Hierarchical State Machines - a Fundamentally Important Way of Design

Presented by
Madhukar Anand

Based on slides by Miro Samek

The Challenge of Event-Driven Systems

- Almost all computers today are **event-driven** systems
- The main programming challenge is to quickly pick and execute the **right** code in reaction to an event
- The reaction depends both on the nature of the event and on the current **context**, that is, the sequence of past events in which the system was involved
- Traditional "bottom up" approaches represent the context ambiguously by a multitude of variables and flags, which results in code riddled with a disproportionate number of convoluted conditional branches (`if-else` or `switch-case` statements in C/C++)

The Significance of "State"

- State machines make the response to an event explicitly dependent on both the nature of the event and the context of the system (**state**)
- **State** captures the relevant aspects of the system's history very efficiently

EXAMPLE: *a character code generated by a keyboard depends if the Shift has been depressed, but not on how many and which specific characters have been typed previously. A keyboard can be said to be in the "shifted" state or in the "default" state.*

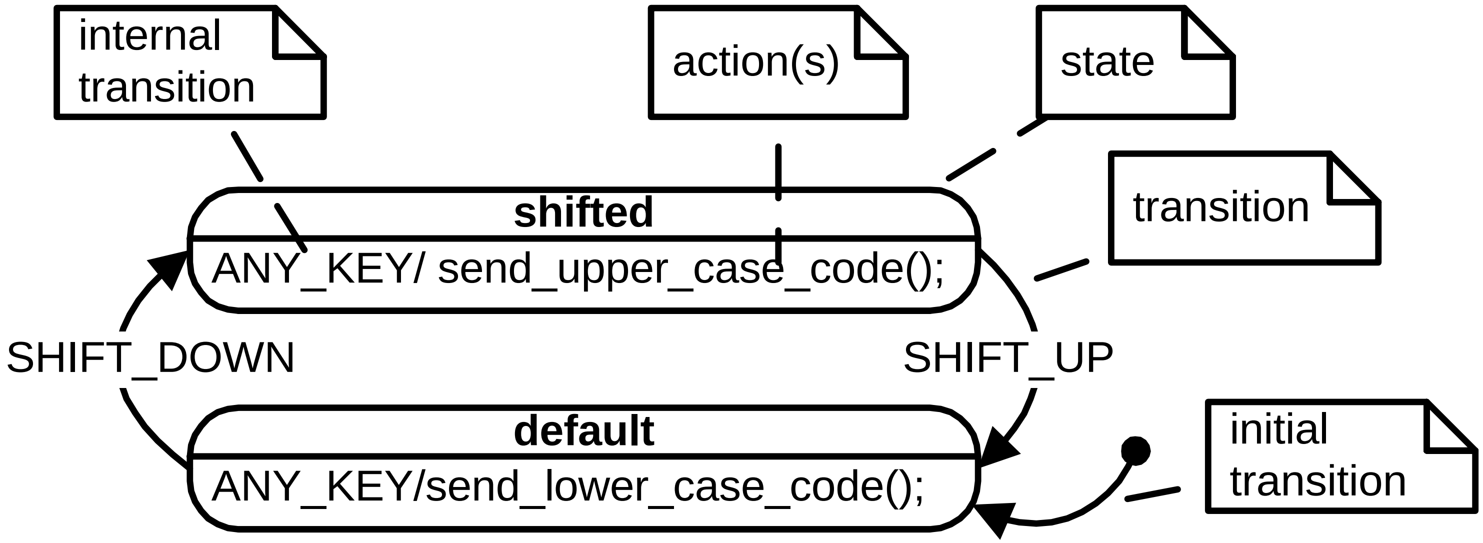
- A state can abstract away all possible (but irrelevant) event sequences and capture only the **relevant** ones

State Machines — Coding Perspective

- When properly represented in software, a state machine radically reduces the number of different paths though the code and simplifies the conditions tested at each branching point
- In all but the most basic coding technique (e.g., the `switch` statement) even the explicit testing of the "state variable" disappears as a conditional statement and is replaced by a table lookup or a function-pointer dereferencing
- This aspect is similar to the effect of polymorphism in OOP, which eliminates branching based on object's class

Visual Representation—State Diagrams

- State machines have a compelling and intuitive graphical representation in form of state diagrams
- State diagrams are directed graphs in which nodes denote states and connectors denote transitions
- The UML provides a standard notation and precise, rich semantics for state machines

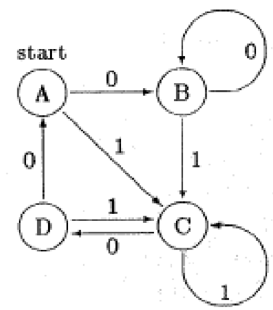


Translating a FSM : The wrong way

• The standard advice for those coding a finite state machine is to use a while loop, a case statement, and a state variable.

