws an exception, the exception is propagated to the caller. If the caller catches the exception, the currently active outermost state and all its direct and indirect inner states are destructed. Innermost states are destructed first. Other states are destructed as soon as all their direct and indirect inner states have been destructed. The inner states of each state are destructed according to the number of their orthogonal region. The state in the orthogonal region with the highest number is always destructed first, then the state in the region with the second-highest number and so on. Continues with step 5 otherwise (the return value is discarded)
5. Processes all posted events (see `process_event()`). Returns to the caller if there are no more posted events

**Throws**: Any exceptions propagated from `ExceptionTranslator::operator()()`. Exceptions never originate in the library itself but only in code supplied through template parameters:

- `operator new()` (used to allocate states)
- `Allocator::allocate()`
- state constructors
- `react` member functions
- `exit` member functions
- transition-actions

```
void terminate();
```

**Effects**:

1. Constructs a function object `action` with a parameter-less `operator()()` returning `result` that terminates the currently active outermost state, discards all remaining events and clears all history information
2. Constructs a function object `exceptionEventHandler` with an `operator()()` returning `result` and accepting an exception event parameter that processes the passed exception event, with the following differences to the processing of normal events:
   - From the moment when the exception has been thrown until right after the execution of the exception event reaction, states that need to be exited are only destructed but no `exit` member functions are called
   - Reaction search always starts with the outermost unstable state
   - As for normal events, reaction search moves outward when the current state cannot handle the event. However, if there is no outer state (an outermost state has been reached) the reaction search is considered unsuccessful. That is, exception events will never be dispatched to orthogonal regions other than the one that caused the exception event
   - Should an exception be thrown during exception event reaction search or reaction execution then the exception is propagated out of the `exceptionEventHandler` function object (that is, `ExceptionTranslator` is **not** used to translate exceptions thrown while processing an exception event)
   - If no reaction could be found for the exception event or if the state machine is not stable after processing the exception event, the original exception is rethrown. Otherwise, a

   `result` object is returned equal to the one returned by `simple_state<>::discard_event()`
3. Passes `action` and `exceptionEventHandler` to `ExceptionTranslator::operator()()`. If `ExceptionTranslator::operator()()` throws an exception, the exception is propagated to the caller. If the caller catches the exception, the currently active outermost state and all its direct and indirect inner states are destructed. Innermost states are destructed first. Other states are destructed as soon as all their direct and indirect inner states have been destructed. The inner states of each state are destructed according to the number of their orthogonal region. The state in the orthogonal region with the highest number is always destructed first, then the state in the region with the second-highest number and so on. Otherwise, returns to the caller

**Throws**: Any exceptions propagated from `ExceptionTranslator::operator()`. Exceptions never originate in the library itself but only in code supplied through template parameters:

- `operator new()` (used to allocate states)
- `Allocator::allocate()`
- state constructors
- `react` member functions
- `exit` member functions
- transition-actions

```
void process_event( const event_base & );
```

**Effects**:

1. Selects the passed event as the current event (henceforth referred to as `currentEvent`)
2. Starts a new reaction search
3. Selects an arbitrary but in this reaction search not yet visited state from all the currently active innermost states. If no such state exists then continues with step 10
4. Constructs a function object `action` with a parameter-less `operator()()` returning `result` that does the following:
   a. Searches a reaction suitable for `currentEvent`, starting with the current innermost state and moving outward until a state defining a reaction for the event is found. Returns `simple_state<>::forward_event()` if no reaction has been found
   b. Executes the found reaction. If the reaction result is equal to the return value of `simple_state<>::forward_event()` then resumes the reaction search (step a). Returns the reaction result otherwise
5. Constructs a function object `exceptionEventHandler` returning `result` and accepting an exception event parameter that processes the passed exception event, with the following differences to the processing of normal events:
   - From the moment when the exception has been thrown until right after the execution of the exception event reaction, states that need to be exited are only destructed but no `exit` member functions are called
   - If the state machine is stable when the exception event is processed then exception event reaction search starts with the innermost state that was last visited during the last normal event reaction search (the exception event was generated as a result of this normal reaction search)
   - If the state machine is unstable when the exception event is processed then exception event reaction search starts with the outermost unstable state
   - As for normal events, reaction search moves outward when the current state cannot

handle the event. However, if there is no outer state (an outermost state has been reached) the reaction search is considered unsuccessful. That is, exception events will never be dispatched to orthogonal regions other than the one that caused the exception event
- Should an exception be thrown during exception event reaction search or reaction execution then the exception is propagated out of the `exceptionEventHandler` function object (that is, `ExceptionTranslator` is **not** used to translate exceptions thrown while processing an exception event)
- If no reaction could be found for the exception event or if the state machine is not stable after processing the exception event, the original exception is rethrown. Otherwise, a `result` object is returned equal to the one returned by `simple_state<>::discard_event()`

6. Passes `action` and `exceptionEventHandler` to `ExceptionTranslator::operator()()`. If `ExceptionTranslator::operator()()` throws an exception, the exception is propagated to the caller. If the caller catches the exception, the currently active outermost state and all its direct and indirect inner states are destructed. Innermost states are destructed first. Other states are destructed as soon as all their direct and indirect inner states have been destructed. The inner states of each state are destructed according to the number of their orthogonal region. The state in the orthogonal region with the highest number is always destructed first, then the state in the region with the second-highest number and so on. Otherwise continues with step 7
7. If the return value of `ExceptionTranslator::operator()()` is equal to the one of `simple_state<>::forward_event()` then continues with step 3
8. If the return value of `ExceptionTranslator::operator()()` is equal to the one of `simple_state<>::defer_event()` then the return value of `currentEvent.intrusive_from_this()` is stored in a state-specific queue. Continues with step 11
9. If the return value of `ExceptionTranslator::operator()()` is equal to the one of `simple_state<>::discard_event()` then continues with step 11
10. Calls `static_cast< MostDerived * >( this )->unconsumed_event ( currentEvent )`. If `unconsumed_event()` throws an exception, the exception is propagated to the caller. Such an exception never leads to the destruction of any states (in contrast to exceptions propagated from `ExceptionTranslator::operator()()`)
11. If the posted events queue is non-empty then dequeues the first event, selects it as `currentEvent` and continues with step 2. Returns to the caller otherwise

**Throws**: Any exceptions propagated from `MostDerived::unconsumed_event()` or `ExceptionTranslator::operator()`. Exceptions never originate in the library itself but only in code supplied through template parameters:

- `operator new()` (used to allocate states)
- `Allocator::allocate()`
- state constructors
- `react` member functions
- `exit` member functions
- transition-actions
- `MostDerived::unconsumed_event()`

```
void post_event(
  const intrusive_ptr< const event_base > & );
```

**Effects**: Pushes the passed event into the posted events queue
**Throws**: Any exceptions propagated from `Allocator::allocate()`

```
void post_event( const event_base & evt );
```

**Effects**: `post_event( evt.intrusive_from_this() );`
**Throws**: Any exceptions propagated from `Allocator::allocate()`

```
void unconsumed_event( const event_base & evt );
```

**Effects**: None
**Note**: This function (or, if present, the equally named derived class member function) is called by process_event() whenever a dispatched event did not trigger a reaction, see process_event() effects, point 10 for more information.

