State Machines and Statecharts

Part 2
State Machines

Bruce Powel Douglass, Ph.D.
All the best lies are actually true!
Agenda

- Approach taken for this talk
- Quick Overview of Finite State Machines
- Quick Overview of Harel Statecharts
- Advanced Statechart features
- Other State Notations
Approach taken for this talk

- This is meant to be a gentle introduction to states and state machines
- This section will be
  - mostly on advanced features of statecharts
  - other state representations
- Ask questions if you don’t think your neighbor is understanding
Finite State Machine Review

- What’s a STATE?
- What’s a TRANSITION?
- What are the three classes of behavior?
- What kinds of things have state?
- Why model states?
Simple Example

- **Idle**
  - \textit{Message Ready} / \textit{Start Timer}
  - \textit{Valid ACK}
  - \textit{tm(Wait Time)} [\textit{Transmit Count} <= \textit{Limit}]
- **Waiting**
  - \textit{tm(Wait Time)} [\textit{Transmit Count} > \textit{Limit}]
- **Sending**
  - \textit{Done} / \textit{Transmit Count++}
  - \textit{Invalid ACK}
Advanced Statechart Features

- Conditional Transitions
- Orthogonal Components
- Concurrency
- Broadcast transitions
- Inherited state behavior
Conditional Transitions

- **Uploading_Data**
  - \[\text{CMDID}(msg) \equiv \text{Upload}\]

- **Processing_Command**
  - \[\text{CMDID}(msg) \equiv \text{Update}\]
  - \[\text{CMDID}(msg) \equiv \text{Store}\]

- **Updating_Parameters**

- **Error**
  - \[\text{isValid}(msg)\]

- **Message_Received**

- **Receiving**
Orthogonal Components

- Model state behavior for independent aspects of objects
- Can be used to model
  - concurrency
  - independent attributes
- Simplify state diagrams by reducing “state explosion”
Orthogonal Components

```
myInstance: myClass

<table>
<thead>
<tr>
<th>tColor</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>ErrorStatus</td>
</tr>
<tr>
<td>tMode</td>
<td>Mode</td>
</tr>
</tbody>
</table>
```

enum tColor {eRed, eBlue, eGreen};

enum boolean {TRUE, FALSE}

enum tMode {eNormal, eStartup, eDemo}

How do you draw the state of this object?
**Approach 1: Enumerate all**

<table>
<thead>
<tr>
<th>eRed, FALSE, eDemo</th>
<th>eBlue, FALSE, eDemo</th>
<th>eGreen, FALSE, eDemo</th>
</tr>
</thead>
<tbody>
<tr>
<td>eRed, TRUE, eDemo</td>
<td>eBlue, TRUE,</td>
<td>eGreen, TRUE,</td>
</tr>
<tr>
<td></td>
<td>eDemo</td>
<td></td>
</tr>
<tr>
<td>eRed, FALSE, eNormal</td>
<td>eBlue, FALSE,</td>
<td>eGreen, FALSE,</td>
</tr>
<tr>
<td></td>
<td>eNormal</td>
<td></td>
</tr>
<tr>
<td>eRed, TRUE, eNormal</td>
<td>eBlue, TRUE,</td>
<td>eGreen, TRUE,</td>
</tr>
<tr>
<td></td>
<td>eNormal</td>
<td></td>
</tr>
<tr>
<td>eRed, FALSE, eStartup</td>
<td>eBlue, FALSE,</td>
<td>eGreen, FALSE,</td>
</tr>
<tr>
<td></td>
<td>eStartup</td>
<td></td>
</tr>
<tr>
<td>eRed, TRUE, eStartup</td>
<td>eBlue, TRUE,</td>
<td>eGreen, TRUE,</td>
</tr>
<tr>
<td></td>
<td>eStartup</td>
<td></td>
</tr>
</tbody>
</table>
Approach 2
What is Concurrency?