• This is bad, as the unstructured control transfers have been modeled in the code with assignments to variable state.

• The state variable serves as a goto statement, and the while and case statements obscure the underlying control structure.




```
type
    states = (A, B, C, D)
var
    ch: char { the input symbol };
    state: states;
begin { FSM-1 }
    state := A;
    while true do case state of
        A: begin
                write('A');
                read(ch);
                if ch = '0'
                then state := B
                else state := C
            end;
        B: begin
                write('B');
                read(ch);
                if ch = '0'
                then state := B
                else state := C
            end;
```

...

Jones, D. W. 1988. How (not) to code a finite state machine. *SIGPLAN Not.* 23, 8 (Aug. 1988), 19-22.

Translating a FSM : A better way

- Its preferable to admit that the original FSM was unstructured and eliminate the cosmetic control structures by replacing them with **goto** statements.

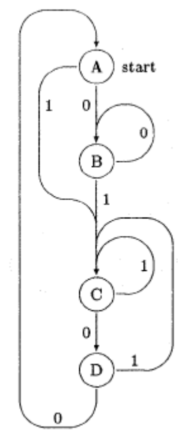
- This example illustrates the fact that any sequential procedure can be viewed as a FSM with the program counter serving as the state variable.

```
label { entries for machine states }
    A, B, C, D;
var
    ch: char { the input symbol };
begin { FSM-2 }
A: write('A');
    read(ch);
    if ch = '0' then goto B else goto C;
B: write('B');
    read(ch);
    if ch = '0' then goto B else goto C;
C: write('C');
    read(ch);
    if ch = '0' then goto D else goto C;
D: write('D');
    read(ch);
    if ch = '0' then goto A else goto C;
end { FSM-2 }
```

Jones, D. W. 1988. How (not) to code a finite state machine. *SIGPLAN Not.* 23, 8 (Aug. 1988), 19-22.

Translating a FSM : The best way

- The original FSM had reasonable structure.
- So, the best way is to redraw it using a technique similar to that used in a recursive transition network.
- Here, the loop entries come from above, iteration connections are drawn from the side, and loop exits are drawn from the bottom.




```
var
   ch: char { the input symbol };
begin { FSM-3 }
   repeat
      { state A }
      write('A');
      read(ch);
      while ch = '0' do begin
         { state B }
         write('B');
         read(ch);
      end { while };
      repeat
         repeat
            { state C }
            write('C');
            read(ch);
         until ch = '0';
         { state D }
         write('D');
         read(ch);
      until ch = '0';
   until false
end { FSM-3 }
```

Jones, D. W. 1988. How (not) to code a finite state machine. *SIGPLAN Not.* 23, 8 (Aug. 1988), 19-22.

The Limitations of Traditional FSMs

- The traditional FSMs tend to become unmanageable, even for moderately involved reactive systems (the “state-explosion” phenomenon)
- In practice, many states are similar, but classical FSMs have no means of capturing such commonalities and require repeating the same behavior in many states
- What’s missing in FSMs is a mechanism of factoring out the common behavior in order to **reuse** it across many states

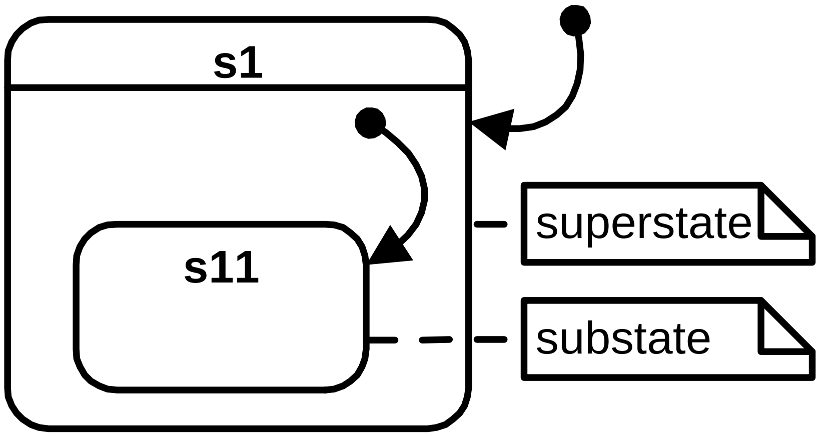
Introducing Statecharts

- Statecharts (invented by David Harel in the 1980's, [Harel 87]) provide exactly what's been missing in classical FSMs: a way of capturing the common behavior in order to reuse it across many states
- The most important innovation of statecharts is the introduction of hierarchically **nested states**
- The UML 1.4 state machines [OMG 01] are an object-based variant of Harel statecharts [Harel 87]. They incorporate several concepts similar to those defined in ROOMcharts, a variant of statechart defined in the ROOM modeling language [Selic+ 94].