## Class template `state_machine` observer functions

```
bool terminated() const;
```

**Returns**: `true`, if the machine is terminated. Returns `false` otherwise
**Note**: Is equivalent to `state_begin() == state_end()`

```
template< class Target >
Target state_cast() const;
```

**Returns**: Depending on the form of `Target` either a reference or a pointer to `const` if at least one of the currently active states can successfully be `dynamic_cast` to `Target`. Returns `0` for pointer targets and throws `std::bad_cast` for reference targets otherwise. `Target` can take either of the following forms: `const Class *` or `const Class &`
**Throws**: `std::bad_cast` if `Target` is a reference type and none of the active states can be `dynamic_cast` to Target
**Note**: The search sequence is the same as for `process_event()`

```
template< class Target >
Target state_downcast() const;
```

**Requires**: For reference targets the compiler must support partial specialization of class templates, otherwise a compile-time error will result. The type denoted by `Target` must be a model of the SimpleState or State concepts
**Returns**: Depending on the form of `Target` either a reference or a pointer to `const` if `Target` is equal to the most-derived type of a currently active state. Returns `0` for pointer targets and throws `std::bad_cast` for reference targets otherwise. `Target` can take either of the following forms: `const Class *` or `const Class &`
**Throws**: `std::bad_cast` if `Target` is a reference type and none of the active states has a most derived type equal to `Target`
**Note**: The search sequence is the same as for `process_event()`

```
state_iterator state_begin() const;
```

```
state_iterator state_end() const;
```

**Return**: Iterator objects, the range [`state_begin()`, `state_end()`) refers to all currently

active innermost states. For an object `i` of type `state_iterator`, `*i` returns a `const state_base_type &` and `i.operator->()` returns a `const state_base_type *`
**Note**: The position of a given innermost state in the range is arbitrary. It may change with each call to a modifier function. Moreover, all iterators are invalidated whenever a modifier function is called

# Header <boost/statechart/asynchronous_state_machine.hpp>

## Class template `asynchronous_state_machine`

This is the base class template of all asynchronous state machines.

### Class template `asynchronous_state_machine` parameters

| Template parameter | Requirements | Semantics | Default |
|---|---|---|---|
| `MostDerived` | The most-derived subtype of this class template | | |
| `InitialState` | A model of the SimpleState or State concepts. The `Context` argument passed to the `simple_state<>` or `state<>` base of `InitialState` must be `MostDerived`. That is, `InitialState` must be an outermost state of this state machine | The state that is entered when the state machine is initiated through the `Scheduler` object | |
| `Scheduler` | A model of the Scheduler concept | see Scheduler concept | `fifo_scheduler<>` |
| `Allocator` | A model of the standard Allocator concept | | `std::allocator< void >` |
| `ExceptionTranslator` | A model of the ExceptionTranslator concept | see ExceptionTranslator concept | `null_exception_translator` |

### Class template `asynchronous_state_machine` synopsis

```
namespace boost
{
namespace statechart
{
```

```
    template<
      class MostDerived,
      class InitialState,
      class Scheduler = fifo_scheduler<>,
      class Allocator = std::allocator< void >,
      class ExceptionTranslator = null_exception_translator >
    class asynchronous_state_machine :
      public state_machine<
        MostDerived, InitialState, Allocator, ExceptionTranslator >
      public event_processor< Scheduler >
    {
      protected:
        typedef asynchronous_state_machine my_base;

        asynchronous_state_machine(
          typename event_processor< Scheduler >::my_context ctx );
        ~asynchronous_state_machine();
    };
  }
  }
```

### Class template `asynchronous_state_machine` constructor and destructor

```
  asynchronous_state_machine(
    typename event_processor< Scheduler >::my_context ctx );
```

**Effects**: Constructs a non-running asynchronous state machine
**Note**: Users cannot create `asynchronous_state_machine<>` subtype objects directly. This can only be done through an object of the `Scheduler` class

```
  ~asynchronous_state_machine();
```

**Effects**: Destructs the state machine
**Note**: Users cannot destruct `asynchronous_state_machine<>` subtype objects directly. This can only be done through an object of the `Scheduler` class

# Header <boost/statechart/event_processor.hpp>

## Class template `event_processor`

This is the base class template of all types that process events. `asynchronous_state_machine<>` is just one possible event processor implementation.

### Class template `event_processor` parameters

| Template parameter | Requirements | Semantics | Default |
|---|---|---|---|

| | | | |
|---|---|---|---|
| `Scheduler` | A model of the Scheduler concept | see Scheduler concept | |

## Class template `event_processor` synopsis

```
namespace boost
{
namespace statechart
{
  template< class Scheduler >
  class event_processor
  {
    public:
      virtual ~event_processor();

      Scheduler & my_scheduler() const;

      typedef typename Scheduler::processor_handle
        processor_handle;
      processor_handle my_handle() const;

      void initiate();
      void process_event( const event_base & evt );
      void terminate();

    protected:
      typedef const typename Scheduler::processor_context &
        my_context;
      event_processor( my_context ctx );

    private:
      virtual void initiate_impl() = 0;
      virtual void process_event_impl(
        const event_base & evt ) = 0;
      virtual void terminate_impl() = 0;
  };
}
}
```

## Class template `event_processor` constructor and destructor

```
event_processor( my_context ctx );
```

**Effects**: Constructs an event processor object and stores copies of the reference returned by `myContext.my_scheduler()` and the object returned by `myContext.my_handle()`
**Note**: Users cannot create `event_processor<>` subtype objects directly. This can only be done through an object of the `Scheduler` class

```
virtual ~event_processor();
```

**Effects**: Destructs an event processor object

**Note**: Users cannot destruct `event_processor<>` subtype objects directly. This can only be done through an object of the `Scheduler` class

### Class template `event_processor` modifier functions

```
void initiate();
```

**Effects**: `initiate_impl();`
**Throws**: Any exceptions propagated from the implementation of `initiate_impl()`

```
void process_event( const event_base & evt );
```

**Effects**: `process_event_impl( evt );`
**Throws**: Any exceptions propagated from the implementation of `process_event_impl()`

```
void terminate();
```

**Effects**: `terminate_impl();`
**Throws**: Any exceptions propagated from the implementation of `terminate_impl()`

### Class template `event_processor` observer functions

```
Scheduler & my_scheduler() const;
```

**Returns**: The `Scheduler` reference obtained in the constructor

```
processor_handle my_handle() const;
```

**Returns**: The `processor_handle` object obtained in the constructor

# Header <boost/statechart/fifo_scheduler.hpp>

## Class template `fifo_scheduler`

This class template is a model of the Scheduler concept.

