Module 3

Design and Optimization of Embedded and Real-time Systems

1 Finite State-Machine (FSM)

Finite state-machine is the foundation of the theory of computation. It can often used as a mathematical model to study formal systems. It is very popular in telecommunication and aerospace industries to design protocol stacks and ensure system reliability. Apart from that it is surprisingly not commonly used by many software engineers, even in UI development where the concept of state is dominant.

Someone might say a state-machine adds complexity, as a traditional state diagram can look quite complicated with numerous states and transitions interwoven like a nest. It's both true and false.

It is true that a traditional state diagram (flat) due to its infamous state explosion problem fails to express any realistic design in a neat way. Later we will see how statecharts (hierarchical) solve this problem with some innovative and intuitive concepts.

It is false that using state-machines increases the complexity of your design. In many cases, complexity is inherent in the system itself, as it is required to handle all possible scenarios. Even when you are not using a state-machine explicitly, you may just be implementing an *implicit* state-machine with *flags* and *if-else* statements without knowing about it. An explicit state-machine simply helps reveal the inherent complexity so that we can visualize it, discuss and reason about it. When you find your state diagram over-complicated, you will likely find a way to simplify it and hence yield a simpler design.

1.1 FSM Notation

What is a finite state-machine? Any textbook on the theory of computation should give you a *formal* definition of FSM in the first few chapters. It is expressed as a tuple of a finite set of states (`S`), a finite set of events (`E`) and a transition function (`S x E -> S`).

In plain programming terms, you have an enumeration of *State* and an enumeration of *Event*. *State* lists all the possible *situations* or *conditions* that your software component is concerned of when reacting to an event. *Event* lists all the different types of signals that your software component may receive, upon which some actions *may* be taken along with a *possible* state change (called transition).

This is a simple example:

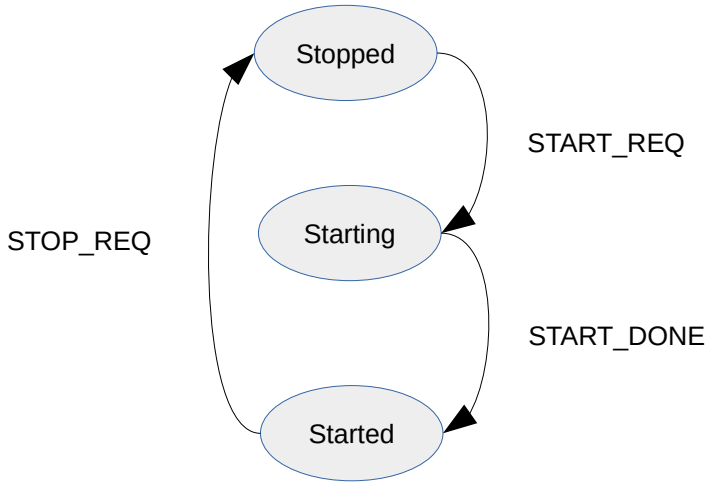
```
enum State {
    STOPPED,
    STARTING,
```

Copyright (C) 2017-2022 Lawrence Lo. All rights reserved.

```
    STARTED
};

enum Event {
    START_REQ,
    START_DONE,
    STOP_REQ
};
```

The corresponding state diagram is shown below. There is no standard convention of a traditional state diagram. Typically we use a circle or eclipse to represent a *state*, a label for an *event* and a directed line for a transition.



Once a state diagram is drawn, software developers have a few options to implement it in executable code like C/C++. We will look at some approaches in the next section.

1.2 Philosophy

Before moving on let's pause for a moment to reflect on what is special about *state*. *State* exists in our software without us thinking about it. Ultimately what software does is to take some actions upon inputs (events) it receives. The actions it takes depend not only on the current input but also on the current *condition* or *state* of the system (which is affected by the history of past inputs). The question is: How do we represent such conditions or states in software?

At the bottom layer it is bits of zeros and ones. At a higher level, we abstract groups of bits into bytes or words, and eventually into *variables*. One simple approach is to just represent states with variables. Upon an input, our software determines what actions to take based on the current values of those variables.

This should sound familiar to us, since we know from our software experiences that programming

Copyright (C) 2017-2022 Lawrence Lo. All rights reserved.

involves (i) defining variables and (ii) use conditional constructs like *if-else* and *switch-case* together with logical operators like `&&` and `||` to check on some combinations of variables. The issue is just a single 32-bit variable can store 2^{32} possible values or states. If you combine multiple of them together in a logical expression, the total number of possible combinations or cases to *consider* could be astronomically higher. Have you seen conditional code like this before?

```
if ((battLevel >= THRESHOLD_1) && (battLevel < THRESHOLD_2) &&
    !((temperature < TEMP_LOW) || (temperature >= TEMP_HIGH)) &&
    (usbHal.IsPluggedIn() && !dcCharger.IsPluggedIn()) &&
    (usbHal.ReadRegister(OVER_CURRENT_ERROR_COUNT) < 3) &&
    (!chargingStarted) {
    // Starts CC USB charging.
    usbHal.StartCharging(CONSTANT_CURRENT_MODE);
    chargingStarted = true;
    ....
}
```

When looking at a particular conditional statement individually we might still understand what will happen under certain specific conditions. In the example above we could understand under what conditions would USB charging be started. However if we *just* do that, we are ignoring all the other conditions. We can't tell if something else is going to happen when the condition above is evaluated to be false, and if so what those actions will be. Either the design is incomplete if it hasn't considered all possible cases, or those other cases are scattered or buried somewhere else in the source code which could be very hard to reveal to form a complete picture.