The Semantics of State Nesting

- If a system *is in* the nested state `s11` (called substate), it also (implicitly) *is in* the surrounding state `s1` (called superstate)

- Any event is first handled in the context of substate `s11`, but all unhandled events are automatically passed over to the next level of nesting (`s1` superstate)



- The substates need only define the **differences** from the superstates, and otherwise can easily share (**reuse**) behavior defined in higher levels of nesting

Programming By Difference

- State nesting lets you define a new state rapidly in terms of an old one, by reusing the behavior from the parent state
- State nesting allows new states to be specified **by difference** rather than created from scratch each time
- State nesting lets you get new behavior almost for free, **reusing** most of what is common from the superstates
- The fundamental character of state nesting comes from the combination of hierarchy and programming-by-difference, which is otherwise known in software as **inheritance**
- State nesting leads to **behavioral inheritance** [Samek+ 00, 02]

Liskov Substitution Principle for States

- Liskov Substitution Principle (LSP) is a universal law of generalization. In the traditional formulation for classes LSP requires that a subclass can be freely substituted for its superclass
- Because behavioral inheritance is just a specific kind of inheritance, the LSP can (and should) be applicable to nested states as well as classes
- LSP generalized for states means that the behavior of a substate should be consistent with the superstate
- Compliance with the LSP (for states) allows you to build better (correct) state hierarchies that make efficient use of abstraction

Guaranteed Initialization and Cleanup

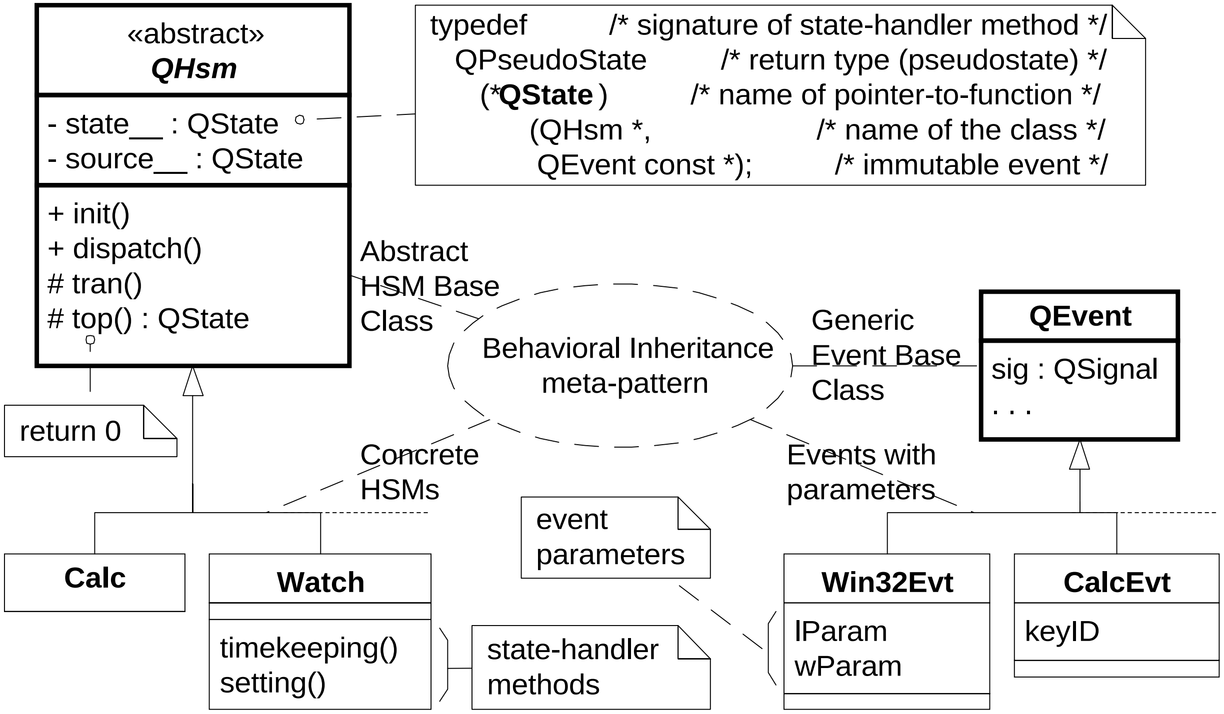
- UML state machines allow states to have optional entry actions executed automatically upon the entry to the state and exit actions executed upon the exit
- The value of entry and exit actions is that they provide means for guaranteed initialization and cleanup, much like class constructors and destructors in OOP
- Entry and exit actions are particularly important and powerful in conjunction with the state hierarchy, because they determine the **identity** of the hierarchical states
- The order of execution of entry actions must always proceed from the outermost state to the innermost state. The execution of exit actions proceeds in exact opposite order

Implementing HSMs

- The goal of this HSM implementation is to provide a **minimal** and generic event-processor that you can use with any event queuing and dispatching mechanism.
- This HSM implementation addresses only:
  - Nested states with full support for behavioral inheritance,
  - Guaranteed initialization and cleanup with state entry and exit actions, and
  - Support for specializing state models via class inheritance.
- The strategy is to provide just enough (but not more!) truly fundamental elements to allow for the efficient construction of all other (higher level) statechart features, including those bundled into the UML specification.