### Class template `fifo_scheduler` parameters

| Template parameter | Requirements | Semantics | Default |
|---|---|---|---|
| `FifoWorker` | A model of the FifoWorker concept | see FifoWorker concept | `fifo_worker<>` |
| `Allocator` | A model of the standard Allocator concept | | `std::allocator< void >` |

### Class template `fifo_scheduler` synopsis

```
namespace boost
```

```
{
namespace statechart
{
  template<
    class FifoWorker = fifo_worker<>,
    class Allocator = std::allocator< void > >
  class fifo_scheduler : noncopyable
  {
    public:
      fifo_scheduler( bool waitOnEmptyQueue = false );

      typedef implementation-defined processor_handle;

      class processor_context : noncopyable
      {
        processor_context(
          fifo_scheduler & scheduler,
          const processor_handle & theHandle );

        fifo_scheduler & my_scheduler() const;
        const processor_handle & my_handle() const;

        friend class fifo_scheduler;
        friend class event_processor< fifo_scheduler >;
      };

      template< class Processor >
      processor_handle create_processor();
      template< class Processor, typename Param1 >
      processor_handle create_processor( Param1 param1 );

      // More create_processor overloads

      void destroy_processor( processor_handle processor );

      void initiate_processor( processor_handle processor );
      void terminate_processor( processor_handle processor );

      typedef intrusive_ptr< const event_base > event_ptr_type;

      void queue_event(
        const processor_handle & processor,
        const event_ptr_type & pEvent );

      typedef typename FifoWorker::work_item work_item;

      void queue_work_item( const work_item & item );

      void terminate();
      bool terminated() const;

      unsigned long operator()(
        unsigned long maxEventCount = 0 );
```

```
    };
  }
  }
```

## Class template `fifo_scheduler` constructor

```
fifo_scheduler( bool waitOnEmptyQueue = false );
```

**Effects**: Constructs a `fifo_scheduler<>` object. In multi-threaded builds, `waitOnEmptyQueue` is forwarded to the constructor of a data member of type `FifoWorker`. In single-threaded builds, the `FifoWorker` data member is default-constructed
**Note**: In single-threaded builds the `fifo_scheduler<>` constructor does not accept any parameters and `operator()()` thus always returns to the caller when the event queue is empty

## Class template `fifo_scheduler` modifier functions

```
template< class Processor >
processor_handle create_processor();
```

**Requires**: The `Processor` type must be a direct or indirect subtype of the `event_processor` class template
**Effects**: Creates and passes to `FifoWorker::queue_work_item()` an object of type `FifoWorker::work_item` that, when later executed in `FifoWorker::operator()()`, leads to a call to the constructor of `Processor`, passing an appropriate `processor_context` object as the only argument
**Returns**: A `processor_handle` object that henceforth identifies the created event processor object
**Throws**: Any exceptions propagated from `FifoWorker::work_item()` and `FifoWorker::queue_work_item()`
**Caution**: The current implementation of this function makes an (indirect) call to global `operator new()`. Unless global `operator new()` is replaced, care must be taken when to call this function in applications with hard real-time requirements

```
template< class Processor, typename Param1 >
processor_handle create_processor( Param1 param1 );
```

**Requires**: The `Processor` type must be a direct or indirect subtype of the `event_processor` class template
**Effects**: Creates and passes to `FifoWorker::queue_work_item()` an object of type `FifoWorker::work_item` that, when later executed in `FifoWorker::operator()()`, leads to a call to the constructor of `Processor`, passing an appropriate `processor_context` object and `param1` as arguments
**Returns**: A `processor_handle` object that henceforth identifies the created event processor object
**Throws**: Any exceptions propagated from `FifoWorker::work_item()` and `FifoWorker::queue_work_item()`
**Note**: `boost::ref()` and `boost::cref()` can be used to pass arguments by reference rather than by copy. `fifo_scheduler<>` has 5 additional `create_processor<>` overloads, allowing to pass up to 6 custom arguments to the constructors of event processors
**Caution**: The current implementation of this and all other overloads make (indirect) calls to global `operator new()`. Unless global `operator new()` is replaced, care must be taken when to

call these overloads in applications with hard real-time requirements

```
void destroy_processor( processor_handle processor );
```

**Requires**: `processor` was obtained from a call to one of the `create_processor<>()` overloads on the same `fifo_scheduler<>` object
**Effects**: Creates and passes to `FifoWorker::queue_work_item()` an object of type `FifoWorker::work_item` that, when later executed in `FifoWorker::operator()()`, leads to a call to the destructor of the event processor object associated with `processor`. The object is silently discarded if the event processor object has been destructed before
**Throws**: Any exceptions propagated from `FifoWorker::work_item()` and `FifoWorker::queue_work_item()`
**Caution**: The current implementation of this function leads to an (indirect) call to global `operator delete()` (the call is made when the last `processor_handle` object associated with the event processor object is destructed). Unless global `operator delete()` is replaced, care must be taken when to call this function in applications with hard real-time requirements

```
void initiate_processor( processor_handle processor );
```

**Requires**: `processor` was obtained from a call to one of the `create_processor()` overloads on the same `fifo_scheduler<>` object
**Effects**: Creates and passes to `FifoWorker::queue_work_item()` an object of type `FifoWorker::work_item` that, when later executed in `FifoWorker::operator()()`, leads to a call to `initiate()` on the event processor object associated with `processor`. The object is silently discarded if the event processor object has been destructed before
**Throws**: Any exceptions propagated from `FifoWorker::work_item()` and `FifoWorker::queue_work_item()`

```
void terminate_processor( processor_handle processor );
```

