UML Hierarchical State Machine OOP Framework in C

Note: The code examples below are from a sample implementation of a hierarchical state machine. The entire sample state machine implementation is shown in the file fsm_example.c.

See **example-state-machine.pdf** in this folder for the state diagram.

1 Introduction

The framework defines FSM, State, and Event classes in C to implement UML Hierarchical State Machines. The framework is implemented in the files fsm.c and fsm.h. The header file exposes the framework API. It would be helpful to have both files open to refer to while reading this document.

The class attributes are standard C structs. Class methods are just C functions. Virtual functions are just function pointers in structs, typically initialized to default functions (or NULL for pure virtual functions).

```
Classes: Fsm, FsmState, FsmEvent
Fsm         - State Machine class
FsmState    - State class
FsmEvent    - Event class
```

Derive subclass structures by including the Base Class structure at the top of derived structures.

Override virtual base class functions by setting non-default function pointers in structs.

In the examples below, and the sample implementation, the objects are global and statically initialized. You can dynamically allocate them and/or initialize them at run time if you want.

1.1 Note on structure static initialization with designated initializers

The examples use the g++ form for structure designated initializers. So why are we using g++ and writing these objects in C? Long story for another day. Bottom line: Designated initializers are not part of the C++ standard, but g++ has an extension that allows them. The standard C99 way is:

```
{.field = value, ...}
```

g++ does it like this

```
{ field : value, ...}
```

Also, with g++, you have to list the fields in the order they are defined in the struct. (Sigh - at least it complains if the field order changes...) This extension also works with gcc, but the preferred way in gcc is to use the C99 technique.

Use the gcc/g++ extension technique if you have C files that will potentially be compiled with both gcc and g++.

If you are using a standard C compiler and don't care about C++, use the C99 technique for static initialization of structures.

You can always obfuscate things a little more and define an initializer macro that is ifdef'ed to detect C++ and/or g++. It's not a bad way to go if you're worried about portability. The framework defines the `DESIG_INIT` macro to do just that.

You don't have to use designated initializers in static initialization. That works in C and C++, but is kinda scary...

Note: You don't have to use static initialization at all. You can write constructors to do everything at run time if you want. In C the constructors kind of clutter things up, you have to explicitly call them anyway, and you can't use them for static initialization. So that's why we didn't include them. But if you want to use them, knock yourself out - they can be handy.

2 Overview

In this framework, State Machine (`Fsm`) objects only know what the current state is. There is no table of states to maintain. All the logic for triggering state transitions is coded in the event handlers.

A default "state handler" function (`FsmStateDefaultHandler`) takes care of finding event objects and calling event handlers for each event that states process. This default state handler also takes care of passing events to nested state machines.

In classic state machine implementations, you write a handler function for each state. The function usually handles each event id as a case in a switch statement, and either processes the event in place or calls an event handler.

In this framework, you do not need to write a state handler for each state. (You may, however, override the default state handler if you want.) You just initialize event objects which associate event ids with event handlers, and write the event handlers. The default state handler takes care of the rest.

Other than the event handler functions, there are no functions to write, unless you want to override the default state handler function.

You do not need to call the default state handler function directly unless you override it. In that case, you can (should!) call it from your override function.

2.1 How to use the State Machine classes

(See sections below for more details. Refer to fsm.h for base class definitions and functions.)

- Define a list of event IDs (an enum) for each FSM [Section 3]
- Define an `Fsm` object for each Finite State Machine (nested or not) [Section 5]
- Define prototypes for each event handler. There may be as many as one per event id per state. [Section 4 and 6]
- Define event objects (an event id and a handler function) for each event handled by each state. [Section 6]
- Create a list (an array) of event objects for each state. [Section 6]
- Define a state object for each state [Section 7]
- Write the event handler functions. [Section 4]

2.2 How to run the State Machine

After initializing the structures, call `FsmInit` for the top level state machine. This sets the initial state and sends an `EVT_FSM_ENTRY` event to the initial state. (Top level state machine event handlers must take care of initializing nested FSMs.)

Then for each event received, call `FsmDispatch` with the id of the event, and `pFsm` pointing to the top level state machine.

3 Define Event IDs

Define an enum for all events handled by your state machine, similar to this:

```
typedef enum {
    EVT_1 = EVT_FSM_EOL,
    EVT_2,
    EVT_3,
    EVT_4
} eMyEvent;
```

Use whatever enum symbols you want. The point is to make the first one equal to `EVT_FSM_EOL`, then let the rest of them be defined sequentially. `EVT_FSM_EOL` is defined in the framework, and is one greater than the last event ID defined by the framework. You must not define any event IDs with values less than `EVT_FSM_EOL`.