Fundamentally it is a problem of *abstraction*. Out of the possible $(2^{32})^N$ possible combinations of values of N integer variables, only a handful are meaningful to us. Defining *states* allows us to think at a higher level of abstraction, i.e. to reason about a relatively small space of *discrete states* than a much larger space of *continuous variables*.

A state diagram makes it very clear and explicit when state is changed and what it is changed to. If we use a combination of variables or flags to represent state implicitly, we need to be concerned of when each one of those variables or flags is *set*, *cleared* and *checked*. As they tend to be scattered in the source code across multiple files, it can be a daunting task especially when the author is somebody else.

1.3 Traditional Implementation

See *Chapter 3 Standard State Machine Implementations* of the PSiCC book for details. This is a summary of those approaches.

1. Double-switch

   Use the first level of switch-case to check states and the second level to check for events. A simplified example is shown below:

Copyright (C) 2017-2022 Lawrence Lo. All rights reserved.

```
void Dispatch(Event event) {
    switch (currentState) {
        case STOPPED:  StoppedStateHandler(event); break;
        case STARTING: StartingStateHandler(event); break;
        case STARTED:  StartedStateHandler(event); break;
    }
}

void StoppedStateHandler(Event event) {
    switch(event) {
        case START_REQ:
            DoStart();
            currentState = STARTING;
            break;
        case START_DONE:
            LOG("Unexpected event START_DONE in state STOPPED.");
            break;
        case STOP_REQ:
            LOG("Ignored event STOP_REQ in state STOPPED.");
            break;
    }
}
```

2. State table

   Rather than having state and event checks in explicit switch-case statements, we encode the *business logic* in a data table so the code itself becomes simpler and generic. The underlying idea is similar to the double-switch approach. There are a few variations of how much information is encoded in the table. Some approaches encode just the handler function for each combination of the current state and event while others include a guard condition and the next state. The example below illustrates the minimal approach:

```
typedef void (*Handler)(Event event);

void StoppedStateStartReq(Event event) {
    DoStart();
    currentState = STARTING;
}
void UnexpectedEvent(Event event) {
    LOG("Unexpected event %d in state %d", event, currentState);
}
void IgnoredEvent(Event event) {
    LOG("Ignored event %d in state %d", event, currentState);
}
...
```

Copyright (C) 2017-2022 Lawrence Lo. All rights reserved.

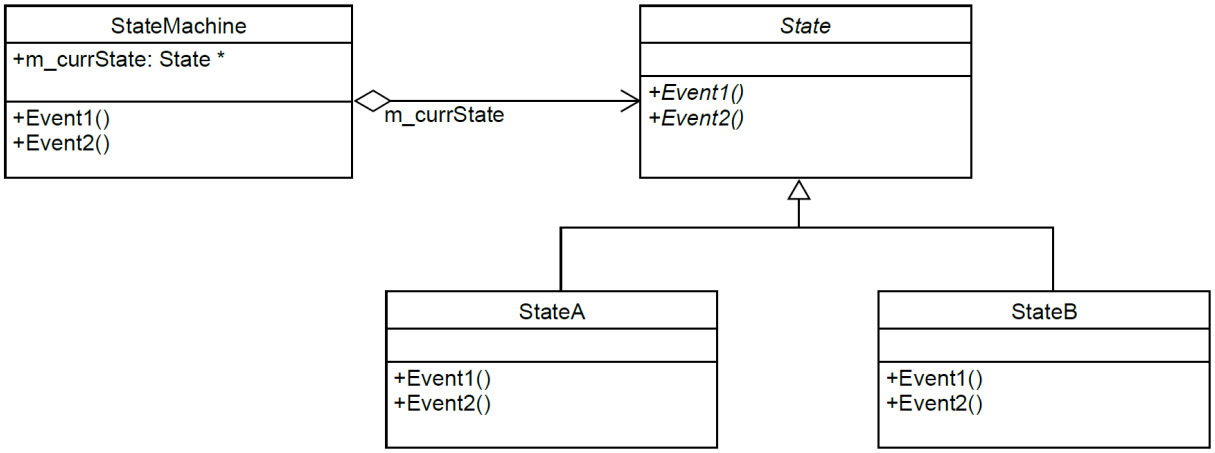
```
const Handler table[STATE_COUNT][EVENT_COUNT] = {
    { StoppedStateStartReq, UnexpectedEvent,        IgnoredEvent        },
    { IgnoredEvent,         StartingStateStartDone, UnexpectedEvent     },
    { IgnoredEvent,         UnexpectedEvent,        StartedStateStopReq }
};

void Dispatch(Event event) {
    table[currentState][event](event);
}
```

3. State pattern.

   This approach comes from the *State Pattern* in the classic GoF book on object-oriented design patterns (*Design Patterns: Elements of Reusable Object-Oriented Software. Erich Gamma, John Vlissides, etc. Addison-Wesley. 1994.*). Each state is represented by a subclass derived from a common base class for all states. Each event is handled by a virtual function declared in the base class, allowing each individual state to customize its handling of the event by overriding the virtual function in its own subclass.