**Requires**: `processor` was obtained from a call to one of the `create_processor<>()` overloads on the same `fifo_scheduler<>` object
**Effects**: Creates and passes to `FifoWorker::queue_work_item()` an object of type `FifoWorker::work_item` that, when later executed in `FifoWorker::operator()()`, leads to a call to `terminate()` on the event processor object associated with `processor`. The object is silently discarded if the event processor object has been destructed before
**Throws**: Any exceptions propagated from `FifoWorker::work_item()` and `FifoWorker::queue_work_item()`

```
void queue_event(
  const processor_handle & processor,
  const event_ptr_type & pEvent );
```

**Requires**: `pEvent.get() != 0` and `processor` was obtained from a call to one of the `create_processor<>()` overloads on the same `fifo_scheduler<>` object
**Effects**: Creates and passes to `FifoWorker::queue_work_item()` an object of type `FifoWorker::work_item` that, when later executed in `FifoWorker::operator()()`, leads to a call to `process_event( *pEvent )` on the event processor object associated with `processor`. The object is silently discarded if the event processor object has been destructed before
**Throws**: Any exceptions propagated from `FifoWorker::work_item()` and `FifoWorker::queue_work_item()`

```
void queue_work_item( const work_item & item );
```

**Effects**: `FifoWorker::queue_work_item( item );`
**Throws**: Any exceptions propagated from the above call

```
void terminate();
```

**Effects**: `FifoWorker::terminate()`
**Throws**: Any exceptions propagated from the above call

```
unsigned long operator()( unsigned long maxEventCount = 0 );
```

**Requires**: Must only be called from exactly one thread
**Effects**: `FifoWorker::operator()( maxEventCount )`
**Returns**: The return value of the above call
**Throws**: Any exceptions propagated from the above call

### Class template `fifo_scheduler` observer functions

```
bool terminated() const;
```

**Requires**: Must only be called from the thread that also calls `operator()()`
**Returns**: `FifoWorker::terminated();`

# Header <boost/statechart/exception_translator.hpp>

## Class template `exception_translator`

This class template is a model of the ExceptionTranslator concept.

### Class template `exception_translator` parameters

| Template parameter | Requirements | Semantics | Default |
|---|---|---|---|
| `ExceptionEvent` | A model of the Event concept | The type of event that is dispatched when an exception is propagated into the framework | `exception_thrown` |

### Class template `exception_translator` synopsis & semantics

```
namespace boost
{
namespace statechart
{
  class exception_thrown : public event< exception_thrown > {};
```

```
    template< class ExceptionEvent = exception_thrown >
    class exception_translator
    {
      public:
        template< class Action, class ExceptionEventHandler >
        result operator()(
          Action action,
          ExceptionEventHandler eventHandler )
        {
          try
          {
            return action();
          }
          catch( ... )
          {
            return eventHandler( ExceptionEvent() );
          }
        }
    };
  }
  }
```

# Header <boost/statechart/null_exception_translator.hpp>

## Class `null_exception_translator`

This class is a model of the ExceptionTranslator concept.

### Class `null_exception_translator` synopsis & semantics

```
  namespace boost
  {
  namespace statechart
  {
    class null_exception_translator
    {
      public:
        template< class Action, class ExceptionEventHandler >
        result operator()(
          Action action, ExceptionEventHandler )
        {
          return action();
        }
    };
  }
  }
```

# Header <boost/statechart/simple_state.hpp>

# Enum `history_mode`

Defines the history type of a state.

```
namespace boost
{
namespace statechart
{
  enum history_mode
  {
    has_no_history,
    has_shallow_history,
    has_deep_history,
    has_full_history // shallow & deep
  };
}
}
```

# Class template `simple_state`

This is the base class template for all models of the SimpleState concept. Such models must not call any of the following `simple_state<>` member functions from their constructors:

```
void post_event(
  const intrusive_ptr< const event_base > & );
void post_event( const event_base & );

template<
  class HistoryContext,
  implementation-defined-unsigned-integer-type
    orthogonalPosition >
void clear_shallow_history();
template<
  class HistoryContext,
  implementation-defined-unsigned-integer-type
    orthogonalPosition >
void clear_deep_history();

outermost_context_type & outermost_context();
const outermost_context_type & outermost_context() const;

template< class OtherContext >
OtherContext & context();
template< class OtherContext >
const OtherContext & context() const;

template< class Target >
Target state_cast() const;
template< class Target >
Target state_downcast() const;

state_iterator state_begin() const;
```

```
state_iterator state_end() const;
```

States that need to call any of these member functions from their constructors must derive from the `state` class template.

## Class template `simple_state` parameters

| Template parameter | Requirements | Semantics | Default |
|---|---|---|---|
| `MostDerived` | The most-derived subtype of this class template | | |
| `Context` | A most-derived direct or indirect subtype of the `state_machine` or `asynchronous_state_machine` class templates or a model of the SimpleState or State concepts or an instantiation of the `simple_state<>::orthogonal` class template. Must be a complete type | Defines the states' position in the state hierarchy | |
| `InnerInitial` | An `mpl::list<>` containing models of the SimpleState or State concepts or instantiations of the `shallow_history` or `deep_history` class templates. If there is only a single inner initial state that is not a template instantiation then it can also be passed directly, without wrapping it into an `mpl::list<>`. The `Context` argument passed to the `simple_state<>` or `state<>` base of each state in the list must correspond to the orthogonal region it belongs to. That is, the first state in the list must pass `MostDerived::orthogonal< 0 >`, the second `MostDerived::orthogonal< 1 >` and so forth. `MostDerived::orthogonal< 0 >` and `MostDerived` are synonymous | Defines the inner initial state for each orthogonal region. By default, a state does not have inner states | *`unspecified`* |
| `historyMode` | One of the values defined in the `history_mode` enumeration | Defines whether the state saves shallow, deep or both histories upon exit | `has_no_history` |

