The boost::fsm library

Reference

Contents

Concepts

ExceptionTranslator concept

An ExceptionTranslator type defines how C++ exceptions occurring during state machine operation are translated to exception events. Every model of this concept must provide an `operator()` with the following signature:

```
template< class Action, class ExceptionEventHandler >
result operator()(
  Action action,
  ExceptionEventHandler eventHandler,
  result handlerSuccessResult );
```

For an ExceptionTranslator object `e` the following expression must be well-formed and have the indicated results:

Expression	Type	Effects/Result

`result action();`
`bool exceptEventHandler(`
`  const event_base & );`
`result`
`  handlerSuccessResult;`
		1. Attempts to execute `action` and to return the result
		2. Catches the exception propagated from `action`, if necessary
		3. Translates the exception to a suitable `event`

`e(`
`  &action,`
`  &exceptEventHandler,`
`  handlerSuccessResult );`
	`result`	subclass and constructs an object of the event
		4. Passes the event object to `exceptEventHandler`
		5. Rethrows the original exception if `exceptEventHandler` returns `false`, returns `handlerSuccessResult` otherwise

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 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. The result is undefined if this is not the case.

Worker concept

todo

Header <boost/fsm/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 subclass of this class template		

`InitialState`	A most-derived direct or indirect subclass of either the `simple_state` or the `state` class template. The type that this class passes as `Context` to its base class template must be equal to `MostDerived`. That is, `InitialState` must be an outermost state of this state machine	The state that is entered when `state_machine<>::initiate()` is called	
`Allocator`	A model of the standard Allocator concept		`std::allocator< void >`
`ExceptionTranslator`	A model of the ExceptionTranslator concept	see ExceptionTranslator concept	`exception_translator<>`

Class template `state_machine` synopsis

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

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

      bool 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;

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

Class template `state_machine` constructor and destructor

```
state_machine();
```

Effects: Constructs a non-running state machine
Postcondition: `terminated()`

```
~state_machine();
```

Effects: `terminate();`

Class template `state_machine` modifier functions

```
bool 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
3. Constructs a function object `exceptionEventHandler` with an `operator()` returning `bool` and accepting an exception event parameter that processes the passed exception event, with the following differences to the processing of normal events:
   - 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 process exception events)
   - If no reaction could be found for the exception event or if the state machine is not stable after processing the exception event, `false` is returned from the `exceptionEventHandler` function object. Otherwise, `true` is returned

4. Passes `action`, `exceptionEventHandler` and the `result` value `handlerSuccessResult` to `ExceptionTranslator::operator()`. If `ExceptionTranslator::operator()` throws an exception, `terminate()` is called and the exception is propagated to the caller. Continues with step 5 otherwise (the return value is discarded)
5. Processes all posted events (see `process_event`)

Returns: `terminated()`
Throws: Any exceptions propagated from `ExceptionTranslator::operator()`. Exceptions never originate in the library itself but only in code supplied through template parameters. That is, `std::bad_alloc` thrown by `Allocator::allocate` as well as any exceptions thrown by user-supplied `react` functions, transition-actions and entry-actions

```
void terminate();
```

Effects: The state machine exits (destructs) all currently active states. Innermost states are exited first. Other states are exited as soon as all their direct and indirect inner states have been exited
Postcondition: `terminated()`

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

Effects:

1. Selects the passed event as the current event
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 the current event, 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` with an `operator()` accepting an exception event parameter and returning `bool` that processes the passed exception event, with the following differences to the processing of normal events:
   - 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 process exception events)

    - If no reaction could be found for the exception event or if the state machine is not stable after processing the exception event, `false` is returned from the `exceptionEventHandler` function object. Otherwise, `true` is returned
6. Passes `action`, an `exceptionEventHandler` callback and the `fsm::result` value `handlerSuccessResult` to `ExceptionTranslator::operator()`. If `ExceptionTranslator::operator()` throws an exception then calls `terminate()` and propagates the exception to the caller
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 current event is stored in a state-specific queue. Continues with step 10
9. If `ExceptionTranslator::operator()` returns the previously passed `handlerSuccessResult` or if the return value is equal to the one of `simple_state::discard_event()` then continues with step 10
10. If the posted events queue is non-empty then dequeues the first event, selects it as the current event and continues with step 2. Returns to the caller otherwise

Returns: `false`, if the machine was terminated before processing the event. Returns `terminated()` otherwise
Throws: Any exceptions propagated from `ExceptionTranslator::operator()`. Exceptions never originate in the library itself but only in code supplied through template parameters. That is, `std::bad_alloc` thrown by `Allocator::allocate` as well as any exceptions thrown by user-supplied reactions, transition-actions and entry-actions

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 event dispatch

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

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 event dispatch