   A pointer to the base class points to the subclass object representing the current state. An event is dispatched to the current state by invoking the corresponding virtual function via that base class pointer. This is an elegant approach. However due to the lack of support for state hierarchy, it isn't really scalable to real-life projects and is overshadowed by more modern approaches including QP.



Copyright (C) 2017-2022 Lawrence Lo. All rights reserved.

2 Hierarchical State-Machine (HSM)

2.1 Statechart Notation

2.1.1 An Anecdote

Please read PSiCC Chapter 2 for a crash course in statechart notation. For a detailed guide read OMG UML Version 2.5.1 Chapter 14 StateMachines (About the Unified Modeling Language Specification Version 2.5.1 (omg.org)).

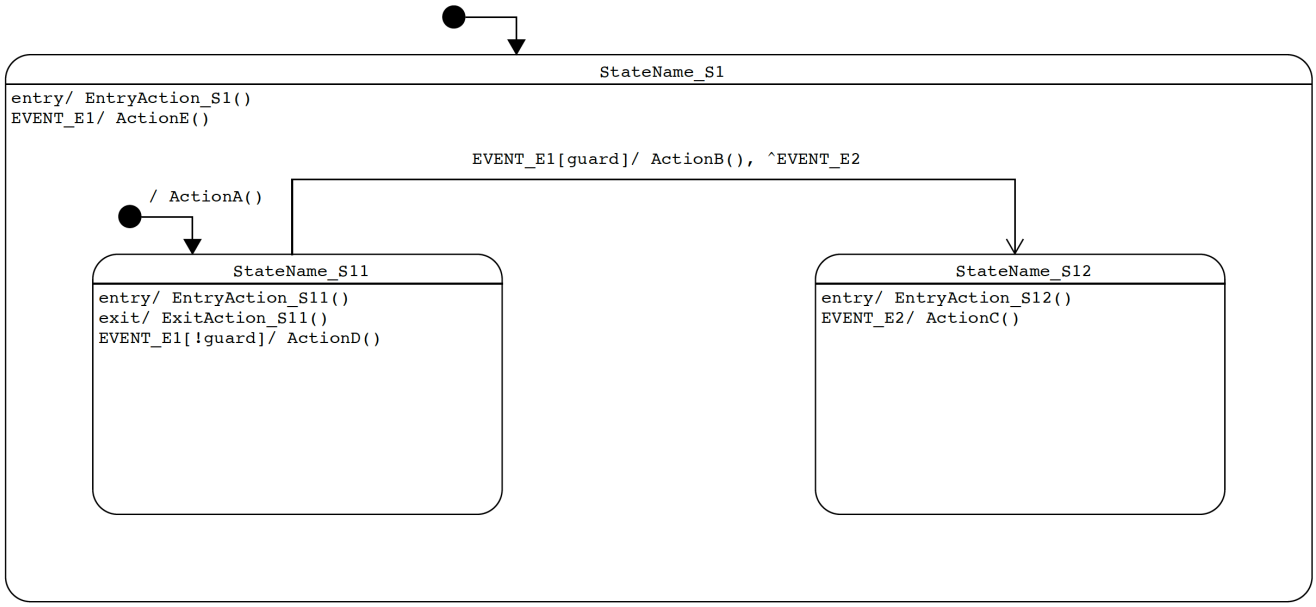
Let's revisit the paper *Statecharts in the Making* by D. Harel. This is another quote from it:

"I recall an anecdote from somewhere in late 1983, in which in the midst of one of the sessions at the IAI the blackboard contained a rather complicated statechart that specified the intricate behavior of some portion of the Lavi avionics system... There was a knock on the door and in came one of the air force pilots from the headquarters of the project. He was a member of the "customer" requirements team, so he knew all about the intended aircraft (and eventually he would probably be able to fly one pretty easily too…), was smart and intelligent, but he had never seen a state machine or a state diagram before, not to mention a statechart. He stared for a moment at this picture on the blackboard, with its complicated mess of blobs, blobs inside other blobs, colored arrows splitting and merging, etc., and asked "What's that?" One of the members of the team said "Oh, that's the behavior of the so-and-so part of the system, and, by the way, these rounded rectangles are states, and the arrows are transitions between states". And that was all that was said. The pilot stood there studying the blackboard for a minute or two, and then said, "I think you have a mistake down here, this arrow should go over here and not over there"; and he was right. For me, this little event was significant, as it really seemed to indicate that perhaps what was happening was "right", that maybe this was a good and useful way of doing things. ***If an outsider could come in, just like that, and be able to grasp something that was pretty complicated but without being exposed to the technical details of the language or the approach, then maybe we are on the right track.*** *Very encouraging."*

2.1.2 Crash Course

This is my even more concise version of a crash course on statechart notation. Let's take a look at a simple and hypothetical statechart below. It doesn't do anything meaningful but does illustrate most of the important notation and concepts. In the subsequent sections, we will gradually introduce more advanced concepts.

Copyright (C) 2017-2022 Lawrence Lo. All rights reserved.



1. State

   a) A *state* is represented by a round-cornered rectangle with its name denoted in the upper compartment.

   b) A state may contain other states. The containing state is called a *superstate* or *composite state*. The contained state is called a *substate* or *nested state*. There can be multiple levels of nesting. This is like superclass and subclass relationship in OOP.

   c) Due to the hierarchy of state composition, this kind of state-machine is called *hierarchical state-machine*, or simply ***HSM***. We will refer to this acronym a lot, so it's important to remember what it means.

   d) When the system is in StateName_S1, it must also be in either StateName_S11 or StateName_S12 but not both.

