The boost::fsm library

Rationale

Introduction

Most of the design decisions made during the development of this library are the result of the following requirements.

boost::fsm should ...

1. be fully type-safe. Whenever possible, type mismatches should be flagged with an error at compile-time
2. not require the use of a code generator. A lot of the existing FSM solutions force the developer to design the state machine either graphically or in a specialized language. All or part of the code is then generated
3. allow for easy transformation of a UML statechart (defined in http://www.omg.org/cgi-bin/doc?formal/03-03-01) into a working state machine. Vice versa, an existing C++ implementation of a state machine should be fairly trivial to transform into a UML statechart. Specifically, the following state machine features should be supported:
   - Hierarchical (composite, nested) states
   - Orthogonal (concurrent) states
   - Entry-, exit- and transition-actions
   - Guards
   - Shallow/deep history
4. produce a customizable reaction when a C++ exception is propagated from user code
5. support synchronous and asynchronous state machines and leave it to the user which thread an asynchronous state machine will run in. Users should also be able to use the threading library of their choice
6. support the development of arbitrarily large and complex state machines. Multiple developers should be able to work on the same state machine simultaneously
7. allow the user to customize all resource management so that the library could be used for applications with hard real-time requirements
8. enforce as much as possible at compile time. Specifically, invalid state machines should not compile
9. offer reasonable performance for a wide range of applications

Why yet another state machine framework?

Before I started to develop this library I had a look at the following frameworks:

- The framework accompanying the book "Practical Statecharts in C/C++" by Miro Samek, CMP Books, ISBN: 1-57820-110-1
  http://www.quantum-leaps.com
  Fails to satisfy at least the requirements 1, 3, 4, 6, 8.
- The framework accompanying "Rhapsody in C++" by ILogix (a code generator solution)
  http://www.ilogix.com/products/rhapsody/rhap_incplus.cfm
  This might look like comparing apples with oranges. However, there is no inherent reason why a code generator couldn't produce code that can easily be understood and modified by humans. Fails to satisfy at least the requirements 2, 4, 5, 6, 8 (there is quite a bit of error checking before code generation, though).
- The framework accompanying the article "State Machine Design in C++"
  http://www.cuj.com/articles/2000/0005/0005f/0005f.htm?topic=articles
  Fails to satisfy at least the requirements 1, 3, 4, 5 (there is no direct threading support), 6, 8.

I believe boost::fsm satisfies all requirements.

State-local storage

This not yet widely known state machine feature is enabled by the fact that every state is represented by a class. Upon state-entry, an object of the class is constructed and the object is later destructed when the state machine exits the state. Any data that is useful only as long as the machine resides in the state can (and should) thus be a member of the state. This feature paired with the ability to spread a state machine over several translation units makes possible virtually unlimited scalability.

In most existing FSM frameworks the whole state machine runs in one environment (context). That is, all resource handles and variables local to the state machine are stored in one place (normally as members of the class that also derives from some state machine base class). For large state machines this often leads to the class having a huge number of data members most of which are needed only briefly in a tiny part of the machine. The state machine class therefore often becomes a change hotspot what leads to frequent recompilations of the whole state machine.

Dynamic configurability

Two types of state machine frameworks

- A state machine framework supports dynamic configurability if the whole layout of a state machine can be defined at runtime ("layout" refers to states and transitions, actions are still specified with normal C++ code). That is, data only available at runtime can be used to build arbitrarily large machines. See "A Multiple Substring Search Algorithm" by Moishe Halibard and Moshe Rubin in June 2002 issue of CUJ for a good example (unfortunately not available online).
- On the other side are state machine frameworks which require the layout to be specified at compile time.

State machines that are built at runtime almost always get away with a simple state model (no hierarchical states, no orthogonal states, no entry and exit actions, no history) because the layout is very often **computed by an algorithm**. On the other hand, machine layouts that are fixed at compile time are almost always designed by humans, who frequently need/want a sophisticated state model in order to keep the complexity at acceptable levels. Dynamically configurable FSM frameworks are therefore often optimized for simple flat machines while incarnations of the static variant tend to offer more features for abstraction.