## Class template `simple_state` synopsis

```
namespace boost
{
namespace statechart
{
  template<
    class MostDerived,
```

```
    class Context,
    class InnerInitial = unspecified,
    history_mode historyMode = has_no_history >
  class simple_state : implementation-defined
  {
    public:
      // by default, a state has no reactions
      typedef mpl::list<> reactions;

      // see template parameters
      template< implementation-defined-unsigned-integer-type
        innerOrthogonalPosition >
      struct orthogonal
      {
        // implementation-defined
      };

      typedef typename Context::outermost_context_type
        outermost_context_type;

      outermost_context_type & outermost_context();
      const outermost_context_type & outermost_context() const;

      template< class OtherContext >
      OtherContext & context();
      template< class OtherContext >
      const OtherContext & context() const;

      template< class Target >
      Target state_cast() const;
      template< class Target >
      Target state_downcast() const;

      // a model of the StateBase concept
      typedef implementation-defined state_base_type;
      // a model of the standard Forward Iterator concept
      typedef implementation-defined state_iterator;

      state_iterator state_begin() const;
      state_iterator state_end() const;

      void post_event(
        const intrusive_ptr< const event_base > & );
      void post_event( const event_base & );

      result discard_event();
      result forward_event();
      result defer_event();
      template< class DestinationState >
      result transit();
      template<
        class DestinationState,
        class TransitionContext,
```

```
          class Event >
        result transit(
          void ( TransitionContext::* )( const Event & ),
          const Event & );
        result terminate();

        template<
          class HistoryContext,
          implementation-defined-unsigned-integer-type
            orthogonalPosition >
        void clear_shallow_history();
        template<
          class HistoryContext,
          implementation-defined-unsigned-integer-type
            orthogonalPosition >
        void clear_deep_history();

        static id_type static_type();

        template< class CustomId >
        static const CustomId * custom_static_type_ptr();

        template< class CustomId >
        static void custom_static_type_ptr( const CustomId * );

        // see transit() or terminate() effects
        void exit() {}

      protected:
        simple_state();
        ~simple_state();
    };
  }
  }
```

## Class template `simple_state` constructor and destructor

```
simple_state();
```

**Requires**: The constructors of all direct and indirect subtypes must be exception-neutral
**Effects**: Constructs a state object
**Throws**: Any exceptions propagated from `Allocator::allocate()` (the template parameter passed to the base class of `outermost_context_type`)

```
~simple_state();
```

**Effects**: Pushes all events deferred by the state into the posted events queue

## Class template `simple_state` modifier functions

```
void post_event(
  const intrusive_ptr< const event_base > & pEvt );
```

**Requires**: If called from a constructor of a direct or indirect subtype then the most-derived type must directly or indirectly derive from the `state` class template**.** All direct and indirect callers must be exception-neutral
**Effects**: `outermost_context().post_event( pEvt );`
**Throws**: Whatever the above call throws

```
void post_event( const event_base & evt );
```

**Requires**: If called from a constructor of a direct or indirect subtype then the most-derived type must directly or indirectly derive from the `state` class template**.** All direct and indirect callers must be exception-neutral
**Effects**: `outermost_context().post_event( evt );`
**Throws**: Whatever the above call throws

```
result discard_event();
```

**Requires**: Must only be called from within `react` member functions, which are called by `custom_reaction<>` instantiations. All direct and indirect callers must be exception-neutral
**Effects**: Instructs the state machine to discard the current event and to continue with the processing of the remaining events (see `state_machine<>::process_event()` for details)
**Returns**: A `result` object. The user-supplied `react` member function must return this object to its caller

```
result forward_event();
```

**Requires**: Must only be called from within `react` member functions, which are called by `custom_reaction<>` instantiations. All direct and indirect callers must be exception-neutral
**Effects**: Instructs the state machine to forward the current event to the next state (see `state_machine<>::process_event()` for details)
**Returns**: A `result` object. The user-supplied `react` member function must return this object to its caller

```
result defer_event();
```

**Requires**: Must only be called from within `react` member functions, which are called by `custom_reaction<>` instantiations. All direct and indirect callers must be exception-neutral
**Effects**: Instructs the state machine to defer the current event and to continue with the processing of the remaining events (see `state_machine<>::process_event()` for details)
**Returns**: A `result` object. The user-supplied `react` member function must return this object to its caller
**Throws**: Any exceptions propagated from `Allocator::allocate()` (the template parameter passed to the base class of `outermost_context_type`)

```
template< class DestinationState >
result transit();
```

**Requires**: Must only be called from within `react` member functions, which are called by `custom_reaction<>` instantiations. All direct and indirect callers must be exception-neutral
**Effects**:

1. Exits all currently active direct and indirect inner states of the innermost common context of this state and `DestinationState`. Innermost states are exited first. Other states are exited

as soon as all their direct and indirect inner states have been exited. The inner states of each state are exited according to the number of their orthogonal region. The state in the orthogonal region with the highest number is always exited first, then the state in the region with the second-highest number and so on.
The process of exiting a state consists of the following steps:
   1. If there is an exception pending that has not yet been handled successfully then only step 5 is executed
   2. Calls the `exit` member function (see synopsis) of the most-derived state object. If `exit()` throws then steps 3 and 4 are not executed
   3. If the state has shallow history then shallow history information is saved
   4. If the state is an innermost state then deep history information is saved for all direct and indirect outer states that have deep history
   5. The state object is destructed
2. Enters (constructs) the state that is both a direct inner state of the innermost common context and either the `DestinationState` itself or a direct or indirect outer state of `DestinationState`
3. Enters (constructs) the tree formed by the direct and indirect inner states of the previously entered state down to the `DestinationState` and beyond depth first. The inner states of each state are entered according to the number of their orthogonal region. The state in orthogonal region 0 is always entered first, then the state in region 1 and so on
4. Instructs the state machine to discard the current event and to continue with the processing of the remaining events (see `state_machine<>::process_event()` for details)

**Returns**: A `result` object. The user-supplied `react` member function must return this object to its caller
**Throws**: Any exceptions propagated from:

- `operator new()` (used to allocate states)
- `Allocator::allocate()` (the template parameter passed to the base class of `outermost_context_type`)
- state constructors
- `exit` member functions

**Caution**: Inevitably destructs this state before returning to the calling `react` member function, which must therefore not attempt to access anything except stack objects before returning to its caller

```
template<
  class DestinationState,
  class TransitionContext,
  class Event >
result transit(
  void ( TransitionContext::* )( const Event & ),
  const Event & );
```

**Requires**: Must only be called from within `react` member functions, which are called by `custom_reaction<>` instantiations. All direct and indirect callers must be exception-neutral
**Effects**:

1. Exits all currently active direct and indirect inner states of the innermost common context of this state and `DestinationState`. Innermost states are exited first. Other states are exited as soon as all their direct and indirect inner states have been exited. The inner states of each state are exited according to the number of their orthogonal region. The state in the orthogonal region with the highest number is always exited first, then the state in the region with the

second-highest number and so on.
The process of exiting a state consists of the following steps:
   1. If there is an exception pending that has not yet been handled successfully then only step 5 is executed
   2. Calls the `exit` member function (see synopsis) of the most-derived state object. If `exit()` throws then steps 3 and 4 are not executed
   3. If the state has shallow history then shallow history information is saved
   4. If the state is an innermost state then deep history information is saved for all direct and indirect outer states that have deep history
   5. The state object is destructed
2. Executes the passed transition action, forwarding the passed event
3. Enters (constructs) the state that is both a direct inner state of the innermost common context and either the `DestinationState` itself or a direct or indirect outer state of `DestinationState`
4. Enters (constructs) the tree formed by the direct and indirect inner states of the previously entered state down to the `DestinationState` and beyond depth first. The inner states of each state are entered according to the number of their orthogonal region. The state in orthogonal region 0 is always entered first, then the state in region 1 and so on
5. Instructs the state machine to discard the current event and to continue with the processing of the remaining events (see `state_machine<>::process_event()` for details)