      StateName_S1 is also called an *OR-state.* Whenever the system is in an OR-state, the system must also be in exactly one of its substates. Later we will see another kind of composite state called an *AND-state*.

   e) The default initial state is indicated by a dark solid circle/dot called the *initial pseudo-state*. The transition from it points to the default state to enter at each nesting level.

      In our example the system enters StateName_S1 followed by entering StateName_S11.

   f) In each state there can be optional *entry action* and *exit action*, labeled as

      `entry/ action_list`

      `exit/ action_list`

Copyright (C) 2017-2022 Lawrence Lo. All rights reserved.

See the next bullet point for an explanation of `action_list`. Essentially an entry action is the activities to be performed every time when a state is entered, no matter how (i.e. via whichever transition) the state is entered. Similarly an exit action is the activities to be performed every time when a state is exited. (This is a core feature of statecharts.)

2. Transition (Non-internal)

   a) A *transition* is a directed line from a *source state* to a *target state* triggered by an occurrence of an event labeled in the following format:

      `EVENT_NAME[guard_condition]/ action_list`

   b) `EVENT_NAME` identifies the type of the triggering event, such as EVENT_E1 or more concretely START_REQ and START_DONE.

   c) `guard_condition` is a logical expression to determine if a transition is enabled. A transition is only *enabled* if the triggering event arrives *and* the associated guard condition is evaluated to true. Examples include:

      [retryCount < 3]

      [temperature >= 0 && temperature < 90]

      Guard conditions uses *extended state variables* which are member variables of state-machine objects. They are very useful in augmenting the discrete states but they are nonetheless the very things state-machines are trying to replace, and therefore we need to find a sweet spot and not overuse them.

   d) `action_list` is a comma-separated list of activities to perform when the transition is triggered or fired. A transition is triggered when it is *enabled* and *selected*. We mentioned above that the guard_condition determines if a transition is enabled. Later we will discuss how a transition gets selected when there is a conflict among multiple enabled transitions (conflict resolution).

      An `action_list` can be empty, meaning that there are no activities to accompany the transition. Otherwise it may contain a combination of (i) actual functions to be called, (2) events to be posted (via *Send()* or *Raise()*) or (3) free-formed descriptions. Examples of `action_list` include:

      / gpioHal.SetPin(LED1), motorDriver.Forward(speed)

      / turn on LED1, forward motor at speed

      / retryCount++, raise(RETRY)

      In the above example, RETRY is called a *reminder* (or *internal*) event. It is posted to itself to trigger some followup activities to be performed. There are different favors regarding

Copyright (C) 2017-2022 Lawrence Lo. All rights reserved.

how an event is posted, which are related to the underlying queuing model.

e) Initial transition is a directed line from an initial pseudo-state (dark solid dot) to the default state to enter. By nature of being *default*, there shouldn't be `EVENT_NAME` or `[guard_condition]` in the label. (There is an exception. If there is a choice point or junction, there can be a `[guard_condition]` in each branch coming out of the choice point or junction.)

3. Internal Transition

a) An internal transition is a transition without state change (a.k.a a *targetless transition*). It does sound contradictory. It remains in the same state without even exiting and re-entering the state, and hence there are no associated exit and entry actions. Since there is no state change there is no need for a directed line. Only a label is needed, which is placed inside the state rectangle at the top-left corner.

Question: How is it different from a transition with the *source state* and the *target state* be the same?

b) Internal transitions can be viewed as the activities to be performed while remaining in a certain state. Often these activities represent the main features of a system when it is in a stable state, such as transferring data, handling periodic timeout events, etc. Together with the transition lines, they allow the complete system behaviors to be represented, which is not possible with traditional state diagrams.

2.2 Statechart Rules

2.2.1 State Hierarchy and Transition Triggering

In a statechart, states are organized in a tree structure. Starting from the topmost root state, each state can contain substates. The relationship between a substate and its containing states (superstates) is like that between a subclass and its superclasses, which is an "is-a" relationship. For example if the system is in StateName_S11 we say it is also in StateName_S1.

When an event arrives, we first determine the set of enabled transitions in the current leaf state (most nested) and all of its superstates. (Recall that a transition is enabled when its guard condition, if any, is evaluated to true.)