- Concurrency is *the simultaneous execution of program statements within a system*

- Types:
  - Pseudo-concurrency (Single CPU)
  - True Concurrency (Multiple CPUs)
Pseudo-Concurrency

- Heavy-weight (process)
  Each process has its own data and code space

- Light-weight (thread)
  Each thread shares a common data and code space
Synchronization Models

- Sharing data
  - Shared variables
  - Message passing

- Types of synchronization
  - Synchronous
  - Asynchronous
  - Balking
  - Timeout
Synchronization Models

- Operations may be
  - Guarded
  - Synchronous
  - Simple (i.e. function calls)

- Events imply
  - Asychronicity
  - Event queues
UML Concurrency

- Each thread is based from a single “active” object
- All components of the active object inherit the composite’s thread
- Each thread must have its own event queue
Concurrent Statecharts

- Many embedded systems consist of multiple threads, each running an FSM
- State charts allow the modeling of these parallel threads
Concurrent State Charts

- States S and T are active at the same time as long as X is active.
  - Either S.A or S.B must be active when S is active
  - Either T.C, T.D, or T.E must be active when T is active
Concurrent State Charts

- When X exits, both S and T exit
  - If S exits first, the FSM containing X must wait until T exits
  - If the two FSMs are always independent, then they must be enclosed at the highest scope
Explicit Synchronization

Starting

A → B
A → D
D → E
D → F

B → A
B → C
C → B
C → E
C → G

D → E
D → G
E → F

C → E
C → F

F → E
F → G

E → C
G → C

explicit control
branching

explicit control
resynchronization

Bruce Powel Douglass, Ph.D.

i-Logix
Example Concurrent FSM

![Diagram of a Concurrent Finite State Machine (FSM)]

- **System**
  - Application_Subsystem
    - Off
    - Error
    - Power_Subsystem
      - Battery
      - Mains
      - Mains On-Line
      - Mains Off-Line
  - Switch to ON
  - Switch to OFF
  - Error Detected
  - POST Complete
  - Operational
  - Startup

 Bruce Powel Douglass, Ph.D. 
 i-Logix
Communication in Concurrent FSMs

- **Broadcast events**
  - Events received by more than one concurrent FSM
  - Results in transitions of the same name in different FSMs

- **Propagated transitions**
  - Transitions which are generated as a result of transitions in other FSMs
Propagation and Broadcasts

Diagram:

- S
  - S_Component_1
    - A
    - B
      - T1
    - C
      - T2 ∧ T3(x,y,z)
  - S_Component_2
    - D
      - T3
    - E
      - T1
  - S_Component_3
    - F
    - G
      - T4
    - H
      - T3
      - T6
      - T5
      - T7
Inherited State Behavior

- Two approaches to inheritance for generalization of reactive classes
  - Reuse (i.e. inherit) statecharts of parent
  - Use custom statecharts for each subclass
- Reuse of statecharts allows
  - specialization of existing behaviors
  - addition of new states and transitions
  - makes automatic code generation possible
Inherited State Behavior

- Assumes Liskov Substitution Principle for generalization:
  A subclass must be freely substitutable for the superclass in any operation

- You CAN
  - Add new states
  - Elaborate substates in inherited states
  - Add new transitions and actions

- You CANNOT
  - Delete inherited transitions or states
Inherited State Models

- **Blower**
  - Off
  - On
  - Switch On / \( f() \)
  - Switch Off

- **Dual Speed Blower**
  - Off
  - On
  - Low
  - High
  - Switch On / \( g() \)
  - Switch Off / \( h() \) \( k() \)

- **Dual Speed Multiheat Blower**
  - Off
  - On
  - Low
  - High
  - Cool
  - Warm
  - Hot
  - Switch On
  - Switch Off / \( k() \)
FSM Example: VVI Pacemaker

- 2 key objects executing concurrently
  - Communications object
  - Pacing Engine object

- Each can be modeled as an FSM

- IT IS NOT APPROPRIATE NOT TO USE CONCURRENCY IN THIS APP
Pacemaker Object Model

- «actor» Programmer
  - Communication Coil Driver
  - Communication Gnome
  - Ventricular Pacing Engine
    - Mode
    - Pacing Rate
    - Pulse Amplitude
    - Pulse Width
  - «actor» Heart
Ventricular Pacing Engine States