**Returns**: A `result` object. The user-supplied `react` member function must return this object to its caller
**Throws**: Any exceptions propagated from:

- `operator new()` (used to allocate states)
- `Allocator::allocate()` (the template parameter passed to the base class of `outermost_context_type`)
- state constructors
- `exit` member functions
- the transition action

**Caution**: Inevitably destructs this state before returning to the calling `react` member function, which must therefore not attempt to access anything except stack objects before returning to its caller

```
result terminate();
```

**Requires**: Must only be called from within `react` member functions, which are called by `custom_reaction<>` instantiations. All direct and indirect callers must be exception-neutral
**Effects**: Exits this state and all its direct and indirect inner states. Innermost states are exited first. Other states are exited as soon as all their direct and indirect inner states have been exited. The inner states of each state are exited according to the number of their orthogonal region. The state in the orthogonal region with the highest number is always exited first, then the state in the region with the second-highest number and so on.
The process of exiting a state consists of the following steps:

1. If there is an exception pending that has not yet been handled successfully then only step 5 is executed
2. Calls the `exit` member function (see synopsis) of the most-derived state object. If `exit()` throws then steps 3 and 4 are not executed
3. If the state has shallow history then shallow history information is saved
4. If the state is an innermost state then deep history information is saved for all direct and indirect outer states that have deep history

5. The state object is destructed

Also instructs the state machine to discard the current event and to continue with the processing of the remaining events (see `state_machine<>::process_event()` for details)
**Returns**: A `result` object. The user-supplied `react` member function must return this object to its caller
**Throws**: Any exceptions propagated from:

- `Allocator::allocate()` (the template parameter passed to the base class of `outermost_context_type`, used to allocate space to save history)
- `exit` member functions

**Note**: If this state is the only currently active inner state of its direct outer state then the direct outer state is terminated also. The same applies recursively for all indirect outer states
**Caution**: Inevitably destructs this state before returning to the calling `react` member function, which must therefore not attempt to access anything except stack objects before returning to its caller

```
template<
  class HistoryContext,
  implementation-defined-unsigned-integer-type
    orthogonalPosition >
void clear_shallow_history();
```

**Requires**: If called from a constructor of a direct or indirect subtype then the most-derived type must directly or indirectly derive from the `state` class template. The `historyMode` argument passed to the `simple_state<>` or `state<>` base of `HistoryContext` must be equal to `has_shallow_history` or `has_full_history`
**Effects**: Clears the shallow history of the orthogonal region specified by `orthogonalPosition` of the state specified by `HistoryContext`
**Throws**: Any exceptions propagated from `Allocator::allocate()` (the template parameter passed to the base class of `outermost_context_type`)

```
template<
  class HistoryContext,
  implementation-defined-unsigned-integer-type
    orthogonalPosition >
void clear_deep_history();
```

**Requires**: If called from a constructor of a direct or indirect subtype then the most-derived type must directly or indirectly derive from the `state` class template. The `historyMode` argument passed to the `simple_state<>` or `state<>` base of `HistoryContext` must be equal to `has_deep_history` or `has_full_history`
**Effects**: Clears the deep history of the orthogonal region specified by `orthogonalPosition` of the state specified by `HistoryContext`
**Throws**: Any exceptions propagated from `Allocator::allocate()` (the template parameter passed to the base class of `outermost_context_type`)

## Class template `simple_state` observer functions

```
outermost_context_type & outermost_context();
```

**Requires**: If called from a constructor of a direct or indirect subtype then the most-derived type must

directly or indirectly derive from the `state` class template
**Returns**: A reference to the outermost context, which is always the state machine this state belongs to

```
const outermost_context_type & outermost_context() const;
```

**Requires**: If called from a constructor of a direct or indirect subtype then the most-derived type must directly or indirectly derive from the `state` class template
**Returns**: A reference to the const outermost context, which is always the state machine this state belongs to

```
template< class OtherContext >
OtherContext & context();
```

**Requires**: If called from a constructor of a direct or indirect subtype then the most-derived type must directly or indirectly derive from the `state` class template
**Returns**: A reference to a direct or indirect context

```
template< class OtherContext >
const OtherContext & context() const;
```

**Requires**: If called from a constructor of a direct or indirect subtype then the most-derived type must directly or indirectly derive from the `state` class template
**Returns**: A reference to a const direct or indirect context

```
template< class Target >
Target state_cast() const;
```

**Requires**: If called from a constructor of a direct or indirect subtype then the most-derived type must directly or indirectly derive from the `state` class template
**Returns**: Has exactly the same semantics as `state_machine<>::state_cast<>()`
**Throws**: Has exactly the same semantics as `state_machine<>::state_cast<>()`
**Note**: The result is **unspecified** if this function is called when the machine is unstable

```
template< class Target >
Target state_downcast() const;
```

**Requires**: If called from a constructor of a direct or indirect subtype then the most-derived type must directly or indirectly derive from the `state` class template. Moreover, `state_machine<>::state_downcast<>()` requirements also apply
**Returns**: Has exactly the same semantics as `state_machine<>::state_downcast<>()`
**Throws**: Has exactly the same semantics as `state_machine<>::state_downcast<>()`
**Note**: The result is **unspecified** if this function is called when the machine is unstable

```
state_iterator state_begin() const;

state_iterator state_end() const;
```

**Require**: If called from a constructor of a direct or indirect subtype then the most-derived type must directly or indirectly derive from the `state` class template
**Return**: Have exactly the same semantics as `state_machine<>::state_begin()` and `state_machine<>::state_end()`

**Note**: The result is **unspecified** if these functions are called when the machine is unstable

### Class template `simple_state` static functions

```
static id_type static_type();
```

**Returns**: A value unambiguously identifying the type of `MostDerived`
**Note**: `id_type` values are comparable with `operator==()` and `operator!=()`. An unspecified collating order can be established with `std::less< id_type >`

```
template< class CustomId >
static const CustomId * custom_static_type_ptr();
```