```
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 individual innermost states in the range is undefined. Their position may change with each call to a modifier function. Moreover, all iterators are invalidated when a modifier function is called

Header <boost/fsm/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 subclass of this class template		
`InitialState`	A most-derived direct or indirect subclass of either the `simple_state` or the `state` class template. The type that this class passes as `Context` to its base class template must be equal to `MostDerived`. That is, `InitialState` must be an outermost state of this state machine	The state that is entered when `Worker::operator()()` is called	
`Worker`	A model of the Worker concept	see Worker concept	`worker<>`
`Allocator`	A model of the standard Allocator concept		`std::allocator< void >`
	A model of the	see	

`ExceptionTranslator`	ExceptionTranslator concept	ExceptionTranslator concept	`exception_translator<>`

Class template `asynchronous_state_machine` synopsis

```
template<
  class MostDerived,
  class InitialState,
  class Worker = worker<>,
  class Allocator = std::allocator< void >,
  class ExceptionTranslator = exception_translator<> >
class asynchronous_state_machine : implementation-defined
{
  public:
    void queue_event( const intrusive_ptr< event_base > & );

  protected:
    asynchronous_state_machine( Worker & myWorker );
    ~asynchronous_state_machine();
};
```

Class template `asynchronous_state_machine` constructor and destructor

```
asynchronous_state_machine( Worker & myWorker );
```

Precondition: No thread of control is currently inside `myWorker.operator()`
Effects: Constructs a non-running asynchronous state machine and registers it with the passed worker
Throws: Whatever `Allocator::allocate` (invoked by the worker) throws

```
~asynchronous_state_machine();
```

Precondition: No thread of control is currently inside `myWorker.operator()`. The worker object passed to the constructor has not yet been destructed
Effects: Terminates the state machine

Class template `asynchronous_state_machine` modifier functions

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

Effects: Pushes the passed event into the queue of the worker object passed to the constructor
Throws: Whatever `Allocator::allocate` (invoked by the worker) throws

Header <boost/fsm/simple_state.hpp>

Typedef `no_reactions`

This is the default value for the `Reactions` parameter of the `simple_state` class template.

Necessary for the rare cases when a state without reactions has inner states.

```
namespace boost
{
namespace fsm
{
  typedef implementation-defined no_reactions;
}
}
```

Enum `history_mode`

Defines the history type of a state.

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

Class template `simple_state`

The base class template of all states that do **not** need to call any of the following `simple_state` member functions from their constructors:

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

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;
```

States that need to call any of these 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 subclass of this class template		
`Context`	A most-derived direct or indirect subclass of either the `state_machine`, `asynchronous_state_machine`, `simple_state` or `state` class templates or an instantiation of the `orthogonal` class template nested in the state base classes. Must be a complete type	Defines the states' position in the state hierarchy	
`Reactions`	An `mpl::list` containing instantiations of the `custom_reaction`, `deferral`, `termination` or `transition` class templates. If there is only a single reaction then it can also be passed directly, without wrapping it into an `mpl::list`	Defines to which events a state can react	`no_reactions`
`InnerInitial`	An `mpl::list` containing most-derived direct or indirect subclasses of either the `simple_state` or the `state` class template or instantiations of either the `shallow_history` or `deep_history` class templates. If there is only a single non-history inner initial state then it can also be passed directly, without wrapping it into an `mpl::list`. The type that each state in the list passes as `Context` to its base class template 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 fsm
{
  template<
```

```
  class MostDerived,
  class Context,
  class Reactions = no_reactions,
  class InnerInitial = unspecified,
  history_mode historyMode = has_no_history >
class simple_state : implementation-defined
{
  public:
    // 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;

    void post_event(
      const intrusive_ptr< 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();

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

Class template `simple_state` constructor and destructor

```
simple_state();
```

Effects: Depending on the `historyMode` parameter, reserves storage to store none, shallow, deep or both histories
Throws: Any exceptions propagated from `Allocator::allocate` (the template parameter passed to the base class of `outermost_context_type`)
Note: The constructors of all direct and indirect subclasses should be exception-neutral

```
virtual ~simple_state();
```

Effects: Depending on the `historyMode` parameter, stores none, shallow, deep or both histories. 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 > & );
```

Effects: Pushes the passed event into the state machine's posted events queue
Throws: Any exceptions propagated from `Allocator::allocate` (the template parameter passed to the base class of `outermost_context_type`)
Note: Unless the direct subclass is the `state` class template, this function must not be called from the constructors of direct and indirect subclasses. All direct and indirect callers should be exception-neutral

```
result discard_event();
```

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: An unspecified value of the `result` enumeration. The user-supplied `react` member function must return this value to its caller
Note: Must only be called from within `react` member functions, which are called by `custom_reaction` instantiations. All direct and indirect callers should be exception-neutral

```
result forward_event();
```