- On
  - tm(PulseWidth)
  - Disable Pace Electronics
  - tm(RefractoryTime) / EnableSensor()
- Refractory
  - Pace Start Cmd
  - Pace Stop Cmd
- Pacing
  - tm(SenseTime) / Disconnect Sensor
  - Enable Pace Electronics
  - Set Parameters / Activate for next cycle
- Waiting_for_Sense
  - V Sense
- Off
Communication Gnome States

Diagram of state transitions and conditions:
- **Disabled**
  - Transition to Enabled
  - Transition to Receiving
  - Transition to Validating
  - Transition to Processing

- **Enable Comm**
  - Transition to Idle
  - Transition to Transmitting

- **Receiving**
  - Transition to Waiting_for_Msg_Timeout
  - Transition to Waiting_for_Msg_Req
  - Transition to Translating

- **Transmitting**
  - Transition to Waiting_for_Msg_Timeout
  - Transition to Waiting_for_Msg_Req
  - Transition to Translating

- **Idle**
  - Transition to Receiving
  - Transition to Transmitting

- **Translating**
  - Transition to Waiting_for_Msg_Timeout
  - Transition to Waiting_for_Msg_Req
  - Transition to Translating

- **Waiting_for_Msg_Timeout**
  - Transition to Enabled
  - Transition to Receiving

- **Waiting_for_Msg_Req**
  - Transition to Enabled
  - Transition to Receiving

- **Validating**
  - Transition to Processing

- **Processing**
  - Transition to Enabled
  - Transition to Receiving

- **Enable Comm**
  - Transition to Enabled
  - Transition to Receiving

- **Disable Comm**
  - Transition to Disabled

 transitions and conditions:
- **tm(MsgTime)**
- **tm(ByteTime)**
- **[isDone()]**
- **[isValid(msg)]**
- **[else]**
Communication Coil Driver States

Idle
- Transmit Byte Ready
  - [isDone( )]
- Pulse Transition / time = 0
  - tm(byteTime)
- tm(transBitTime)

Receiving_Bit
- Pulse Transition / time++
- tm(bitWait) / bit = decode(time)
  - shift in bit
  - if byte full, enqueue to Gnome

Waiting_for_Bit

Waiting_to_Transmit
- tm(transWait)

Transmitting_Bit
New Pacemaker Spec

- Both Atrial and Ventricular Pacing must be supported:
  - AAI, AAT, VVI, VVT, AVI
- Behavior for AAI is the same as VVI except it is a different object instance
- Behavior for AAT is the same as VVT except it is a different object instance
- Atrial behavior in AVI is different from ventricular behavior
Pacemaker Inherited States

- Communication Coil Driver
- Communication Gnome
- Programmer
- Pacing Engine
  - Mode
  - Pacing Rate
  - Pulse Amplitude
  - Pulse Width
- Ventricular Pacing Engine
- Atrial Pacing Engine
- Heart
Pacing Engine States

Diagram showing the states and transitions of a pacing engine:

- Off
  - ToOn
  - ToOff
- Idle
  - ToIdle
- On
  - Self Inhibited
    - tm(PulseWidth)
      - Self Triggered
        - tm(PulseWidth)
        - tm(SenseTime)
        - Sense
        - ToTriggered
- Refractory
  - tm(rf)
  - tm(SenseTime)
  - Sense
  - Waiting
  - Pacing
    - tm(SenseTime)
    - Sense
  - ToInhibited
- Dual
  - ToDual
Atrial AVI Mode State

Diagram:
- **Atrial Pacing Engine AVI Mode**
  - **Refractory**
    - Transition: V Refractory Done
  - **Pacing**
    - Transition: Atrial Pacing Done
  - **Waiting for Sense**
    - Transitions: V Sense, tm(V_Sense Time) ^ Atrial Pacing Start, Set Parameters / Activate for Next Cycle
Ventricular AVI Mode State

Ventricular Pacing Engine AVI Mode

- Refractory
  - Atrial Pacing Done
  - $tm(\text{refractoryTime})^+$
  - V_Refractory_Done / Enable Sensor

- Waiting for Sense
  - Atrial Pacing Start
  - VSense

- Waiting for A Pace
  - Set Parameters / Activate for Next Cycle
What is shown in Statecharts?