Structure of the HSM Implementation

- All concrete state machines derive from the abstract `QHsm` base class
- "State" (`QState`) is represented as *pointer-to-member-function* of the `QHsm` class



- All events are instances of `QEvent` class, or subclasses of `QEvent` (for events with parameters).

The `QHsm` Base Class

- The `QHsm` base class provides the following methods:
  - `init()` to trigger the topmost initial transition.
  - `dispatch()` to dispatch an event for processing according to the state machine semantics
  - `tran()` for taking a state transition
- Clients derive concrete state machines from the `QHsm` class
- Clients add behavior by adding state handler methods to the `QHsm` subclass
- Clients call `QHsm::init()` method once
- Clients call `QHsm::dispatch()` repetitively for each event

State Handlers

- A state handler method takes immutable pointer to `QEvent` (`QEvent const *`) and returns a pointer to the superstate handler if it doesn't handle the event, or `NULL` if it does.
- State handlers use internally the `QHsm` method `tran()` to code state transitions. Transitions are coded in the source state.
- The signature of state handler is determined by the `QState` pointer-to-function (pointer-to-member-function in C++):

```
typedef QPseudoState (*QState)(QHsm *, QEvent const*); // C
typedef QPseudoState (QHsm::*QState)(QEvent const*); // C++
```

- C / C++ doesn't allow to define strictly recursive signature

Dispatching Events – QHsmDispatch()

- QHsmDispatch() traverses the state hierarchy starting from the current state (me->state__):

```
void QHsmDispatch(QHsm *me, QEvent const *e) {
  for (me->source__ = me->state__; me->source__ != 0;
       me->source__ = (QState)(*me->source__)(me, e))
  {}
}
```

- At each level QHsmDispatch() passes the event to the corresponding state handler method
- The processing ends when some state handler handles the event (returns NULL)
- The top state (defined in QHsm) always returns NULL

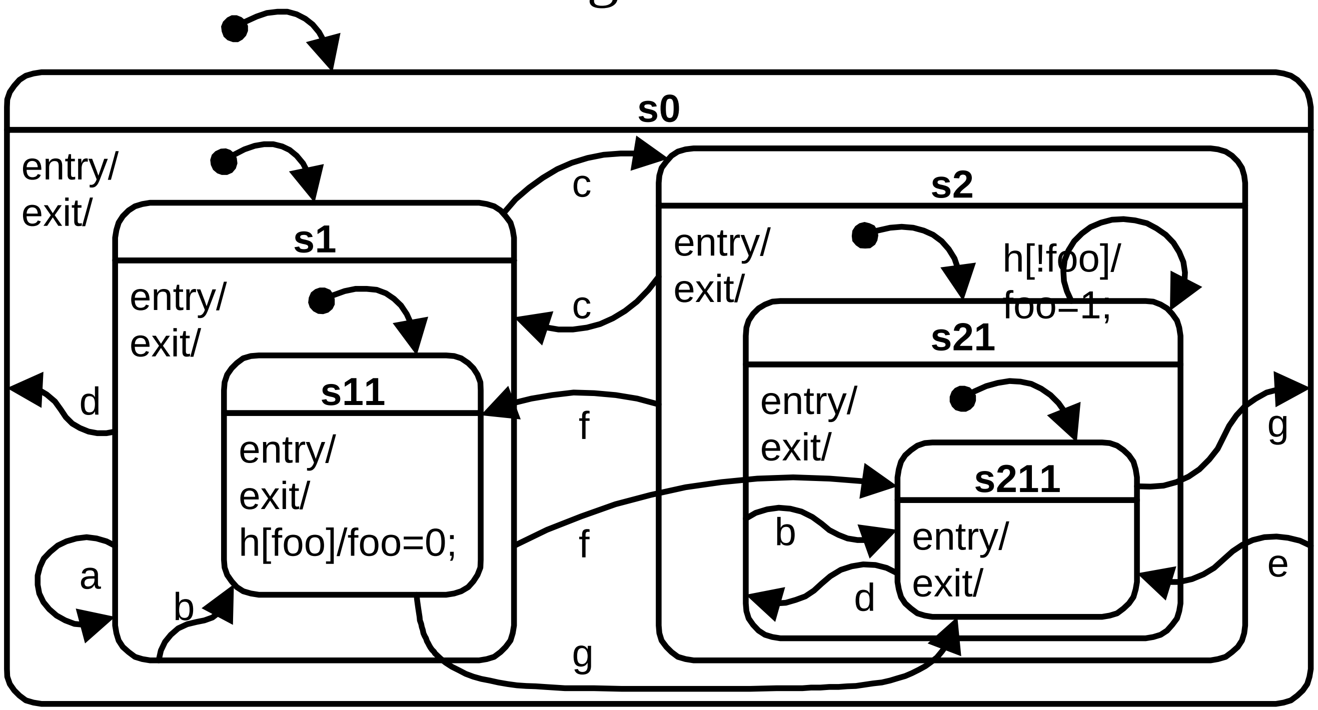
QSignal and QEvent

```
typedef unsigned short QSignal;
struct QEvent {
   QSignal sig;      /* signal of the event instance */
   /* ...    other QEvent attributes not shown here */
};
enum {                              /* reserved signals */
   Q_INIT_SIG = 1, Q_ENTRY_SIG, Q_EXIT_SIG,
   Q_USER_SIG       /* the first signal free to use */
};
```