However, fully-featured dynamic FSM libraries do exist. So, the question is:

Why not use a dynamically configurable FSM library for all state machines?

One might argue that a dynamically configurable FSM framework is all one ever needs because **any** state machine can be implemented with it. However, due to its nature such a framework has a number of disadvantages when used to implement static machines:

- No compile-time optimizations and validations can be made. For example, boost::fsm determines the innermost common outer state (aka LCA, least common ancestor) of the transition-source and destination state at compile time. Moreover, compile time checks ensure that the state machine is valid (e.g. that there are no transitions between orthogonal states).
- Double dispatch must inevitably be implemented with some kind of a table. As argued under Double dispatch, this scales badly.
- To warrant fast table lookup, states and events must be represented with an integer. To keep the table as small as possible, the numbering should be continuous, e.g. if there are ten states, it's best to use the ids 0-9. To ensure continuity of ids, all states are best defined in the same header file. The same applies for the events. Again, this does not scale.
- Because events carrying parameters are not represented by a type, some sort of a generic event with a property map must be used and type-safety is enforced at runtime rather than at compile time.

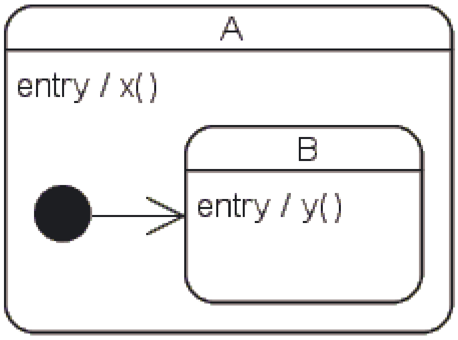
It is for these reasons, that boost::fsm was built from ground up to **not** support dynamic configurability. However, this does not mean that it's impossible to dynamically shape a machine implemented with this library. For example, guards can be used to make different transitions depending on input only available at runtime. However, such layout changes will always be limited to what can be foreseen before compilation. A somewhat related library, the boost::spirit parser framework, allows for roughly the same runtime configurability.

Error handling

There is not a single word about error handling in the UML state machine semantics specifications. Moreover, most existing FSM solutions also seem to ignore the issue.

Why an FSM library should support error handling

Consider the following state configuration:



Both states define entry actions (x() and y()). Whenever state A becomes current, a call to x() will immediately be followed by a call to y(). y() could depend on the side-effects of x(). Therefore, executing y() does not make sense if x() fails. This is not an esoteric corner case but happens in every-day state machines all the time. For example, x() could acquire memory the contents of which is later modified by y(). There is a different but in terms of error handling equally critical situation in the Tutorial under Getting state information out of the machine when `Running::~Running` accesses its outer state `Active`. Had the entry action of `Active` failed and had `Running` been entered anyway then `Running`'s exit action would have invoked undefined behavior.
The error handling situation with outer and inner states resembles the one with base and derived classes: If a base class constructor fails (by throwing an exception) the construction is aborted, the derived class constructor is not called and the object never comes to life.

If an FSM framework does not account for failing actions, the user is forced to adopt cumbersome workarounds. For example, a failing action would have to post an appropriate error event and set a global error variable to true. Every following action would first have to check the error variable before doing anything. After all actions have completed (by doing nothing!), the previously posted error event would have to be processed what would lead to the remedy action being executed. Please note that it is not sufficient to simply queue the error event as other events could still be pending. Instead, the error event has absolute priority and would have to be dealt with immediately.

So, to be safe, programmers would have to encapsulate the code of **every** action in `if ( ! error ) { /* action */ }` blocks. Moreover, a `try { /* action */ } catch ( ... ) { /* post error event */ error = true; }` statement would often have to be added because called functions might throw and letting an exception propagate out of a user action would at best terminate the state machine immediately. Writing all this boiler-plate code is simply boring and quite unnecessary.