- Complete state space
- Static structural view
- Supports
  - Nesting
  - Concurrency
  - Propagated transitions
  - Broadcast Transitions
Other State Notations

- State Transition Tables
- State Specifications
- Augmented Message Sequence Diagrams
- Timing Diagrams
- Petri Nets
State Transition Tables

- Arranged as
  - Source x Target state
  - Source State x Transition

- Statecharts are very good at showing the \textit{structure} of the state space

- Tables are very good at identifying missing transitions

- Shlaer & Mellor say you should do \textit{both}
# State Table for VVI Engine

## transitions

<table>
<thead>
<tr>
<th>States</th>
<th>Stop</th>
<th>Start</th>
<th>Done</th>
<th>Timeout</th>
<th>V Sense</th>
<th>Set Param</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Off</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2 Refractory</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3 Pacing</td>
<td>1</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4 Waiting</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
What’s shown in State Tables?

- Complete state space
- Good for seeing missing/erroneous transitions
- No concurrency (one thread per table)
- Propagated transitions
- Broadcast transitions
- No actions
Object (Module) State Specifications

- Work in conjunction with statecharts and state tables
- Textual specifications
State Specifications

- State
  - Name
  - Description
  - Activities
  - Transitions accepted

- Transitions
  - Name
  - Guards
  - Event List
  - Actions List
State Specifications

- Easy to define requirements which are
  - Testable
  - Traceable (good for TUV, FDA, DoD)
- Can fully describe and define the states and transitions
- Recommendation: *Put all three in a single object behavioral document*
  - Statecharts
  - State tables
  - State specifications
Augmented Sequence Diagrams

- Dynamic
  - Do not show full state space
- Show specific thread through the state space
  - “Scenario”
- Can be augmented with State indicators
- Good for “walking through” behavior
- Do not replace static structural views
Sequence Diagrams

- Vertical lines represent objects
- Horizontal arrows represent messages (incl. transitions)
- Time flow from the top of the page downwards
- Sequence only is shown normally
Sequence Diagrams

Object 1

Object 2

Object 3

Object 4

msg 1

msg 3

msg 5

msg 6

msg 7

msg 8

msg 9

msg 4

msg 2

Time
Augmented Sequence Diagrams

```
Pacing_Engine
  Pacing_Rate_Time()
    Waiting
    Pacing_Rate_Time()
      tm(Pacing_Rate_Time)
      Waiting
      Pacing_Rate_Time()
        tm(Pacing_Rate_Time)
        Waiting
        Pacing_Rate_Time()

Pace_Timer
  Ventricular_Event()
    Waiting

Ventricular_Sensor
```

Bruce Power Douglass, Ph.D.
Adding Time Annotations

Pacing_Engine  Pacing_Timer  Ventricular_Sensor

Pacing_Rate_Time()

Ventricular_Event()

{ 800 ms +/- 10 ms }

Ventricular_Event()

Pulse_Width_Time()

{ b - a < 2 ms }
What’s shown in Augmented Sequence Diagrams?

- Dynamic scenarios
  - typically a single state chart will result in many ASDs
- Good place to add dynamic timing information
- Not all messages result in state transitions
Timing Diagrams

- Familiar
  - Used by electrical engineers
- Show state along vertical axis
- Show linear time along horizontal axis
- Depict particular scenarios
- For usage see
  - *Doing Hard Time: Using Object Oriented Programming and Software Patterns in Real Time Applications* (Addison-Wesley, Spring 1999)
Simple Timing Diagram

- **State**: Off, Refractory, Waiting for V Sense, Pacing
- **Time**: Pace Start Cmd, Ventricular Sense, Timeout, Done