- The `sig` attribute of `QEvent` conveys the type of the event (what happened).
- Signals must be of a scalar type and are typically enumerated. The four lowest signals are reserved.
- Event parameters are added by deriving new event classes from `QEvent`

Annotated Example

- Let's code in C the following non-trivial HSM:



- The state machine has six states `s0`, `s1`, `s11`, `s2`, `s21`, and `s211`, and its alphabet consists of eight signals: `a` through `h`.

Subclassing QHsm (in C)

```
typedef struct QHsmTst QHsmTst;
struct QHsmTst {             /* QhsmTst state machine */
   QHsm super_;                          /* extends QHsm */
   int foo__;     /* private extended state variable */
};
QHsmTst *QHsmTstCtor(QHsmTst *me);
void QHsmTst_initial(QHsmTst *me, QEvent const *e);
QSTATE QHsmTst_s0(QHsmTst*me, QEvent const *e);
QSTATE QHsmTst_s1(QHsmTst*me, QEvent const *e);
QSTATE QHsmTst_s11(QHsmTst*me, QEvent const *e);
QSTATE QHsmTst_s2(QHsmTst*me, QEvent const *e);
QSTATE QHsmTst_s21(QHsmTst*me, QEvent const *e);
QSTATE QHsmTst_s211(QHsmTst*me, QEvent const *e);
```

- Declare the constructor, initial pseudostate and all six state handler methods (the unusual indentation indicates state nesting)

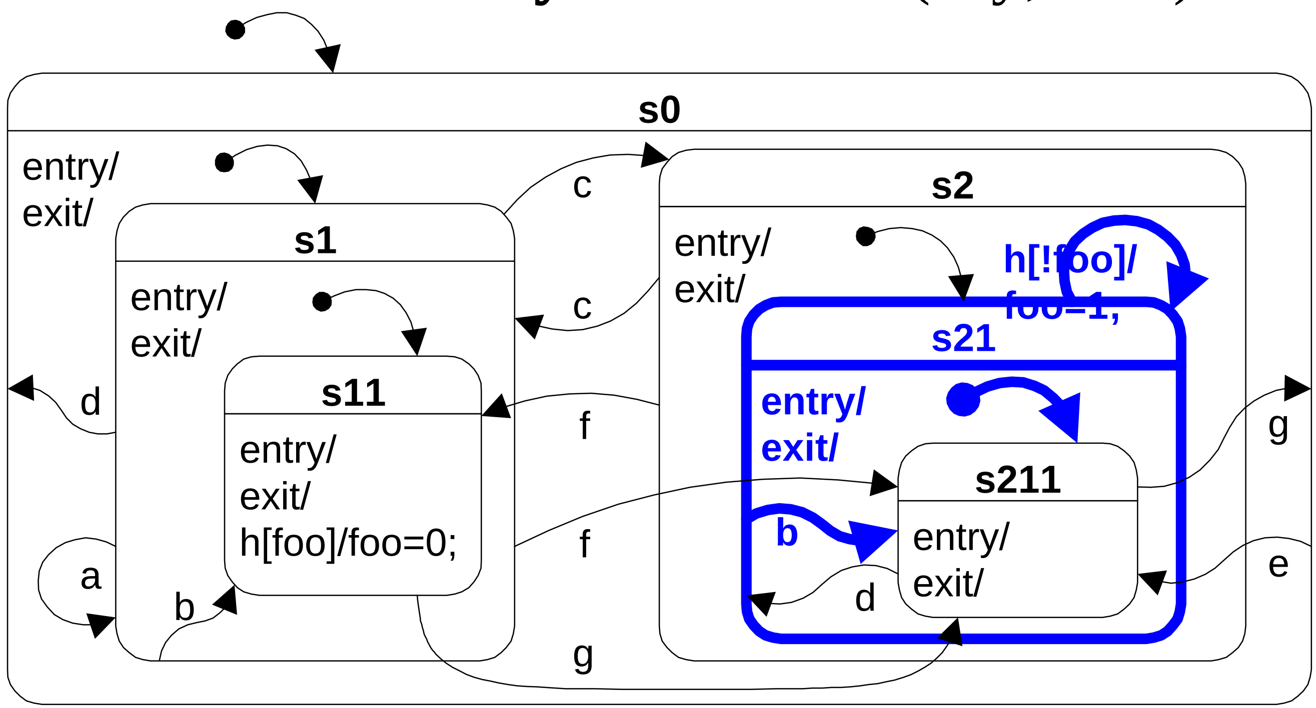
The Constructor and Initial Pseudostate

```
QHsmTst *QHsmTstCtor(QHsmTst *me) {
   QHsmCtor_(&me->super_, /* construct the superclass */
             (QPseudoState)QHsmTst_initial);
   return me;
}

void QHsmTst_initial(QHsmTst *me, QEvent const *e) {
   printf("top-INIT;");
   me->foo__ = 0;    /* init. extended state variable */
   Q_INIT(QHsmTst_s0);   /* the topmost initial tran. */
}
```