Error handling support in boost::fsm

- C++ exceptions are used for all error handling. Except from exit-actions (mapped to state-destructors and exceptions should almost never be propagated from destructors), exceptions can be propagated from all user functions.
- A customizable per state machine policy specifies how to convert all exceptions propagated from user code. Out of the box, an `exception_thrown` event is generated.
- An exception event is always processed immediately and thus has absolute priority over any possibly pending events. The event queue stays as it was until the exception event has been processed.
- The processing logic is as follows:
    - Exception events resulting from failed `react` functions are sent to the current state.
    - Exception events resulting from failed entry actions are sent to the immediate outer state.
    - Exception events resulting from failed transition actions are sent to the innermost common outer state.

In the last two cases the state-machine is not in a stable state when the exception event is generated and leaving it there (e.g. by ignoring the exception event) would violate an invariant of state machines. So, the exception event reaction must either be a transition or a termination to bring the machine back into a stable state. That's why the framework checks that the state machine is stable after processing an exception event. If this is not the case the state machine is terminated and the exception is rethrown.

Asynchronous state machines

The design of the `asynchronous_state_machine<>` and `worker<>` class templates follow from the requirements:

1. The user must be able to specify in which thread a particular machine will run:
   The `worker<>` class template is not associated with a particular thread. Instead, users can choose to either call `worker<>::operator()` directly from the current thread or pass an appropriate function object to a new thread.
2. An arbitrary number of state machines might run in the same thread:
   Multiple state machine objects can be constructed passing the same `worker<>` object. The state machines will then share the same thread-safe queue and event loop.
3. Out of the box, the boost::thread library should be employed. However, it should be possible to use any other threading library or run asynchronous machines on OS-less systems:
   In such cases, the locking and waiting logic can be fully customized by implementing a new class template that is interface-compatible with `worker<>`.

User actions: Member functions vs. function objects

All user-supplied functions (`react` member functions, entry-, exit- and transition-actions) must be class members. The reasons for this are as follows:

- The concept of state-local storage mandates that state-entry and state-exit actions (mapped to constructors and destructors) are implemented as members.
- `react` member functions and transition actions often access state-local data. So, it is most natural to implement these functions as members of the class the data of which the functions will operate on anyway.

Speed versus scalability tradeoffs

Quite a bit of effort has gone into making the library scaleable **and** keeping it fast for small simple machines. While I believe it should perform reasonably in most applications, the scalability does not come for free. Small, carefully handcrafted state machines will thus easily outperform equivalent boost::fsm machines. To get a picture of how big the gap is I implemented a simple benchmark in the BitMachine example. The Handcrafted example is a handcrafted variant of the 1-bit-BitMachine implementing the same benchmark.

I tried to create a fair but somewhat unrealistic **worst-case** scenario:

- For both machines exactly one object of the only event is allocated before starting the test. This same object is then sent to the machines over and over.
- The Handcrafted machine employs GOF-visitor double dispatch. The states are preallocated so that event dispatch & transition amounts to nothing more than two virtual calls and one pointer assignment.

The Benchmarks - compiled with MSVC7.1 (single threaded), running on an Intel Pentium M 1600 - produced the following results:

- Handcrafted: 10 nanoseconds to dispatch one event and make the resulting transition.
- 1-bit-BitMachine with customized memory management: 210 nanoseconds to dispatch one event and make the resulting transition.

Although this is a big difference I still think it will not be noticeable in most real-world applications. No matter whether an application uses handcrafted or boost::fsm machines it will...

- almost never run into a situation where a state machine is swamped with as many events as in the benchmarks. Unless a state machine is abused for parsing, it will typically spend a good deal of time waiting for events
- often run state machines in their own threads. This adds considerable locking and task-switching overhead. Performance tests with the PingPong example, where two asynchronous state machines exchange events, gave the following times to process one event and perform the resulting transition (using the library out-of-the-box):
    - Single-threaded (no locking and waiting): 1000ns
    - Multi-threaded with one worker thread (the worker uses mutex locking but never has to wait for events): 3700ns
    - Multi-threaded with two worker threads (both workers use mutex locking and exactly one worker always waits for an event): 6400ns

  Handcrafted machines will also pay the 2700ns/5400ns per event multithreading overhead, making raw dispatch and transition speed much less important
- almost always allocate events with `new` and destroy them after consumption. This will add a few cycles, even if event memory management is customized
- often use state machines that employ orthogonal states and other advanced features. This forces the handcrafted machines to use a more adequate and more time-consuming book-keeping

Therefore, in real-world applications event dispatch and transition not normally constitutes a bottleneck and the relative gap between handcrafted and boost::fsm machines also becomes much smaller than in the worst-case scenario.

