Practical UML Statecharts

Miro Samek
miro@quantum-leaps.com
Dr. Miro Samek is the author of "Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems" (Newnes 2008), has written numerous articles for magazines, including a column for C/C++ Users Journal, is a regular speaker at the Embedded Systems Conferences, and serves on the editorial review board of the Embedded Systems Design magazine. For a number of years, he worked in various Silicon Valley companies as an embedded software architect and before that he worked as an embedded software engineer at GE Medical Systems (now GE Healthcare).

Contact: miro@quantum-leaps.com
Outline

Event-Driven Programming

• Hierarchical state machines
• Real-time frameworks
• Questions & Answers
Most computer systems are event-driven
Event-Driven System Example: Vending Machine
Traditional Sequential Program Flow

Program flow determined by sequence of instructions

```
/* wait for Button 1 press */
while (Button1_GPIO != DEPRESSED) {
  
}
```

```
/* wait for Button 2 press */
OSSemPend (&Button 2_Semaphore, ...);
```

FLOWCHART

START

Function 1

Function 2

Function 3

Function 4
Event-Driven Program Flow

Program flow determined by order of events

- Event loop
- ISR1()
- ISR2()
- Event queue (FIFO)
- Event dispatcher
- Event dispatching based on event type
- Queue empty
- Application code
- Event
- EventN_handler();
- Event2_handler();
- Event1_handler();
- Idle_handler();
Event-driven program flow (cont'd)

- Events are first-class objects
- Events are processed asynchronously
- Events are processed in Run-to-Completion (RTC) fashion
- Events are queued

```
Event
  signal : uint16_t

ADCEvent
  data : uint16_t

EthEvent
  payload[1024] : uint8_t
  length : uint16_t
```
Challenges of event-driven programming

(a) wait4eventA(); /* process A */
while (...) {
    wait4eventB(); /* process B */
}
wait4eventC(); /* process C */

(b) event A arrived
    process A
    else
    event B arrived
    process B
    else
    event C arrived
    process C

(c) process_A()
    process_B()
    process_C()

Where is the structure?

- Hidden in static variables
- "spaghetti" code inside event handlers
Event-action paradigm—spaghetti code

Dim Op1, Op2 ' Previously input operand.
Dim DecimalFlag As Integer ' Decimal point present yet?
Dim NumOps As Integer ' Number of operands.
Dim LastInput ' Indicate type of last keypress event.
Dim OpFlag ' Indicate pending operation.
Dim TempReadout

Private Sub Operator_Click(Index As Integer)
    TempReadout = Readout
    If LastInput = "NUMS" Then
        NumOps = NumOps + 1
    End If
    Select Case NumOps
    Case 0
        If Operator(Index).Caption = "-" And LastInput <> "NEG" Then
            Readout = "-" & Readout
            LastInput = "NEG"
        End If
    Case 1
        Op1 = Readout
        If Operator(Index).Caption ="-" And LastInput <> "NUMS"
            And OpFlag <> "=" Then
            Readout = "-"
            LastInput = "NEG"
        End If
    End Select
End Sub

Bunch of flags and variables

Complex conditional code based on the flags and variables
Outline

• Event-driven programming

Hierarchical state machines

• Real-time frameworks

• Q & A
State machine

- Event-action paradigm applied locally within each state
Recovering the structure with a state machine

```c
wait4eventA();
/* process A */
while (...) {
  wait4eventB();
  /* process B */
}
wait4eventC();
/* process C */
```

```c
while (...) {
  wait4eventB();
  /* process B */
}
wait4eventC();
/* process C */
```

```
while (...) {
  wait4eventB();
  /* process B */
}
wait4eventC();
/* process C */
```
**Statecharts vs. Flowcharts**

Completely distinct: different use of CPU!

- **Statechart: on the arrows**
  - Otherwise CPU idle

- **Flowchart: in the boxes**
  - CPU never idle

(a) 

```
E1 / action1();
E2 / action2();
E3 / action3();
```

(b) 

```
do X
   do Y
   do Z
   do W
```
State-transition explosion

OPER

operand1

DIGIT_0_9, POINT

result

DIGIT_0_9, POINT

operand2

OFF

EQUALS

OFF

OFF

OFF

OFF
Reuse of behavior through state nesting

- Programming by difference
  - Behavioral inheritance
State entry and exit actions