- In C you need to explicitly construct the superclass `QHsm`
- In the initial pseudostate you must take the initial transition via the macro `Q_INIT()`

What Elements Go Into a State Handler?

- To find out which elements go to a given state handler, you follow around the **boundary** of the state (say, `s21`) in the diagram



- You need to include: all transitions originating at the boundary, entry actions, exit actions, internal transitions, and the initial transition

Coding a State Handler

- Each state maps to a state handler method. For example, state `s21` maps to `QHsmTst_s21()` state handler.
- All state handler methods have the same skeleton (housekeeping code, [Douglass 99])

```
QSTATE QHsmTst_s21(QHsmTst *me, QEvent const *e) {
   switch (e->sig) { /* demultiplex events based on signal */
      /* . . . */
   }
   return (QSTATE)QHsmTst_s2;  /* designate the superstate */
}
```

Coding Entry and Exit Actions

- You intercept the reserved signals `Q_ENTRY_SIG` or `Q_EXIT_SIG`, enlist actions you want to execute, and terminate with "`return 0`" (event handled)

```
QSTATE QHsmTst_s21(QHsmTst *me, QEvent const *e) {
   switch (e->sig) { /* demultiplex events based on signal */
   case Q_ENTRY_SIG: printf("s21-ENTRY;"); return 0;
   case Q_EXIT_SIG: printf("s21-EXIT;");   return 0;
      /* . . . */
   }
   return (QSTATE)QHsmTst_s2;  /* designate the superstate */
}
```

Coding the Initial Transition

- You intercept the reserved signal `Q_INIT_SIG`, enlist the actions, and then designate the target substate through the macro `Q_INIT()` , after which you exit state handler with "`return 0`" (event handled)

```
QSTATE QHsmTst_s21(QHsmTst *me, QEvent const *e) {
   switch (e->sig) { /* demultiplex events based on signal */
   /* . . . */
   case Q_INIT_SIG:  /* intercept the reserved init signal */
       printf("s21-INIT;");
       Q_INIT(QHsmTst_s211);     /* designate the substate */
       return 0;                                 /* event handled */
   /* . . . */
   }
   return (QSTATE)QHsmTst_s2;  /* designate the superstate */
}
```

Coding a Regular Transition

- You intercept the custom defined signal (e.g., `B_SIG`), enlist the actions, and then designate the target state through the macro `Q_TRAN()` , after which you exit state handler with "`return 0`" (event handled)

```
QSTATE QHsmTst_s21(QHsmTst *me, QEvent const *e) {
   switch (e->sig) { /* demultiplex events based on signal */
   /* . . . */
   case B_SIG:                 /* intercept the custom signal */
      printf("s21-B;");
      Q_TRAN(QHsmTst_s211);  /* designate the target state */
      return 0;                                /* event handled */
   /* . . . */
   }
   return (QSTATE)QHsmTst_s2;  /* designate the superstate */
}
```