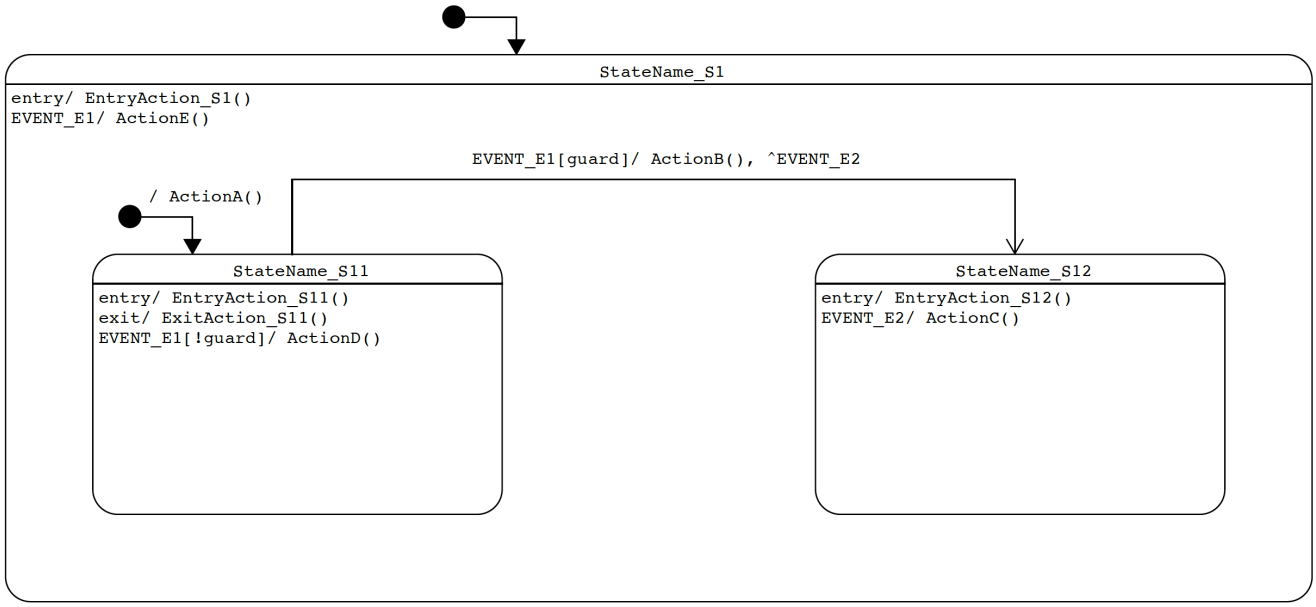
There is a conflict if there are more than one enabled transitions in the set. The system will attempt to resolve the conflict by selecting one transition out of the set in the order from the most nested state outward to the root state. The conflict cannot be unresolved if there are more than one enabled transitions in the *same* state, and we will call the statechart malformed.

In other words, priority is given to a more nested state, in a way similar to how overriding functions are

Copyright (C) 2017-2022 Lawrence Lo. All rights reserved.

resolved from the most derived class to the base class in OOP.

In the example statechart above (reproduced here), we observe that:



1. When EVENT_E1 arrives in StateName_S11,
   a) if guard is true, ActionB() will be called and the system transitions to StateName_S12.
   b) if guard is false, ActionD() will be called and the system stays in the same state.
   c) ActionE() in the superstate will never be called.
2. When EVENT_E2 arrives in StateName_S11, there are no enabled transitions and the event will be discarded.
3. When EVENT_E2 arrives in StateName_S12, ActionC() will be called.
4. When EVENT_E1 arrives in StateName_S12, ActionE() in the superstate will be called.

Note – There is a subtle difference between a guard condition and an *if* statement in the action list. In the former case, if a guard condition is false the transition is not enabled, and therefore will not be considered for selection. In the latter case, no matter whether the if-condition is true or false the transition, when selected, has already fired, and therefore overrides any other transitions in its parent states.

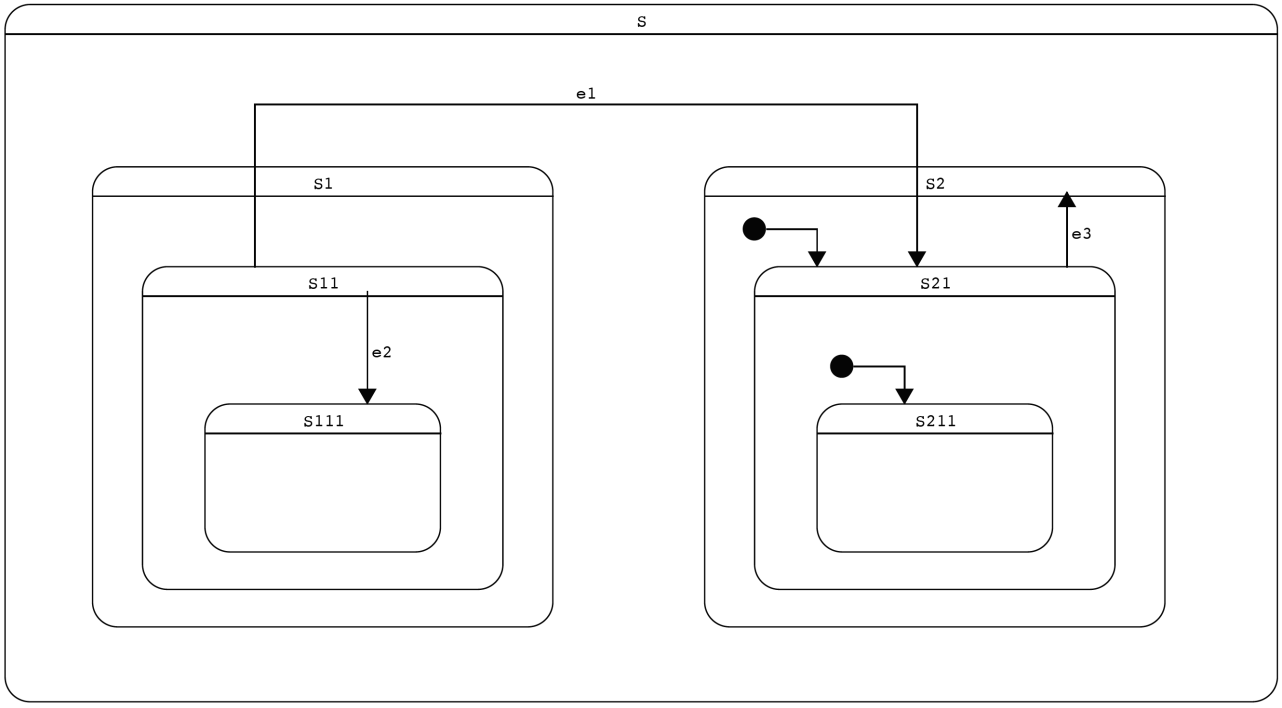
2.2.2 State Exits and Entries in a Transition