The event IDs defined by the framework are `EVT_FSM_ENTRY`, `EVT_FSM_EXIT`, `EVT_FSM_SUPERSTATE_ENTRY`, and `EVT_FSM_SUPERSTATE_EXIT` . You must write handlers for these event IDs in any state where you want to perform entry and/or exit actions for the state.

`FsmInit` sends `EVT_FSM_ENTRY` and `EVT_FSM_EXIT`.

The default state handler sends `EVT_FSM_ENTRY` and `EVT_FSM_EXIT` events to states at the current hierarchy level when state transitions occur, and sends `EVT_FSM_SUPERSTATE_ENTRY` and `EVT_FSM_SUPERSTATE_EXIT` to the substate. `EVT_FSM_ENTRY` and `EVT_FSM_SUPERSTATE_ENTRY` are sent when a transition to the state occurs. `EVT_FSM_EXIT` and `EVT_FSM_SUPERSTATE_EXIT` are sent when a transition away from the state occurs.

States can distinguish transitions at the superstate and current hierarchy levels. The intent of these distinct events is to allow two different methods of reentry from a superstate. In the first method, reentry from the superstate always restarts the nested state machine at a particular state. In the second method, reentry from the superstate allows the nested state machine to resume from where it left off.

`EVT_FSM_ENTRY` and `EVT_FSM_EXIT` tell the state that a transition has occurred at the current hierarchy level.

`EVT_FSM_SUPERSTATE_ENTRY` and `EVT_FSM_SUPERSTATE_EXIT` tell the state that a transition occurred in the superstate.

4 Event Handlers

You must write event handlers (functions) for all events handled in each state in your state machine. If a state doesn't handle an event, no handler for that state/event is required.

4.1 Event Handler Arguments

Event handlers have a specific prototype:

```
typedef FsmStatePtr (*FsmEvtHandler)(FsmState* pState, FsmEvent * pEvent);
```

You can use the `FSM_EVENT_HANDLER` macro to define your prototypes:

```
FSM_EVENT_HANDLER( MyNestedFsm1State1Evt_Entry );
```

Event handlers return a pointer to the next state (`FsmState*` or `FsmStatePtr`) when an event causes a transition. Event handlers return `NULL` when an event does not cause a transition.

If an event handler "consumes" an event (i.e., no further processing of the event is to be performed by a higher level state machine), the event handler sets

```
pEvent->consumed = true;
```

The 'consumed' field is false on entry to the event handler.

Note: If an event handler returns a pointer to the current state, it will cause exit actions and entry actions to be performed even though there is no transition. Most likely you want the event handler to return NULL when there is no transition.

Also Note: The handler function prototype for entry and exit events (`EVT_FSM_ENTRY`, `EVT_FSM_EXIT`, `EVT_FSM_SUPERSTATE_ENTRY`, and `EVT_FSM_SUPERSTATE_EXIT`) is the same as for other event handlers. However, exit event handlers should return `NULL`; i.e., *do not transition in an exit event handler*. The framework default state handler ignores `non-NULL` values returned by exit event handlers (otherwise, in this framework, that causes an infinite recursive loop).

4.2 Hierarchical Event Processing

The default state handler processes events in this order

- `EVT_FSM_ENTRY` passed to top-level state first, then `EVT_FSM_SUPERSTATE_ENTRY` passed to nested FSM. Lowest-level FSM event handler should set `pEvent->consumed = true` . This means entry events are processed in top-down order, which is similar to the order of class constructor execution in C++.
- When the handler sees `EVT_FSM_EXIT`, it sends `EVT_FSM_SUPERSTATE_EXIT` to the substate first, then sends `EVT_FSM_EXIT` to the current hierarchy state.
- All other events, including `EVT_FSM_EXIT`, are processed in bottom-up order; i.e., sub-states get the opportunity to override higher level states. If a sub-state event handler sets `pEvent->consumed = true` , the event is not passed to higher-level states' event handlers.

- EVT_FSM_EXIT/EVT_FSM_SUPERSTATE_EXIT event handlers should ***never*** set pEvent->consumed = true. This allows higher-level states to do exit processing after lower-level states, which is similar to destructor processing order in C++..

