state machine design doc

Document #:	state machine v1
Date:	2022-02-04
Project:	Programming Language C++
Audience:	dev team

Reply-to:	Marco Rubini <malorubi@outlook.com>

Contents

1 Introduction

We want to design the architecture of a state machine library component that satisfies the following requirements:

— custom states and transitions

— custom events

— timeout events

— guard conditions on transitions

— extended states

(Extended states are associated to one or more variables, which define quantitative aspects of the state)

— state hierarchies

(Refer to UML statechart hierarchical design)

— state entry and exit actions

— internal transitions

*(Internal transitions allow handling an event without changing state, and without executing **on-entry** and **on-exit** actions)*

The following features could be considered:

— event hierarchies and polymorphic event handlers

— event deferral

(In a hierarchical state machine, allow a substate to block/defer the handling of a specific event by a superstate.)

— anonymous transitions *(transitions that are triggered immediately after* on-entry*, without any associated event)*

— error handling

(Custom actions performed when no transition can handle an event, or an exception escapes from an user method)

— recovery from errors

— heartbeat events

(Like timeout events, but repeated after a fixed interval, for a fixed number of beats)

2 Competing designs

2.1 Variant-based approach

2.1.1 Context

A state machine context is a property of a state machine visible to all states, guards and transitions.

It is passed by value as first parameter to all methods of states, guards and transitions.

It has reference semantics, therefore modifications are visible to all entities.

It allows firing events. Events fired will be handled by an event queue.

It allows accessing the current state using `is` and `current_state` member templates.

```
namespace forest::sm
{
  // Context concepts
  template<class T>
  concept Context = requires(T context)
  {
    requires std::copy_constructible<T>;

    // models forest::StateMachine
    typename T::state_machine_type;

    { context.fire_event( /*any-event*/ ) } -> std::same_as<void>;
    { context.is</*any-state*/>() } -> std::boolean_testable;
    { context.current_state</*any-state*/>() } -> std::same_as</*any-state-reference*/>;
  };
}
```

2.1.2 State

A state is any **std::move-constructible** user-defined class.

A state can optionally define a `on_entry` member method, which implements the **on-entry** action.

A state can optionally define a `on_exit` member method, which implements the **on-exit** action.

```
namespace forest::sm
{
  // State concepts
  template<class T>
  concept State = requires(T state)
  {
    requires std::move_constructible<T>;
  };

  // State traits

  template<class T, class Context>
  concept StateWithEntryAction = forest::State<T> && requires(T state, Context context)
  {
    state.on_entry(context);
  };

  template<class T, class Context>
  concept StateWithExitAction = forest::State<T> && requires(T state, Context context)
  {
    state.on_exit(context);
  };
}

// Examples
struct state_disconnected
{
  template<Context Ctx>
  void on_entry(Ctx context);

  template<Context Ctx>
  void on_exit(Ctx context);
};

struct state_connected
{
  int ip_address;
};
```

(todo: consider exposing events to `on_entry` *and* `on_exit` *methods, since they are always invoked after a transition occurs.)*

2.1.3 Event

An event is a **std::move-constructible** user defined class.

```
namespace forest::sm
{
  // Event concept
  template<class T>
  concept Event = requires (T event)
  {
    requires std::move_constructible<T>;
```

```
  };
}

// Examples
struct event_connect
{
  int ip_address;
};
```

2.1.4 Guard

A guard is a predicate invocable given a context, a state and an event.

```
namespace forest::sm
{
  // Guard concept
  template<class T, class Context, class State, class Event>
  concept Guard = requires(T guard, Context context, State state, Event event)
  {
    requires forest::Context<Context>;
    requires forest::State<State>;
    requires forest::Event<Event>;

    { guard(context, state, event) } -> std::boolean_testable;
  };
}
```

2.1.5 Transition

A transition is invocable given a context, a state and an event.

The result of a transition is the state assigned to the state machine when the transition ends.

If a transition does not return anything, the transition is considered internal, therefore it does not change the state nor trigger **on-enter** or **on-exit** actions.

A Transition can optionally have a Guard, implemented by an accepts member method.

A Transition with no guard accepts all events for which it can be invoked.

```
namespace forest::sm
{
  template<class T, class Context, class State, class Event>
  concept Transition = requires(T transition, Context context, State state, Event event)
  {
    requires forest::Context<Context>;
    requires forest::State<State>;
    requires forest::Event<Event>;
    requires std::copy_constructible<T>;

    { transition(context, state, event) } -> /*State-or-void*/;
  };
```

```
  template<class T, class Context, class State, class Event>
  concept GuardedTransition = Transition<T, Context, State, Event>
    && requires(T transition, Context context, State state, Event event)
  {
    { transition.accepts(context, state, event) } -> std::boolean_testable;
  };
}

// Examples
struct transition_connect
{
  template<Context Ctx>
  bool accepts(Ctx context, state_disconnected& state, event_connect event) const
  {
    return true;
  }

  template<Context Ctx>
  state_connected operator()(Ctx context, state_disconnected& state, event_connect event) const
  {
    return state_connected {.ip_address = event.ip_address};
  }
};
```

2.1.6 Transition table

A transition table is a collection of transitions.

```
namespace forest::sm
{
  template<std::move_constructible... Transitions>
  using transition_table = std::tuple<Transitions...>;
}
```

2.1.7 State machine

Given a transition table and a starting state, a state machine can process events and evolve its internal state.