Let's define some terminology for this discussion:

Copyright (C) 2017-2022 Lawrence Lo. All rights reserved.

1. *Current state* – The leaf (most nested) state the system is in at the moment.
2. *Transition source* – The originating state of a transition line.
3. *Transition target* – The terminating state of a transition line.
4. *Final state* – The new leaf state after transition.

A *transition source* can be the same as the *current state*, or a parent of the current state. A *final state* can be the same as a *transition target*, or a child of the transition target.



For example, when e1 arrives in S111, the transition source (S11) is the parent of the current state (S111). The transition target (S21) is the parent of the final state (S211) due to initial transition.

During a state transition, states will be exited and entered, and the corresponding exit and entry actions will be *automatically* performed. The sequence of actions can be divided into the following three main steps:

1. **Exit actions from the current state up to *but not including* the transition source.** When both are the same there will be no actions in this step.
2. **Exit actions from the transition source up to but not including the *lowest (least) common ancestor* (LCA), followed by entry actions from there into the transition target.**

   According to this rule, even when the transition source is the same as the transition target, that source/target state is still exited and re-entered.

Copyright (C) 2017-2022 Lawrence Lo. All rights reserved.

Exception cases with the QP implementation:

a) If the transition source is a parent of the transition target (e2), the transition source is not exited and re-entered.

b) If the transition source is a substate of the transition target (e3), the transition target is not exited and re-entered.

See Figure 2.10 on page 81 of PSiCC by Miro Samek for details (Book: Practical UML Statecharts in C/C++, 2nd Ed. (state-machine.com)).

Note 1 – This interpretation of *local transition* does not seem to be entirely agreeing with the UML 2.5.1 spec. See Section *14.2.3.8.1 Transition kinds relative to source* and Section *14.2.4.9 TransitionKind* of UML 2.5.1 spec available at About the Unified Modeling Language Specification Version 2.5.1 (omg.org).

Note 2 – Also see SCXML spec at State Chart XML (SCXML): State Machine Notation for Control Abstraction (w3.org). There is a notion of transitions with type of "*internal*", which are very similar to *local transitions* in UML 2.5.1:

"*The behavior of transitions with 'type' of "internal" is identical, except in the case of a transition whose source state is a compound state and whose target(s) is a descendant of the source. In such a case, an internal transition will not exit and re-enter its source state, while an external one will, as shown in the example below.*"

3. **Entry actions from *but not including* the transition target into the final state.**

   If the transition target is the same as the final state there will be no actions in this step; otherwise the system will follow the initial transitions repeatedly until reaching the final state.

Though the rules may sound complicated, they are very intuitive. Try to follow the transitions above (e1, e2 and e3) to check if the results obtained by the rules match your intuition.

3 Introduction to Quantum Platform (QP)

Now that we know what statecharts are and how more complex they are compared to traditional state diagrams, we are sure that conventional methods for FSM (e.g double-switch, state table and state pattern) are not sufficient to implement statecharts. Let's see how QP solves this problem elegantly.

3.1 Basic Ideas

Like the state table method, QP represents states with functions. It implements one function for each state (including all composite and leaf states). It maintains a function pointer to point to the function of the current leaf state. QP is different from the state table method in that it does not implement a separate function for each event a state handles.

Copyright (C) 2017-2022 Lawrence Lo. All rights reserved.

Like the state pattern method, QP represents each *state-machine* with a class. However it does *not* implement each state in its own class. Instead QP implements each state as a member function of the state-machine class which automatically provides a context for all the states.

Like the double-switch method, it uses a switch-case construct to handle different events in a given state.

In other words, QP combines the good parts of existing methods and patterns to form a new one.

3.2 State Member Function

The center piece of QP is the use of a member function to implement each state of an HSM. QP provides a base class QHsm as the basis for all state-machine classes. Developers derive their own application state-machine classes from QHsm and implement their states as member functions of those derived classes.

We will see a complete example in the next section. First we present the basic form of a state member function:

```
QState MyHsm::MyState(MyHsm * const me, QEvt const * const e) {
    switch (e->sig) {
        case Q_ENTRY_SIG: {                     // Entry actions.
            ...
            return Q_HANDLED();
        }
        case Q_EXIT_SIG: {                      // Exit actions.
            ...
            return Q_HANDLED();
        }
        case Q_INIT_SIG: {                      // Initial transition.
            ...
            return Q_TRAN(&MyHsm::MyDefaultSubState);
        }
        case MY_EVENT_A: {
            ...
            return Q_TRAN(&MyHsm::MyTargetState); // State transition.
        }
        case MY_EVENT_B: {
            ...
            return Q_HANDLED();                 // Internal transition.
        }
        case MY_EVENT_C: {
            if (me->MyGuard()) {
                ...
                return Q_HANDLED();             // Guard condition passes
            }
            break;                              // Ignored in this state.
        }
    }                                           // Event not handled.
    return Q_SUPER(&MyHsm::MySuperState);       // Returns super state.
}
```

Copyright (C) 2017-2022 Lawrence Lo. All rights reserved.