- Guaranteed initialization and cleanup
- Superstates are entered before substates
  - like class constructors in OOP
- Superstates are exited after substates
  - like class destructors in OOP
UML state machine semantics—QHsmTst example

```
QHsmTst example, built on Sep 25 2007 at 09:11:31,
QEP/C: 3.4.01.
Press ESC to quit...
top:INIT;s-ENTRY;s2-ENTRY;s2-INIT;s21-ENTRY;s211-ENTRY;
1:  s1-I;
2:  s1-A;s11-EXIT;s1-ENTRY;s1-INIT;s11-ENTRY;
3:  s1-D;s11-EXIT;s1-ENTRY;s1-INIT;s11-ENTRY;
4:  s1-C;s11-EXIT;s1-ENTRY;s1-INIT;s11-ENTRY;
5:  s1-E;s111-EXIT;s1-ENTRY;s1-INIT;s111-ENTRY;
6:  s1-D;s111-EXIT;s1-ENTRY;s111-ENTRY;
7:  s1-E;s11-EXIT;s1-ENTRY;s1-INIT;s11-ENTRY;
8:  s1-A;s11-EXIT;s1-ENTRY;s1-INIT;s11-ENTRY;
9:  s1-G;s11-EXIT;s1-ENTRY;s1-INIT;s11-ENTRY;
10: s1-I;
11: s2-I;
12: s-I;
13: Bye, Bye!
```

Diagram:
- States: s, s2, s1, s11, s21, s211
- Transitions:
  - Entry/Exit:
  - Transition A, B, C, D, E, F, G

Note: The diagram and textual example illustrate state machine semantics with specific transitions and state entries/exports.
Coding a HSM in QP/C++

```cpp
QState Calc::on(Calc *me, QEvent const *e) {
    switch (e->sig) {
        case Q_ENTRY_SIG: { // entry action
            BSP_message("on-ENTRY");
            return Q_HANDLED();
        }
        case Q_EXIT_SIG: { // exit action
            BSP_message("on-EXIT");
            return Q_HANDLED();
        }
        case Q_INIT_SIG: { // initial transition
            BSP_message("on-INIT");
            return Q_TRAN(&Calc::ready);
        }
        case C_SIG: { // state transition
            BSP_clear(); // clear the display
            return Q_TRAN(&Calc::on);
        }
        case OFF_SIG: { // state transition
            return Q_TRAN(&Calc::final);
        }
    }
    return Q_SUPER(&QHsm::top); // superstate
}
```
Outline

• Event-driven programming
• Hierarchical state machines
Real-time frameworks
• Q & A
Problems with the simple event-loop

- Single event queue prevents prioritization of work
- Event dispatcher is incompatible with state machines
Slicing by event-signal destroys the notion of state

Event-handlers

States

“off”

“on”

Events

OFF_SIG

ON_SIG

S

ta

E

v

e

n
-

h

d
l

a

r

p

s

ers

Events

ON_SIG

OFF_SIG

States

“off”

“on”

Events

{ powerOn (), “on” }

{ powerOff (), “off” }

{ powerOn (), “on” }

{ powerOff (), “off” }
State machine framework based on cooperative kernel

- Use multiple priority queues bound to state machines
- Don’t sort events based on the signal (vertical slicing)
State machine framework based on preemptive kernel

- RTC does not mean that state machines cannot preempt each other
- Each state machine executes in its own thread of control
  \( (\text{State Machine} + \text{Event Queue} + \text{Thread}) = \text{Active Object} \)
Minimal active object framework (QP)

- **QP event processor**
  - `state : QHsmState`
  - `init()`
  - `dispatch()`
  - Configurable to derive from other classes with the compatible interface

- **QF real-time framework**
  - `thread`
  - `eQueue`
  - `prio`
  - `start()`
  - `postFIFO()`
  - `postLIFO()`
  - `thread`
  - `eQueue`
  - `Queue`

- **QEvent**
  - `sig : QSignal`
  - `dynamic_ : uint8_t`

- **QTimeEvt**
  - `ctr`
  - `postIn()`
  - `postEvery()`
  - `disarm()`
  - `rearm()`

- **"Star Wars" application**
  - **Ship**
  - **Missile**
  - **Tunnel**
  - **Mine1**
    - `n`
  - **Mine2**
    - `n`

- **ObjectPosEvt**
- **ObjectImageEvt**
Summary

State machines complement imperative languages (C, C++, Java, C#, etc.)

State machines “explode” without state hierarchy

State machines are impractical without a framework

Once you try an event-driven, state machine framework you will not want to go back to “spaghetti” and raw RTOS/OS

Welcome to the 21 century!
Outline

• Event-driven programming
• Hierarchical state machines
• Real-time frameworks

Questions & Answers