5 Define Fsm Objects

You can define Fsm objects directly:

```
Fsm myTopFsm = { name : "top",pState : NULL };      // instantiate the top level (superstate) FSM
Fsm myNested1Fsm = { name : "nested_1", pState : NULL };   // instantiate a nested FSM
Fsm myNested2Fsm = { name : "nested_2", pState : NULL };   // instantiate a nested FSM
```

Or you can use the FSM macro:

```
FSM(myTopFsm,     "top",      NULL ); // instantiate the top level (superstate) FSM
FSM(myNested1Fsm, "nested_1", NULL ); // instantiate a nested FSM
FSM(myNested2Fsm, "nested_2", NULL ); // instantiate a nested FSM
```

Note there is no difference instantiating a nested state machine or a non-nested one. To nest an FSM, initialize the pNestedFsm field of an FsmState object to point to the nested FSM. (See details in Section 7 "State Objects" below.)

6 Define Event objects and lists

You need to create event objects which associate event ids with functions that handle the events. The same id may have different handlers in different states - you need a separate event object for each (id, handler) pair.

So you need to define prototypes for your handlers, instantiate event objects, then put pointers to the event objects into event lists. Each event list defines a set of events that a state handles.

The framework defines one event object: fsmNullEvent. You have to terminate each event list with a pointer to the fsmNullEvent object so the default state handler can find the end of the list. It doesn't matter what order events are placed in the list.

The example below defines event handler prototypes, event objects, and an event list for a single state using the FSM_EVENT_HANDLER and FSM_EVENT macros:

```
//++++ Nested FSM 1 State 1 events ++++

// Method prototypes
FSM_EVENT_HANDLER( MyNestedFsm1State1Evt_Entry );
FSM_EVENT_HANDLER( MyNestedFsm1State1Evt_EVT3 );

// Objects
FSM_EVENT( nested1State1Evt_Entry, EVT_FSM_ENTRY, MyNestedFsm1State1Evt_Entry );
FSM_EVENT( nested1State1Evt_EVT3,  EVT_3,         MyNestedFsm1State1Evt_EVT3  );

// Event list array
FsmEvent* nested1State1_EventList[] = {
            &nested1State1Evt_Entry,  // superstate entry events ignored
            &nested1State1Evt_EVT3,
            // keep this last
            &fsmNullEvent
};
```

Do similarly for the events handled by each state. The example uses a naming scheme to help identify the state which event handlers belong to. It's not required - this is just C, so you can name the functions anything you want. The important thing is to initialize event objects to associate event ids with handlers, then put event objects in a list for each state. You'll use the event list in the state object (`FsmState`) instantiation [see Section 7 below].

7 State Objects

7.1 Instantiating State Objects

Instantiate and initialize state objects (`FsmState`) for each state in the state machine. The example below sets the parent state machine to the nested FSM, uses the event list created in Section 6 above, has no nested state machine to pass events to, and uses an override state handler.

```
FsmState nested1State_1 = {
    pFsm : &myNested1Fsm,
    pNestedFsm : NULL,
    eventList : nested1State1_EventList,
    name : "Nested1State1",
    pfnStateHandler : MyFsmStateHandler
};
```

To use the default state handler, initialize `pNestedFsm` to `FsmStateDefaultHandler`.

You can also use the `FSM_STATE` macro to instantiate `FsmState` objects:

```
FSM_STATE(nested1State_1,&myNested1Fsm,NULL,nested1State1_EventList,"Nested1State1",MyFsmStateHandler );
```

You only need to initialize the data structure for each state. There is no state handler to write unless you override the default state handler.

7.2 Nested FSMs

To create a nested FSM hierarchy, create lists of FSMs. Then initialize each state object's nested FSM list pointer (`nestedFsmList`) to point to an FSM list.

The example below creates a top-level state with a nested state machine:

```
// Top state 1 Nested FSM list array
Fsm* topState1_NestedFsmList[] = {
           &myNested1Fsm,
           NULL
};

FsmState topState_1 = {
    pFsm : &myTopFsm,
    nestedFsmList : topState1_NestedFsmList,
    eventList : topState1_EventList,
    name : "TopState1",
    pfnStateHandler : MyFsmStateHandler
};
```

Or, with the `FSM_STATE` macro:

```
FSM_STATE(topState_1,&myTopFsm,topState1_NestedFsmList,topState1_EventList,"TopState1",MyFsmStateHandler);
```

This allows a single state to have multiple nested FSMs. This is particularly useful for implementing UML "orthogonal regions" (sometimes called "cooperative state machines") as nested component FSMs (see the pdf file *Pattern Orthogonal.pdf* by Miro Samek in the example folder for a discussion of this technique).