BitMachine measurements with more states and with different levels of optimization:

Machine configuration # states / # outgoing transitions per state	Event dispatch & transition time [nanoseconds]		
	Out of the box	Same as out of the box but with `BOOST_FSM_USE_NATIVE_RTTI` defined	Same as out of the box but with customized memory management
2 / 1	680	790	210
4 / 2	690	850	210
8 / 3	690	910	220
16 / 4	710	990	230
32 / 5	740	1090	240
64 / 6	820	1250	310

Possible optimizations

Currently, `std::list`s are used for event and state storage. These could be replaced with an intrusive linked list container what would eliminate 50% of the `operator new` and `operator delete` calls made during an event dispatch & transition cycle of the smallest BitMachine. I would guess that this could speed it up by 25%-50%. However, dispatch time is not affected and can quickly consume considerable time, as the 6-bit-BitMachine shows. Moreover, most states of real-world machines are quite deeply nested and the average transition involves the deallocation and allocation of 2 states. Since std::list allocations occur only once per transition and orthogonal region, the relative performance gain of this optimization becomes much smaller for typical machines and does not seem to be worth the effort of hand-crafting an intrusive linked list.

Memory management customization

Out of the box, all internal data is allocated on the normal heap. This should be satisfactory for applications where all the following prerequisites are met:

- There are no deterministic reaction time (hard real-time) requirements.
- The application will typically not process more than a few events per second. Of course, this figure depends on your platform. A typical desktop PC could easily cope with more than 100000 events per second.
- The application will never run long enough for heap fragmentation to become a problem. This is of course an issue for all long running programs not only the ones employing this library. However, it should be noted that fragmentation problems could show up earlier than with traditional FSM frameworks.

Should a system not meet any of these prerequisites customization of all memory management (not just boost::fsm's) should be considered, which is supported as follows:

- By passing a class offering a `std::allocator` interface for the `Allocator` parameter of the `state_machine` class template. The `rebind` member template is used to customize memory allocation of the internal containers.
- By replacing the `simple_state`, `state` and `event` class templates with ones that have a customized `operator new` and `operator delete`. This can be as easy as inheriting your customized class templates from the framework-supplied class templates **and** your preferred small-object/deterministic/constant-time allocator base class.

`simple_state` and `state` subclass objects are constructed and destructed only by the state machine. It would therefore be possible to use the `state_machine` allocator instead of forcing the user to overload `operator new` and `operator delete`. However, a lot of systems employ at most one instance of a particular state machine, which means that a) there is at most one object of a particular state and b) this object is always constructed, accessed and destructed by one and the same thread. We can exploit these facts in a much simpler (and faster) `new`/`delete` implementation (for example, see UniqueObject.hpp in the BitMachine example). However, this is only possible as long as we have the freedom to customize memory management for state classes separately.

RTTI customization

RTTI is used for event dispatch and `state_downcast<>()`. Currently, there are exactly two options:

1. By default, a speed-optimized internal implementation is employed
2. The library can be instructed to use native C++ RTTI instead by defining

```
BOOST_FSM_USE_NATIVE_RTTI
```

Just about the only reason to favor 2 is the fact that state objects need to store one pointer less, meaning that in the best case the memory footprint of a state machine object could shrink by 15%. However, on most platforms executable size also grows considerably when C++ RTTI is turned on. So, given the small per machine object savings, option 2 only makes sense in applications that ...