Coding a Transition With a Guard

- You intercept the custom defined signal (e.g., `H_SIG`), and you immediately test the guard inside an `if (…)`. If the guard evaluates FALSE you break to return the superstate.

```
QSTATE QHsmTst_s21(QHsmTst *me, QEvent const *e) {
   switch (e->sig) { /* demultiplex events based on signal */
   case H_SIG:                    /* self transition with a guard */
      if (!me->foo__) {              /* test the guard condition */
         printf("s21-H;");
         me->foo__ = !0;
         Q_TRAN(QHsmTst_s21);                   /* self transition */
         return 0;
      }
      break;                                  /* event not handled */
   }
   return (QSTATE)QHsmTst_s2;  /* designate the superstate */
}
```

The Complete s21 State Handler

```
QSTATE QHsmTst_s21(QHsmTst *me, QEvent const *e) {
   switch (e->sig) {
   case Q_ENTRY_SIG: printf("s21-ENTRY;"); return 0;
   case Q_EXIT_SIG: printf("s21-EXIT;");   return 0;
   case Q_INIT_SIG: printf("s21INIT;");
                    Q_INIT(QHsmTst_s211);  return 0;
   case B_SIG: printf("s21-B;");
                    Q_TRAN(QHsmTst_s211);  return 0;
   case H_SIG:           /* self transition with a guard */
      if (!me->foo__) {     /* test the guard condition */
         printf("s21-H;");
         me->foo__ = !0;
         Q_TRAN(QHsmTst_s21);         /* self transition */
         return 0;
      }
      break;            /* break to return the superstate */
   }
   return (QSTATE)QHsmTst_s2; /* return the superstate */
}
```

Test Harness

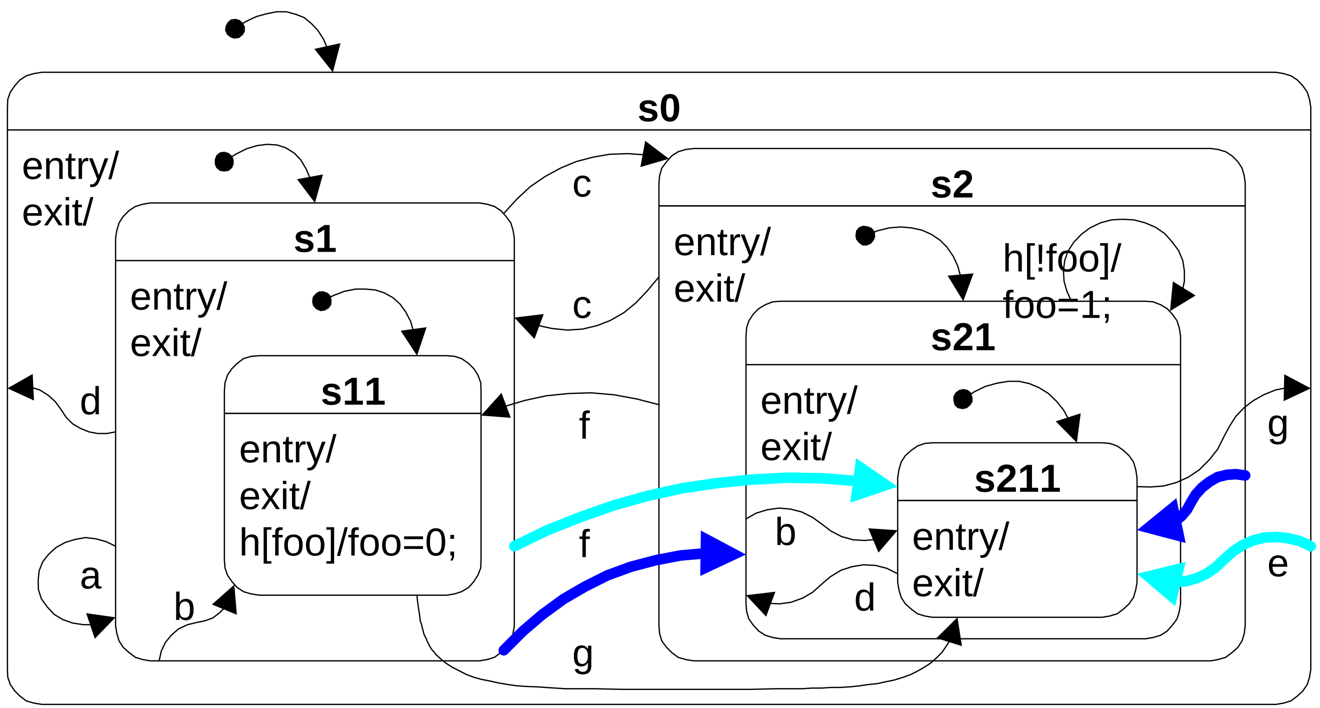
```
#include "qhsm.h“          /* include the HSM interface */
static QHsmTst test;             /* instantiate the HSM */

int main() {
   printf("QHsmTst example, version 1.00, libraries: %s\n",
           QHsmGetVersion());
   QHsmTstCtor(&test); /* explicitly construct the HSM */
   QHsmInit((QHsm *)&test, 0);   /* initial transition */
   for (;;) {
      char c;
      printf("\nSignal<-");
      c = getc(stdin);
      getc(stdin);                          /* discard '\n' */
      if (c < 'a' || 'h' < c) {
         return 0;
      }
      QHsmDispatch((QHsm *)&test, &testQEvt[c - 'a']);
   }
   return 0;
}
```