The basic form above contains almost all important concepts of a statechart.

1. MyState is a member function of the state-machine class MyHsm. Since it is declared as a *static* member function, the "*this*" pointer to the object is passed explicitly into the function via the parameter *me*. This is not necessary but is done this way in QP to support certain compilers that do not support pointer-to-member-functions properly. You can have multiple instances of a state-machine class.

   Note: In some other state-machine frameworks, the "*me*" parameter is also known as the *context* or *model* of the state-machine and is referred to explicitly via a pointer like "*me*" here.

2. The event to be handled is passed in as the second parameter *e*. It is *passed by reference* as a pointer to the event base class *QEvt*. From its member *sig* the switch-case statement differentiates which event has arrived.

   Once it has figured out the event type, it can downcast *e* to the actual subclass derived from QEvt. It is done this way to avoid the overhead of virtual functions or RTTI.

3. QP provides built-in event types, namely `Q_ENTRY_SIG`, `Q_EXIT_SIG` and `Q_INIT_SIG` as annotated above. We can add as many application specific events as we need, such as `MY_EVENT_A`, `MY_EVENT_B` and `MY_EVENT_C` shown above.

4. A guard condition is implemented with an *if* statement. Note that if a guard condition is evaluated to false, the transition is *disabled* and the event must be propagated to the super states until a transition is selected (or the event is discarded if none is found).

   Question – What is the difference between the following two transitions?

   a) `MY_EVENT_C[EvalCond()] / ...`

   b) `MY_EVENT_C/ if (EvalCond()) ...`

5. The return value of a state member function informs QP the result of event handling by this state. There are three options via macros provided by QP:

   a) **`Q_HANDLED()`** - The event is handled by either an *entry action*, *exit action* or an *internal transition*.

   b) **`Q_TRAN()`** - The event is handled by a *state transition* (including *initial transition*) to a different or the same state. The target state is specified as a pointer to member function. All exit and entry actions along the path of a transition will be performed *automatically* by QP which dispatches `Q_ENTRY_SIG`, `Q_EXIT_SIG` and `Q_INIT_SIG` to all involved states in the proper order.

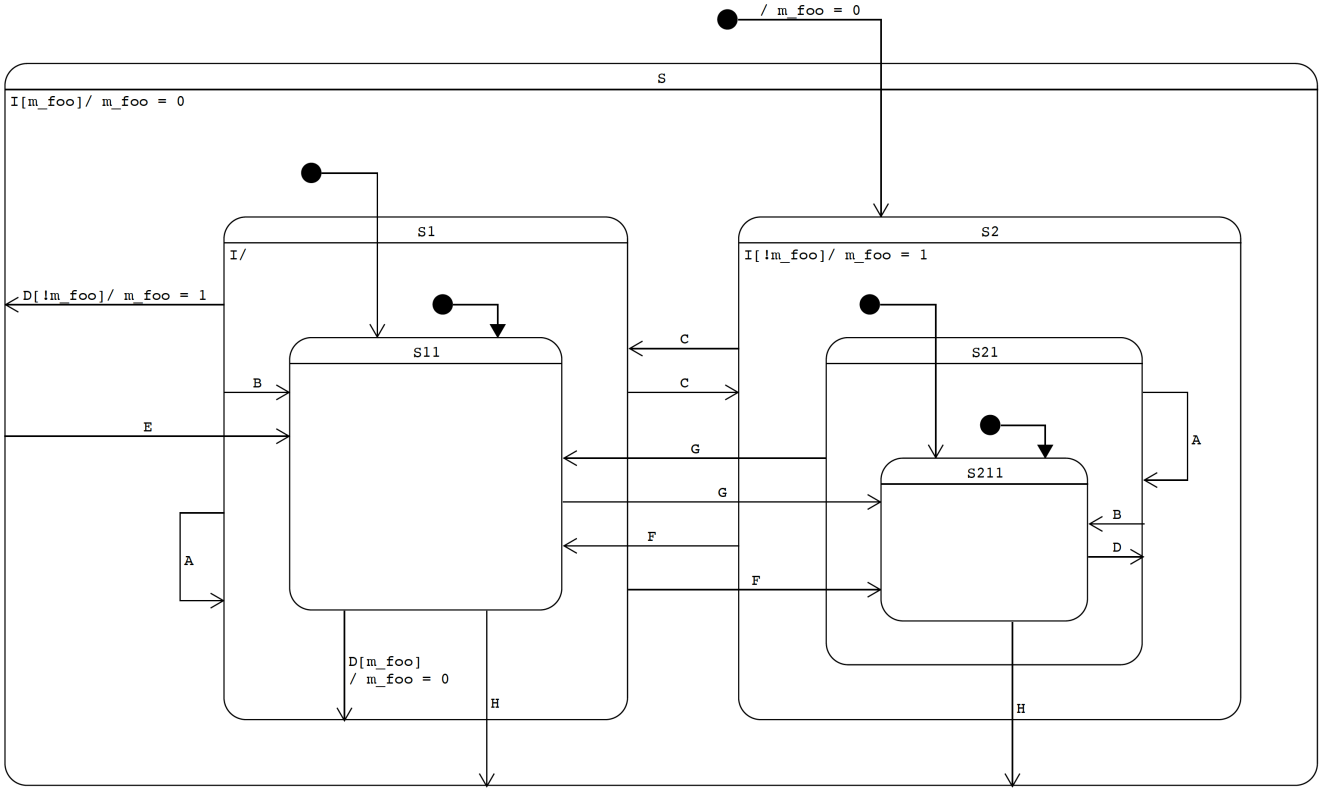
   c) **`Q_SUPER()`** - The event is not handled by this state. That is there is no enabled transition to be selected in this state. The immediate super state is specified as a pointer to member function. If this state is already at the highest level, the event will be propagated to the built-