- are not processor-bound, and
- need to cut down on overall memory footprint (executable size plus memory allocated at runtime), and
- need to simultaneously run a large number of identical state machines

Double dispatch

At the heart of every state machine lies an implementation of double dispatch. This is due to the fact that the incoming event **and** the current state define exactly which reaction the state machine will produce. For each event dispatch, one virtual call is followed by a linear search for the appropriate reaction, using one RTTI comparison per reaction. The following alternatives were considered but rejected:

- Acyclic visitor: This double-dispatch variant satisfies all scalability requirements but performs badly due to costly inheritance tree cross-casts. Moreover, a state must store one v-pointer for **each** reaction what slows down construction and makes memory management customization inefficient. In addition, C++ RTTI must inevitably be turned on, with negative effects on executable size. boost::fsm originally employed acyclic visitor and was about 4 times slower than it is now (MSVC7.1 on Intel Pentium M). The dispatch speed might be better on other platforms but the other negative effects will remain.
- GOF Visitor: The GOF Visitor pattern inevitably makes the whole machine depend upon all events. That is, whenever a new event is added there is no way around recompiling the whole state machine. This is contrary to the scalability requirements.
- Two-dimensional array of function pointers: To satisfy requirement 6, it should be possible to spread a single state machine over several translation units. This however means that the dispatch table must be filled at runtime and the different translation units must somehow make themselves "known", so that their part of the state machine can be added to the table. There simply is no way to do this automatically **and** portably. The only portable way that a state machine distributed over several translation units could employ table-based double dispatch relies on the user. The programmer(s) would somehow have to **manually** tie together the various pieces of the state machine. Not only does this scale badly but is also quite error-prone.

Resource usage

Memory

On a 32-bit box, one empty state typically needs less than 50 bytes of memory. Even **very** complex machines will usually have less than 20 simultaneously current states so just about every machine should run with less than one kilobyte of memory (not counting event queues). Obviously, the per-machine memory footprint is offset by whatever state-local members the user adds.

Processor cycles

The following ranking should give a rough picture of what feature will consume how many cycles:

1. `state_cast<>`: By far the most cycle-consuming feature. Searches linearly for a suitable state, using one `dynamic_cast` per visited state.
2. State entry and exit: Profiling of the fully optimized 1-bit-BitMachine suggested that about 100ns of the 210ns total are spent destructing the exited state and constructing the entered state. Obviously, transitions where the innermost common outer state is "far" from the leaf states and/or with lots of orthogonal states can easily cause the destruction and construction of quite a few states leading to significant amounts of time spent for a transition.
3. `state_downcast<>`: Searches linearly for the requested state, using one virtual call and one RTTI comparison per visited state.
4. History: For a state containing a history pseudo state a binary search through the (usually small) history map must be performed on each entry and exit. History slot allocation is performed exactly once, before first entry.
5. Event dispatch: One virtual call followed by a linear search for a suitable reaction, using one RTTI comparison per visited reaction.
6. Orthogonal states: One additional virtual call for each exited state **if** there is more than one current leaf state before a transition. It should also be noted that the worst-case event dispatch time is multiplied in the presence of orthogonal states. For example, if two orthogonal leaf states are added to a given current state configuration, the worst-case time is tripled.

Limitations

Deferring and posting events

For performance reasons and because synchronous state machines often do not need to queue events, it is possible to operate such machines entirely with stack-allocated events. However, as soon as events need to be deferred and/or posted there is no way around queuing and allocation with `new`. The interface of `simple_state::post_event` enforces the use of `boost::intrusive_ptr` at compile time. But there is no way to do the same for deferred events because allocation and deferral happen in completely unrelated places. Of course, a "wrongly" allocated event could easily be transformed into one allocated with `new` and pointed to by `boost::intrusive_ptr` with a virtual `clone()` function. However, in my experience, event deferral is needed only very rarely in synchronous state machines and the asynchronous variant enforces the use of `boost::intrusive_ptr` anyway. So, most users won't run into this limitation and I rejected the `clone()` idea because it could cause inefficiencies casual users wouldn't be aware of. In addition, users not needing event deferral would nevertheless pay with increased code size.

Junction points

UML junction points are not supported because arbitrarily complex guard expressions can easily be implemented with `custom_reaction`s.