**Requires**: If a custom type identifier has been set then `CustomId` must match the type of the previously set pointer
**Returns**: The pointer to the custom type identifier for `MostDerived` or `0`
**Note**: This function is not available if `BOOST_STATECHART_USE_NATIVE_RTTI` is defined

```
template< class CustomId >
static void custom_static_type_ptr( const CustomId * );
```

**Effects**: Sets the pointer to the custom type identifier for `MostDerived`
**Note**: This function is not available if `BOOST_STATECHART_USE_NATIVE_RTTI` is defined

# Header <boost/statechart/state.hpp>

## Class template `state`

This is the base class template for all models of the State concept. Such models typically need to call at least one of the following `simple_state<>` member functions from their constructors:

```
void post_event(
  const intrusive_ptr< const event_base > & );
void post_event( const event_base & );

template<
  class HistoryContext,
  implementation-defined-unsigned-integer-type
    orthogonalPosition >
void clear_shallow_history();
template<
  class HistoryContext,
  implementation-defined-unsigned-integer-type
    orthogonalPosition >
void clear_deep_history();

outermost_context_type & outermost_context();
const outermost_context_type & outermost_context() const;

template< class OtherContext >
OtherContext & context();
```

```
template< class OtherContext >
const OtherContext & context() const;

template< class Target >
Target state_cast() const;
template< class Target >
Target state_downcast() const;

state_iterator state_begin() const;
state_iterator state_end() const;
```

States that do not need to call any of these member functions from their constructors should rather derive from the `simple_state` class template, what saves the implementation of the forwarding constructor.

### Class template `state` synopsis

```
namespace boost
{
namespace statechart
{
  template<
    class MostDerived,
    class Context,
    class InnerInitial = unspecified,
    history_mode historyMode = has_no_history >
  class state : public simple_state<
    MostDerived, Context, InnerInitial, historyMode >
  {
    protected:
      struct my_context
      {
        // implementation-defined
      };

      typedef state my_base;

      state( my_context ctx );
      ~state();
  };
}
}
```

Direct and indirect subtypes of `state<>` must provide a constructor with the same signature as the `state<>` constructor, forwarding the context parameter.

# Header <boost/statechart/shallow_history.hpp>

### Class template `shallow_history`

This class template is used to specify a shallow history transition target or a shallow history inner initial state.

### Class template `shallow_history` parameters

| Template parameter | Requirements | Semantics |
|---|---|---|
| `DefaultState` | A model of the SimpleState or State concepts. The type passed as `Context` argument to the `simple_state<>` or `state<>` base of `DefaultState` must itself pass `has_shallow_history` or `has_full_history` as `historyMode` argument to its `simple_state<>` or `state<>` base | The state that is entered if shallow history is not available |

### Class template `shallow_history` synopsis

```
namespace boost
{
namespace statechart
{
  template< class DefaultState >
  class shallow_history
  {
    // implementation-defined
  };
}
}
```

# Header <boost/statechart/deep_history.hpp>

## Class template `deep_history`

This class template is used to specify a deep history transition target or a deep history inner initial state. The current deep history implementation has some limitations.

### Class template `deep_history` parameters

| Template parameter | Requirements | Semantics |
|---|---|---|
| `DefaultState` | A model of the SimpleState or State concepts. The type passed as `Context` argument to the `simple_state<>` or `state<>` base of `DefaultState` must itself pass `has_deep_history` or `has_full_history` as `historyMode` argument to its `simple_state<>` or `state<>` base | The state that is entered if deep history is not available |

### Class template `deep_history` synopsis

```
namespace boost
{
namespace statechart
{
  template< class DefaultState >
  class deep_history
  {
    // implementation-defined
  };
}
}
```

# Header <boost/statechart/event_base.hpp>

## Class `event_base`

This is the common base of all events.

### Class `event_base` synopsis

```
namespace boost
{
namespace statechart
{
  class event_base
  {
    public:
      intrusive_ptr< const event_base >
        intrusive_from_this() const;

      typedef implementation-defined id_type;

      id_type dynamic_type() const;

      template< typename CustomId >
      const CustomId * custom_dynamic_type_ptr() const;

    protected:
      event_base( unspecified-parameter );
      virtual ~event_base();
  };
}
}
```

### Class `event_base` constructor and destructor

```
event_base( unspecified-parameter );
```

**Effects**: Constructs the common base portion of an event

```
virtual ~event_base();
```

**Effects**: Destructs the common base portion of an event

### Class `event_base` observer functions

```
intrusive_ptr< const event_base > intrusive_from_this() const;
```

**Returns**: Another `intrusive_ptr< const event_base >` referencing `this` **if** `this` is already referenced by an `intrusive_ptr<>`. Otherwise, returns an `intrusive_ptr< const event_base >` referencing a newly created copy of the most-derived object

```
id_type dynamic_type() const;
```

**Returns**: A value unambiguously identifying the most-derived type
**Note**: `id_type` values are comparable with `operator==()` and `operator!=()`. An unspecified collating order can be established with `std::less< id_type >`

```
template< typename CustomId >
const CustomId * custom_dynamic_type_ptr() const;
```

**Requires**: If a custom type identifier has been set then `CustomId` must match the type of the previously set pointer
**Returns**: A pointer to the custom type identifier or `0`
**Note**: This function is not available if `BOOST_STATECHART_USE_NATIVE_RTTI` is defined

# Header <boost/statechart/event.hpp>

## Class template `event`

This is the base class template of all events.

### Class template `event` synopsis

```
namespace boost
{
namespace statechart
{
  template< class MostDerived >
  class event : implementation-defined
  {
    public:
      static id_type static_type();

      template< class CustomId >
      static const CustomId * custom_static_type_ptr();

      template< class CustomId >
      static void custom_static_type_ptr( const CustomId * );
```

```
      protected:
        event();
        virtual ~event();
    };
  }
  }
```

### Class template `event` constructor and destructor

```
event();
```

**Effects**: Constructs an event

```
virtual ~event();
```

**Effects**: Destructs an event

### Class template `event` static functions

```
static id_type static_type();
```

**Returns**: A value unambiguously identifying the type of `MostDerived`
**Note**: `id_type` values are comparable with `operator==()` and `operator!=()`. An unspecified collating order can be established with `std::less< id_type >`

```
template< class CustomId >
static const CustomId * custom_static_type_ptr();
```

**Requires**: If a custom type identifier has been set then `CustomId` must match the type of the previously set pointer
**Returns**: The pointer to the custom type identifier for `MostDerived` or `0`
**Note**: This function is not available if `BOOST_STATECHART_USE_NATIVE_RTTI` is defined

```
template< class CustomId >
static void custom_static_type_ptr( const CustomId * );
```

**Effects**: Sets the pointer to the custom type identifier for `MostDerived`
**Note**: This function is not available if `BOOST_STATECHART_USE_NATIVE_RTTI` is defined

# Header <boost/statechart/transition.hpp>

## Class template `transition`

This class template is used to specify a transition reaction. Instantiations of this template can appear in the `reactions` member `typedef` in models of the SimpleState and State concepts.