Effects: Instructs the state machine to forward the current event to the next state (see `state_machine::process_event` for details)

Returns: An unspecified value of the `result` enumeration. The user-supplied `react` member function must return this value to its caller
Note: Must only be called from within `react` member functions, which are called by `custom_reaction` instantiations. All direct and indirect callers should be exception-neutral

```
result defer_event();
```

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: An unspecified value of the `result` enumeration. The user-supplied `react` member function must return this value to its caller
Note: Must only be called from within `react` member functions, which are called by `custom_reaction` instantiations. All direct and indirect callers should be exception-neutral

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

Effects:

1. Exits (destructs) all currently active direct and indirect inner states of the innermost common outer state 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
2. Enters (constructs) the state that is both a direct inner state of the innermost common outer state 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` depth first
4. Enters (constructs) the tree formed by the direct and indirect inner initial states of `DestinationState` depth first
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: An unspecified value of the `result` enumeration. The user-supplied `react` member function must return this value to its caller
Throws: Any exceptions propagated from `operator new` (used to allocate states), state constructors or `Allocator::allocate` (the template parameter passed to the base class of `outermost_context_type`)
Note: Must only be called from within `react` member functions, which are called by `custom_reaction` instantiations. All direct and indirect callers should be exception-neutral
Caution: Inevitably exits (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 & );
```

Effects:

1. Exits (destructs) all currently active direct and indirect inner states of the innermost common outer state 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
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 outer state 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` depth first
5. Enters (constructs) the tree formed by the direct and indirect inner initial states of `DestinationState` depth first
6. 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: An unspecified value of the `result` enumeration. The user-supplied `react` member function must return this value to its caller
Throws: Any exceptions propagated from `operator new` (used to allocate states), state constructors, the transition action or `Allocator::allocate` (the template parameter passed to the base class of `outermost_context_type`)
Note: Must only be called from within `react` member functions, which are called by `custom_reaction` instantiations. All direct and indirect callers should be exception-neutral
Caution: Inevitably exits (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();
```

Effects: Terminates the state and 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: An unspecified value of the `result` enumeration. The user-supplied `react` member function must return this value to its caller
Note: Must only be called from within `react` member functions, which are called by `custom_reaction` instantiations. All direct and indirect callers should be exception-neutral
Caution: Inevitably exits (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

Class template `simple_state` observer functions

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

Returns: A reference to the outermost context, which is always the state machine this state belongs to
Note: Unless the direct subclass is the `state` class template, this function must not be called from the constructors of direct and indirect subclasses

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

Returns: A reference to the const outermost context, which is always the state machine this state belongs to
Note: Unless the direct subclass is the `state` class template, this function must not be called from

the constructors of direct and indirect subclasses

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

Returns: A reference to a direct or indirect context
Note: Unless the direct subclass is the `state` class template, this function must not be called from the constructors of direct and indirect subclasses

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

Returns: A reference to a const direct or indirect context
Note: Unless the direct subclass is the `state` class template, this function must not be called from the constructors of direct and indirect subclasses

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

Returns: Has exactly the same semantics as `state_machine::state_cast`
Note: Unless the direct subclass is the `state` class template, this function must not be called from the constructors of direct and indirect subclasses. The result is **unspecified** if this function is called when the machine is not stable

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

Returns: Has exactly the same semantics as `state_machine::state_downcast`
Note: Unless the direct subclass is the `state` class template, this function must not be called from the constructors of direct and indirect subclasses. The result is **unspecified** if this function is called when the machine is not stable

Class template `simple_state` static functions

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

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

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

Returns: The pointer to the custom type identifier for `MostDerived` or 0. If `!= 0`, `CustomId` must match the type of the previously set pointer. The result is undefined if this is not the case
Note: This function is not available if `BOOST_FSM_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_FSM_USE_NATIVE_RTTI` is defined

Header <boost/fsm/state.hpp>

Class template `state`

This is the base class template of all states that need to call any of the following `simple_state` member functions from their constructors:

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

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;
```

States that do not need to call any of these 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 fsm
{
  template<
    class MostDerived,
    class Context,
    class Reactions = no_reactions,
    class InnerInitial = unspecified,
    history_mode historyMode = has_no_history >
  class state : public simple_state<
    MostDerived, Context, Reactions, InnerInitial, historyMode >
  {
    protected:
      struct my_context
      {
        // implementation-defined
      };

      typedef state my_base;

      state( my_context ctx );
```

```
        virtual ~state();
    };
}
}
```

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

Revised 12 December, 2003

Copyright © 2003 Andreas Huber Dönni. All Rights Reserved.