Dynamic choice points

Currently there is no direct support for this UML element because its behavior can often be implemented with `custom_reaction`s. In rare cases this is not possible, namely when a choice point happens to be the initial state. Then, the behavior can easily be implemented as follows:

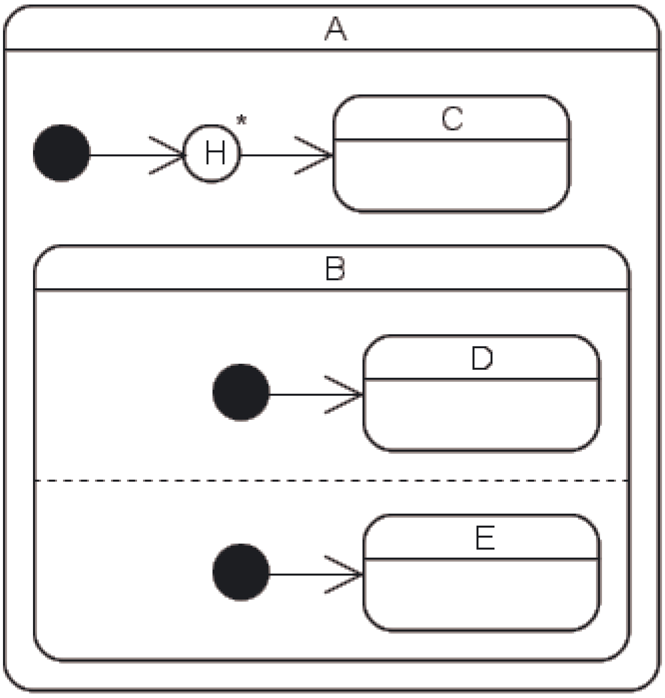
```
struct make_choice : fsm::event< make_choice > {};

// universal choice point base class template
template< class MostDerived, class Context >
```

```
struct choice_point : fsm::state< MostDerived, Context,
  fsm::custom_reaction< make_choice > >
{
  typedef fsm::state< MostDerived, Context,
    fsm::custom_reaction< make_choice > > base_type;
  typedef typename base_type::my_context my_context;
  typedef choice_point my_base;

  choice_point( my_context ctx ) : base_type( ctx )
  {
    base_type::post_event(
      boost::intrusive_ptr< make_choice >( new make_choice() ) )
  }
};

// ...

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

struct Destination1;
struct Destination2;
struct Destination3;
struct MyChoicePoint : choice_point< MyChoicePoint, Machine >
{
  MyChoicePoint( my_context ctx ) : my_base( ctx ) {}

  fsm::result react( const make_choice & )
  {
    if ( /* ... */ )
    {
      return transit< Destination1 >();
    }
    else if ( /* ... */ )
    {
      return transit< Destination2 >();
    }
    else
    {
      return transit< Destination3 >();
    }
  }
};
```

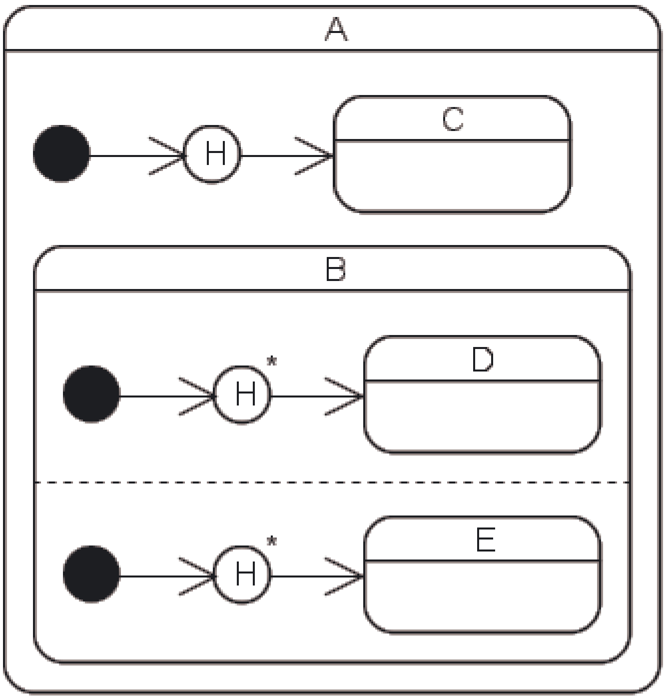
`choice_point` is not currently part of boost::fsm, mainly because I fear that beginners could use it in places where they would be better off with `custom_reaction`. If the demand is high enough I will add it to the library.