### Class template `transition` parameters

| Template | | | |
|---|---|---|---|

| parameter | Requirements | Semantics | Default |
|---|---|---|---|
| `Event` | A model of the Event concept or the class `event_base` | The event triggering the transition. If `event_base` is specified, the transition is triggered by all models of the Event concept | |
| `Destination` | A model of the SimpleState or State concepts or an instantiation of the `shallow_history` or `deep_history` class templates. The source state (the state for which this transition is defined) and `Destination` must have a common direct or indirect context | The destination state to make a transition to | |
| `TransitionContext` | A common context of the source and `Destination` state | The state of which the transition action is a member | *`unspecified`* |
| `pTransitionAction` | A pointer to a member function of `TransitionContext`. The member function must accept a `const Event &` parameter and return `void` | The transition action that is executed during the transition. By default no transition action is executed | *`unspecified`* |

## Class template `transition` synopsis

```
namespace boost
{
namespace statechart
{
  template<
    class Event,
    class Destination,
    class TransitionContext = unspecified,
    void ( TransitionContext::*pTransitionAction )(
      const Event & ) = unspecified >
  class transition
  {
    // implementation-defined
  };
}
}
```

## Class template `transition` semantics

When executed, one of the following calls to a member function of the state for which the reaction was defined is made:

- `transit< Destination >()`, if no transition action was specified
- `transit< Destination >( pTransitionAction, currentEvent )`, if a transition action was specified

# Header <boost/statechart/in_state_reaction.hpp>

## Class template `in_state_reaction`

This class template is used to specify an in-state reaction. Instantiations of this template can appear in the `reactions` member `typedef` in models of the SimpleState and State concepts.

### Class template `in_state_reaction` parameters

| Template parameter | Requirements | Semantics |
|---|---|---|
| `Event` | A model of the Event concept or the class `event_base` | The event triggering the in-state reaction. If `event_base` is specified, the in-state reaction is triggered by all models of the Event concept |
| `ReactionContext` | Either the state defining the in-state reaction itself or one of it direct or indirect contexts | The state of which the action is a member |
| `pAction` | A pointer to a member function of `ReactionContext`. The member function must accept a `const Event &` parameter and return `void` | The action that is executed during the in-state reaction |

### Class template `in_state_reaction` synopsis

```
namespace boost
{
namespace statechart
{
  template<
    class Event,
    class ReactionContext,
    void ( ReactionContext::*pAction )( const Event & ) >
  class in_state_reaction
  {
    // implementation-defined
  };
}
}
```

### Class template `in_state_reaction` semantics

When executed, `pAction` is called, passing the triggering event as the only argument. Afterwards, a call is made to the `discard_event` member function of the state for which the reaction was defined.

# Header <boost/statechart/termination.hpp>

## Class template `termination`

This class template is used to specify a termination reaction. Instantiations of this template can appear in the `reactions` member `typedef` in models of the SimpleState and State concepts.

### Class template `termination` parameters

| Template parameter | Requirements | Semantics |
|---|---|---|
| `Event` | A model of the Event concept or the class `event_base` | The event triggering the termination. If `event_base` is specified, the termination is triggered by all models of the Event concept |

### Class template `termination` synopsis

```
namespace boost
{
namespace statechart
{
  template< class Event >
  class termination
  {
    // implementation-defined
  };
}
}
```

### Class template `termination` semantics

When executed, a call is made to the `terminate` member function of the state for which the reaction was defined.

# Header <boost/statechart/deferral.hpp>

## Class template `deferral`

This class template is used to specify a deferral reaction. Instantiations of this template can appear in the `reactions` member `typedef` in models of the SimpleState and State concepts.

### Class template `deferral` parameters

| Template parameter | Requirements | Semantics |
|---|---|---|
| `Event` | A model of the Event concept or the class `event_base` | The event triggering the deferral. If `event_base` is specified, the deferral is triggered by all models of the Event concept |

### Class template `deferral` synopsis

```
namespace boost
{
namespace statechart
{
  template< class Event >
  class deferral
  {
    // implementation-defined
  };
}
}
```

### Class template `deferral` semantics

When executed, a call is made to the `defer_event` member function of the state for which the reaction was defined.

# Header <boost/statechart/custom_reaction.hpp>

## Class template `custom_reaction`

This class template is used to specify a custom reaction. Instantiations of this template can appear in the `reactions` member `typedef` in models of the SimpleState and State concepts.

### Class template `custom_reaction` parameters

| Template parameter | Requirements | Semantics |
|---|---|---|
| `Event` | A model of the Event concept or the class `event_base` | The event triggering the custom reaction. If `event_base` is specified, the custom reaction is triggered by all models of the Event |

| | | |
|---|---|---|
| | | concept |

### Class template `custom_reaction` synopsis

```
namespace boost
{
namespace statechart
{
  template< class Event >
  class custom_reaction
  {
    // implementation-defined
  };
}
}
```

### Class template `custom_reaction` semantics

When executed, a call is made to the user-supplied `react` member function of the state for which the reaction was defined. The `react` member function must have the following signature:

```
result react( const Event & );
```

and must call exactly one of the following reaction functions and return the obtained `result` object:

```
result discard_event();
result forward_event();
result defer_event();
template< class DestinationState >
result transit();
template<
  class DestinationState,
  class TransitionContext,
  class Event >
result transit(
  void ( TransitionContext::* )( const Event & ),
  const Event & );
result terminate();
```

# Header <boost/statechart/result.hpp>

## Class `result`

Defines the nature of the reaction taken in a user-supplied `react` member function (called when a `custom_reaction` is executed). Objects of this type are always obtained by calling one of the reaction functions and must be returned from the `react` member function immediately.

```
namespace boost
{
```

```
namespace statechart
{
  class result
  {
    public:
      result( const result & other );
      ~result();

    private:
      // Result objects are not assignable
      result & operator=( const result & other );
  };
}
}
```

## Class `result` constructor and destructor

```
result( const result & other );
```

**Requires**: `other` is **not** consumed
**Effects**: Copy-constructs a new `result` object and marks `other` as consumed. That is, `result` has destructive copy semantics

```
~result();
```

**Requires**: `this` is marked as consumed
**Effects**: Destructs the result object

---

Revised 08 July, 2006