An Example Session

```
 1: QHsmTst example, version 1.00, libraries: QHsm 2.2.5
 2: top-INIT;s0-ENTRY;s0-INIT;s1-ENTRY;s1-INIT;s11-ENTRY;
 3: Signal<-a
 4: s1-A;s11-EXIT;s1-EXIT;s1-ENTRY;s1-INIT;s11-ENTRY;
 5: Signal<-e
 6: s0-E;s11-EXIT;s1-EXIT;s2-ENTRY;s21-ENTRY;s211-ENTRY;
 7: Signal<-e
 8: s0-E;s211-EXIT;s21-EXIT;s2-EXIT;s2-ENTRY;s21-NTRY;
    s211-ENTRY;
 9: Signal<-a
10:
11: Signal<-h
12: s21-H;s211-EXIT;s21-EXIT;s21-ENTRY;s21-INIT;s211-ENTRY;
13: Signal<-h
14:
15: Signal<-x
```

Changing the State Machine

EXERCISE: modify the state machine by moving transition `'e'` from `s0` to `s2`, and by changing target of transition `'f'` in state `s1` from `s211` to `s21`. Test the modified HSM.



Summary

- You can quite easily (once you know the pattern) implement HSMs in C and C++. In fact, coding a non-trivial HSM turned out to be an exercise in following a few simple rules.
- With just a bit of practice, you will forget that you are "translating" state models into code; rather, you will directly code state machines in C or C++, just as you directly code classes in C++ or Java.
- At this point, you will no longer struggle with convoluted `if-else` statements and gazillions of flags. You will start *thinking* at a higher level of abstraction.
- Thus, a sufficiently small and truly practical implementation of statecharts can trigger a **paradigm shift** in your way of thinking about programming reactive systems. I call this paradigm shift Quantum Programming (QP) [Samek 02].

Discussion /Criticism from Users

- Run to completion (RTC) semantics.
  - Avoids internal concurrency issues. But it is not good for ensuring timely response into higher priority interrupts.
- Can different timing semantics be captured with such a framework?
  - For instance, how easy is to encode temporal scopes for states?
- How hard is debugging?
  - “Run-to-completion“ semantics results in excessive self-posting of events and queuing.

Discussion /Criticism from Users

- “There is no real value in separating semantics (state-chart description) from functionality (code which uses the state-machine), but this turned out to be a maintenance nightmare” - Amazon review.
- QP does not offer orthogonal states, which is needed to model concurrent aspects of a system.
  - Somewhat salvaged by the publish/subscribe framework but still clumsy.

Frank Schuhardt, http://www.amazon.com/gp/pdp/profile/A2IA4GCO91XW95/

References

- [Beck 00] Beck, Kent, *Extreme Programming Explained,* Addison-Wesley, 2000.
- [Douglass 99] Douglass, Bruce Powel, *Doing Hard Time, Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns.* Addison-Wesley, 1999.
- [Duby 01] Duby, Carolyn, "Class 203: Implementing UML Statechart Diagrams", *Proceedings of Embedded Systems Conference, San Francisco* 2001.
- [Ganssle 98] Ganssle, Jack G., "The Challenges of Real-Time Programming", *Embedded Systems Programming*, July 1998 pp. 20-26.
- [Gamma+ 95] Gamma, Erich, et al., *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
- [Harel 87] Harel, David, "Statecharts: A Visual Formalism for Complex Systems", *Science of Com-puter Programming*, 8, 1987, pp. 231-274.
- [Labrosse 99] Labrosse, Jean J., *MicroC/OS-II, The Real-Time Kernel*. R&D Publ., 1999.
- [OMG 01] Object Management Group, Inc., *OMG Unified Modeling Language Specification v1.4*, http://www.omg.org, September 2001.
- [Samek+ 00] Samek, Miro and Paul Y. Montgomery, “State-Oriented Programming”, *Embedded Systems Programming*, August 2000 pp. 22-43
- [Samek 02] Samek, Miro, *Practical Statecharts in C/C++: Quantum Programming for Embedded Systems*, CMP Books, 2002, ISBN: 1-57820-110-1.
- [Selic+ 94] Selic, Bran, Garth Gullekson, and Paul. T. Ward, *Real-Time Object Oriented Modeling*, John Wiley & Sons, 1994.