Deep history of orthogonal regions

Deep history of states with orthogonal regions is currently not supported:

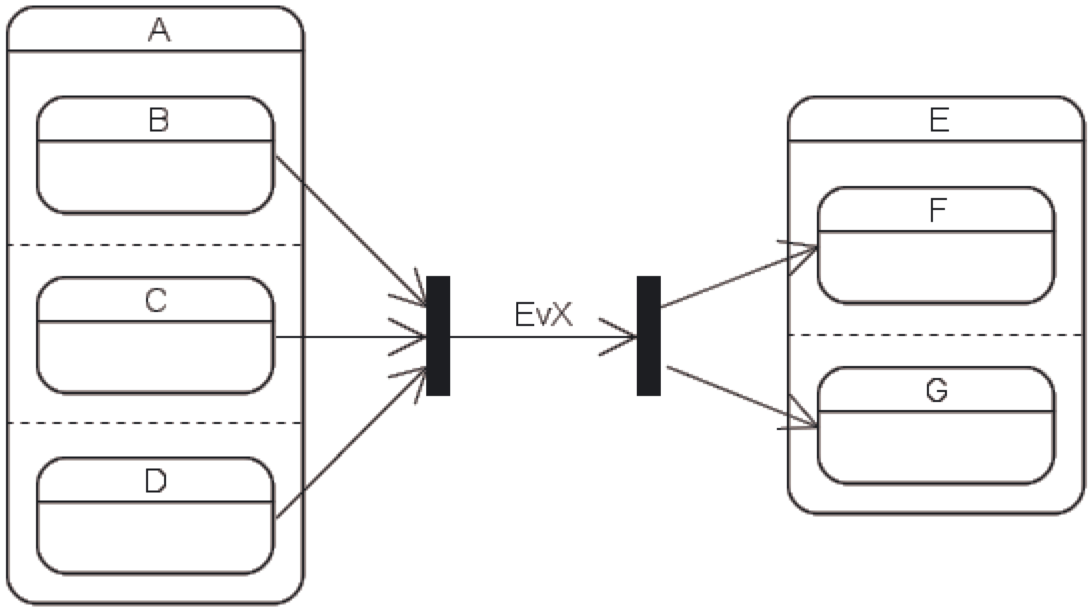


Attempts to implement this state chart will lead to a compile-time error because B has orthogonal regions and its direct or indirect outer state contains a deep history pseudo state. In other words, a state containing a deep history pseudo state must not have any direct or indirect inner states which themselves have orthogonal regions. This limitation stems from the fact that full deep history support would be more complicated to implement and would consume more resources than the currently implemented limited deep history support. Moreover, full deep history behavior can easily be implemented with shallow history:



Of course, this only works if C, D, E or any of their direct or indirect inner states do not have orthogonal regions. If not so then this pattern has to be applied recursively.