Copyright (C) 2017-2022 Lawrence Lo. All rights reserved.

in *top* state (think of it as the paper) which automatically discards the event. You *must* make sure you return the correct super state via Q_SUPER() especially when you are copying-and-pasting code from an existing state; otherwise it would cause an assert at run-time as QP detects a malformed state machine.

3.3 Example

This example is based on Figure 2.11 on Page 88 of the PSiCC book.



It is implemented in the Demo active object. We haven't formally covered active objects yet, but now we just need to know an active object *is an* HSM running in its own thread.

See Src/app/Demo/Demo.cpp for the complete implementation. The state member function for S1 is extracted here for explanation.

```
QState Demo::S1(Demo * const me, QEvt const * const e) {
    switch (e->sig) {
        case Q_ENTRY_SIG: {
            EVENT(e);
            return Q_HANDLED();
        }
        case Q_EXIT_SIG: {
            EVENT(e);
            return Q_HANDLED();
        }
```

Copyright (C) 2017-2022 Lawrence Lo. All rights reserved.

```
        case Q_INIT_SIG: {
            return Q_TRAN(&Demo::S11);
        }
        case DEMO_A_REQ: {
            EVENT(e);
            return Q_TRAN(&Demo::S1);
        }
        case DEMO_B_REQ: {
            EVENT(e);
            return Q_TRAN(&Demo::S11);
        }
        case DEMO_C_REQ: {
            EVENT(e);
            return Q_TRAN(&Demo::S2);
        }
        case DEMO_D_REQ: {
            EVENT(e);
            if (!me->m_foo) {
                me->m_foo = 1;
                return Q_TRAN(&Demo::S);
            }
            break;
        }
        case DEMO_F_REQ: {
            EVENT(e);
            return Q_TRAN(&Demo::S211);
        }
        case DEMO_I_REQ: {
            EVENT(e);
            return Q_HANDLED();
        }
    }
    return Q_SUPER(&Demo::S);
}
```

Do you see the *direct mapping* between code and statechart? This is a simple yet powerful concept to translate design into code and vice versa. Do you have the experiences of trying to understand code written by someone else or even by yourself a few months back, with lots of variables set, clear and checked in many places?

Some sample log output is shown here for reference.

```
2890 CONSOLE_UART2> demo ?
[Commands]
test            Test function
a               A evt
b               B evt
c               C evt
d               D evt
e               E evt
f               F evt
g               G evt
h               H evt
```

Copyright (C) 2017-2022 Lawrence Lo. All rights reserved.

```
i               I evt
?               List commands
```

128825 CONSOLE_UART2> demo c
133344 DEMO(29): S2 DEMO_C_REQ from CONSOLE_UART2(2) seq=0
133344 DEMO(29): S211 EXIT
133344 DEMO(29): S21 EXIT
133344 DEMO(29): S2 EXIT
133344 DEMO(29): S1 ENTRY
133344 DEMO(29): S11 ENTRY

138788 CONSOLE_UART2> demo g
140892 DEMO(29): S11 DEMO_G_REQ from CONSOLE_UART2(2) seq=0
140892 DEMO(29): S11 EXIT
140892 DEMO(29): S1 EXIT
140892 DEMO(29): S2 ENTRY
140892 DEMO(29): S21 ENTRY
140892 DEMO(29): S211 ENTRY

141571 CONSOLE_UART2> demo h
143149 DEMO(29): S211 DEMO_H_REQ from CONSOLE_UART2(2) seq=0
143149 DEMO(29): S211 EXIT
143149 DEMO(29): S21 EXIT
143149 DEMO(29): S2 EXIT
143149 DEMO(29): S1 ENTRY
143149 DEMO(29): S11 ENTRY

143562 CONSOLE_UART2> demo d
145183 DEMO(29): S11 DEMO_D_REQ from CONSOLE_UART2(2) seq=0
145183 DEMO(29): S1 DEMO_D_REQ from CONSOLE_UART2(2) seq=0
145183 DEMO(29): S11 EXIT
145183 DEMO(29): S1 EXIT
145183 DEMO(29): S1 ENTRY
145183 DEMO(29): S11 ENTRY

145600 CONSOLE_UART2> demo d
147024 DEMO(29): S11 DEMO_D_REQ from CONSOLE_UART2(2) seq=0
147024 DEMO(29): S11 EXIT
147024 DEMO(29): S11 ENTRY

Copyright (C) 2017-2022 Lawrence Lo. All rights reserved.