It exposes a `context_type` member typedef, the type passed as parameter to all states, transitions and guards.

It exposes a `initial_state` member typedef, the state entered by the state machine after initialization.

It exposes a `state_type` member typedef, a variant-like type that can store any state.

It exposes a `process_event` member method template that can handle an event.

It exposes a `current_state` member method template, which can be used to access the internal state. If `current_state` is invoked with a template parameter different from the current state, `std::bad_cast` is thrown.

It exposes a `is` member method template, which can be used to query the type of the internal state.

It exposes a `start` member method, which completes initialization enter enters the initial state.

It exposes a `stop` member method, which transitions to a `terminate` pseudo-state.

```
namespace forest::sm
{
  template<class T>
```

```
  concept StateMachine = requires(T state_machine)
  {
    typename T::context_type;
    typename T::transition_table;
    typename T::initial_state;
    typename T::state_type;

    { state_machine.current_state</*some-state*/>() } -> std::same_as</*some-state-ref*/>;
    { state_machine.is</*some-state*/>() } -> std::same_as<bool>;
    { state_machine.process_event(/*some-event*/) } -> std::same_as<void>;
    { state_machine.start(/*optional-event*/) } -> std::same_as<void>;
    { state_machine.stop(/*optional-event*/); } -> std::same_as<void>;
  };
}
```

2.1.8 State machine control flow

After initialization, the state machine is in a special `initialization_state`, which is implementation defined. After the method `start` is invoked, a special `initialization_event` is fired and triggers an implementation defined transition that transitions to the user defined `initial_state`, constructed with no arguments.

This behaviour can be customized by providing an event to the start method, and a transition from the initialization state handling it.

A state machine's internal state evolves after invoking `process_event` with any event, as described by the transition table.

Invoking stop fires a `stop_event`, which triggers an implementation defined transition that transitions unconditionally to `stop_state`. Any further event processing is impossible from `stop_state`.

This behaviour can be customized by providing a transition handling the `stop_event`.

2.1.9 Timeout events

A timeout event allows exiting a state after a fixed amount of time, starting from the moment the state's **on-entry** action is executed.

2.1.9.1 Proposal 1

Timeouts can be handled like any other event, but we need a way to schedule the firing of the event in the future from the state's **on-entry** action.

Therefore the state machine's context exposes an overload of `fire_event` with the following signature:

```
// Fires an event in the future
template<Event E>
void fire_event(E event, std::chrono::milliseconds ms);
```

The problem of this proposal is that the event will be fired unconditionally, even if the machine's state has changed. It can be hard to track if the timeout event needs to be handled or not.

2.1.9.2 Proposal 2

Another approach is to define a subtype of State called TimedState, that expose a `timeout_duration` method.

```
namespace forest::sm
{
  template<class T>
  concept TimedState = forest::State<T> && requires(T state)
```

```
  {
    { state.timeout_duration() } -> std::same_as<std::chrono::milliseconds>;
  };
}
```

Right before a TimedState **on-entry** action is executed, the state machine starts a timer.

The timer is accessible from the context variable using two methods, `entry_time` and `entry_time_elapsed`.

The timer can also be reset from any guard or transition using `reset_timer`.

After `timeout_duration` milliseconds have elapsed without any transition triggering a change of state, an event of type `forest::timeout_event` is fired, and can be handled with traditional transitions and guards.

2.1.10 State hierarchies

State hierarchies can be modelled with single inheritance.

```
struct super_state
{
  void on_entry();
  void on_exit();
};

struct child_state : super_state
{
  int value;

  void on_entry();
  void on_exit();
};
```

A transition that accepts a `super_state&` parameter can also be invoked from any state that inherits publicly from `super_state`.

```
struct other_state
{};

struct some_event
{};

// Transition that can be invoked from any state child of `super_state`
struct super_transition
{
  template<forest::Context Ctx>
  other_state operator()(Ctx context, super_state& state, some_event event)
  {
    if (context.is<child_state>()) {
      return other_state{static_cast<child_state&>(state).value};
    } else {
      return other_state{0};
    }
  }
};
```

The transition has access to all properties of `super_state`, and can use the context's `is` and `current_state` methods to inspect the state's dynamic type. This allows safe downcasting.