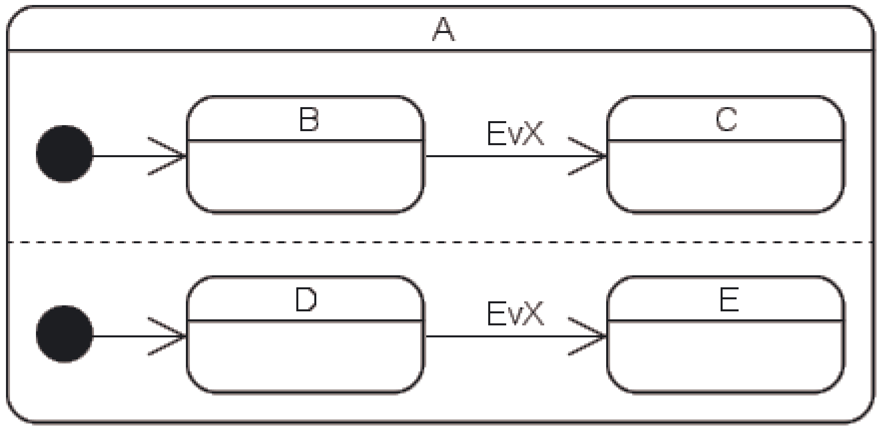
Synchronization (join and fork) bars



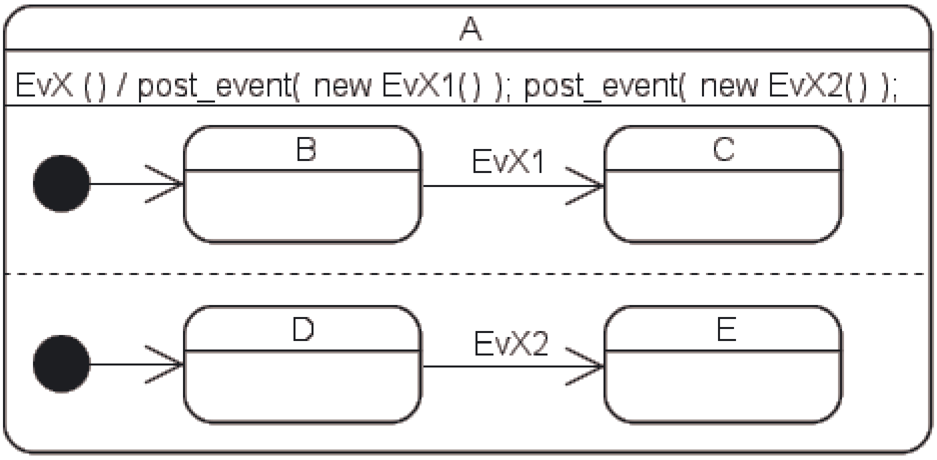
Synchronization bars are not supported, that is, a transition always originates at exactly one state and always ends at exactly one state. In my experience join bars are sometimes useful but their behavior can easily be emulated with guards. Fork bars are needed only rarely. Their support would complicate the implementation quite a bit.

Event dispatch to orthogonal regions

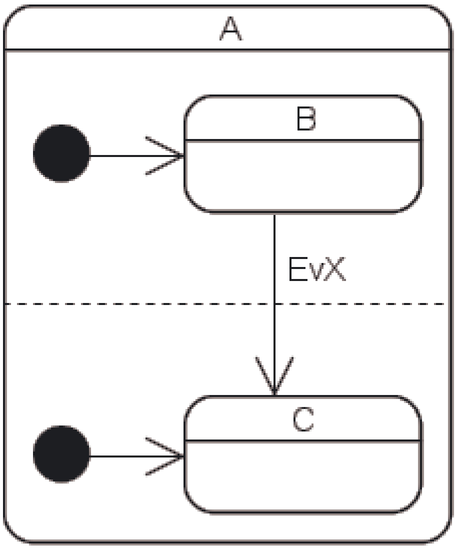
The boost::fsm event dispatch algorithm is different to the one specified in David Harel's original paper and in the UML standard. Both mandate that each event is dispatched to all orthogonal regions of a state machine. Example:



Here the Harel/UML dispatch algorithm specifies that the machine must transition from (B,D) to (C,E) when an EvX event is processed. Because of the subtleties that Harel describes in chapter 7 of his paper, an implementation of this algorithm is not only quite complex but also much slower than the simplified version employed by boost::fsm, which stops searching for reactions as soon as it has found one suitable for the current event. That is, had the example been implemented with this library, the machine would have transitioned non-deterministically from (B,D) to either (C,D) or (B,E). This version was chosen because, in my experience, in real-world machines different orthogonal regions often do not specify transitions for the same events. For the rare cases when they do, the UML behavior can easily be emulated as follows:



Transitions across orthogonal regions



Such transitions are currently flagged with an error at compile time (the UML specifications explicitly allow them while Harel does not mention them at all). I decided to not support them because I have erroneously tried to implement such a transition several times but have never come across a situation where it would make any sense. If you need to make such transitions, please do let me know!

Revised 17 November, 2003