The default state handler dispatches events to all nested FSMs in the order of the nested FSM list. All event handlers for nested FSMs run in the same thread of execution (e.g., a single thread or task in a Real Time Operating System) as the state handler. For nested states to communicate or synchronize with each other, they must post events to a queue for the dispatcher to read. Event handlers must not call "dispatch" directly; otherwise, the framework's Run To Completion model will be broken.

The framework does not implement queues. It is up to the implementation to provide queues and queue handling.

7.3 Overriding the Default State Handler

To override the default state handler, initialize `FsmState` objects' `pfnStateHandler` to point to a different function than `FsmStateDefaultHandler`.

A simple override state handler creates log entries for all events as they pass through the state machine. In this case, set `pfnStateHandler` for all states to point to your override function, such as:

```
pfnStateHandler : MyFsmStateHandler
```

Then implement `MyFsmStateHandler` like this:

```
bool MyFsmStateHandler(FsmState *pState, int eventId)
{
   bool consumed;

   FSM_ENTER_LOG("%s,%s,%s", pState->pFsm->name, pState->name,  EVT_NAME(eventId));

   consumed = FsmStateDefaultHandler(pState, eventId);

   FSM_EXIT_LOG("%s,%s,%s,%sconsumed", pState->pFsm->name, pState->name,
                        EVT_NAME(eventId), (consumed==0? "not_" : ""));

   return consumed;

} // MyFsmStateHandleEvent
```

7.4 Extending the State structure

If you need extended state variables to evaluate guard conditions in your event handlers (i.e., you have conditional transitions), you can derive your own State structure from the base class by including the base class structure as the first field in your derived structure. In the example below, the derived structure is named `MyFsmState`.

```
typedef struct {
     <some field definitions>
} MyExtraStuff;

typedef struct {
     FsmState     base;
     MyExtraStuff myStuff;
} MyFsmState;
```

Then create your state objects as `MyFsmState` objects. When your event handlers want to change states, they need to return pointers to `FsmState` objects, so they'll need to cast `MyFsmState` pointers to FsmState pointers for the return value (you wouldn't have to do the cast in C++, but, alas, this is C).

Likewise, to access the extended variables in the `MyFsmState` struct, event handlers need to cast the event handler `FsmStatePtr` argument to a `MyFsmState` pointer.

As long as the FsmState structure is the first element of the derived structure, the framework will handle everything correctly.

8 Trace Log Output

Macros `FSM_ENTER_LOG`, `FSM_EXIT_LOG`, and `FSM_RUN_LOG` are defined for tracing. This is a very handy thing to do. We implemented tracing outside the framework to allow tracing in some state machines and not others, rather than having traces from all state machines jumbled together.

The tracing macros, when used as shown in fsm_example.c, create log output that can be pasted into a .csv text file, and opened with a spreadsheet program. If using Microsoft Excel, open the .csv file and paste the contents into fsm-trace.xlsx, beginning at cell A2. This spreadsheet has conditional formatting to make reading the trace output easier.

To use the macros:

- Override the default state handler
- In your state handler, include the `FSM_ENTER_LOG` / `FSM_EXIT_LOG` macros, as shown in Section 0
- Use `FSM_RUN_LOG` as you like, typically in event handlers.
- Define `#define FSM_TRACE 1` in your .c file before including fsm.h

9 Example State Machine

Refer to the diagram in example-state-machine.pdf.

The example state machine has a top-level state machine and two nested state machines.

When the machine transitions to Top State 1, Nested FSM1 always resumes from the state it was in when the machine transitioned away from Top State 1. In this state machine, we want to resume without going through entry and exit events, so entry/exit events from the superstate are ignored.

When the machine transitions to Top State 2, Nested FSM 2 always resumes from Nested State 1. When the machine transitions away from Top State 2, Nested FSM 2 receives exit events for whatever state it is in. The entry events for Nested State 1 occur when the machine transitions back to Top State 2.

The event handlers in the example FSM do the housekeeping required by the framework; i.e. :

- return a pointer to the next state for a transition, or NULL for no transition
- set the `pState->consumed` field if necessary
- setup reentry for substates in entry event handlers

Normally your state machine will have more to do in the event handlers.