Bruce Powel Douglass, Ph.D.
Complex Timing Diagram

- Initiation Time
- Trailing Jitter

- Rise Time
- Execution Time
- Dwell Time
- Fall Time

- Leading Jitter
- Slack Time

- Period

- Deadline
Example with jitter and rise times
Example with Dwell and Slack

<table>
<thead>
<tr>
<th>Time</th>
<th>Initiation</th>
<th>Execution</th>
<th>Dwell</th>
<th>Deadline</th>
</tr>
</thead>
</table>

Bruce Powel Douglass, Ph.D.
Concurrency in Timing Diagrams

- Concurrency can be shown by creating horizontal “bands” of states
  - Usually one band per object

- Shows the timing relationships between concurrent threads
Concurrency in Timing Diagrams

Waveform Parameter

Queue

Waveform Display

Acquiring
Storing

Idle
Inserting
Removing

Off
Acquiring
Scaling
Displaying

Bruce Powel Douglass, Ph.D.
Other Applications of Timing Diagrams

- Show timing relationships of functional call threads
- Show testable time budgets
- Assist in understanding RMA results
- Shows sequence of states and object reactions to events
What’s shown in Timing Diagrams?

- Good view of overall time
- Timing of interaction of concurrent states
- Timing details
  - Jitter
  - Execution time
  - Dwell time
  - Slack time
  - Rise and Fall time
Petri Nets

- Petri nets are a generic modeling tool
- FSMs are a special case of Petri nets
- Petri nets are defined as a set of
  - Places which hold tokens
  - Tokens small filled circles
  - Arcs directed lines
  - Transitions bars connecting arcs from places to places
- Petri nets can show concurrency by permitting multiple tokens
Petri Net Rules

- A Petri net is executed by moving tokens
- A transition can fire iff all of its input places contain tokens
- The firing of a transition
  - Removes a token from each input place
  - Puts a token in each output place
- The number of tokens a place can hold is called its **capacity**
Simple Petri Net

place

arc

transition

Bruce Powel Douglass, Ph.D.  
i-Logix  
Page 65
Standard Programming Constructs

Sequencing

Selection (or contention)

Explicit Control Branching

Explicit Control Synchronization

Looping

Concurrent threads
Pacemaker Petri Net

- Comm Off
- Idle
- Sending
- Receiving
- Checking
- RSO
- RSO
- RSO
- RSO
- RSO
- PSC
- PSC
- PSC
- PSC
- PSC

- Pacing Off
- Waiting for V Sense
- Refractory
- Pacing

RSO  Reed Switch Opens
PSC  Pace Stop Command
Example above shows a queuing model between two asynchronous threads: ECG Waveform acquisition and display.
What’s Shown in Petri Nets?

- Generalized behavior (incl. state behavior)
- Concurrency
- Can be augmented with time
- Many different extensions are available
- Petri nets suffer from
  - lack of scalability because they are flat like Mealy-Moore state models
  - lack of tools
FSMs and Development Process

- FSMs apply to *OBJECTS*
  - Sensor object
  - Queue object
  - Pacemaker pacing engine object
  - Language parser object
Structured Process

- Identify behavioral functions that exhibit state behavior
- For each such function, design a FSM
  - For each state, define
    - Valid transitions
    - Actions
    - Activities
- Decide on an implementation strategy
Object Oriented Process

- Identify classes and objects
- Identify which classes have FSMs
- Define a single FSM for each relevant class
  - For each state, define
    - Valid transitions
    - Actions
    - Activities
- Decide on an implementation strategy
Implementation Strategies

- Case/Switch statements
- FSM Generator
- Centralized state machine
- Separate state machines for each FSM object
Case/ Switch Statements

Switch (stateVar) {
    case state1: ... break;
    case state2: ... break;
    case state3: ... break;
    case default: // invalid state
};
Case/ Switch Statements

Switch (stateVar) {
    case state1: switch(transition) {
        case T1: ... break;
        case T2: ... break;
        case default: // invalid transition
    };
    break;
}
Centralized State machine

- Object 1
  - State Table
  - Transition
  - Action list
  - New State

- State Central
  - ProcessTransition(...)

Bruce Powel Douglass, Ph.D.  i-Logix
Separate State machines

class myClass: public FSM
{
    FSM myFSM;
};
## Separate State Machines

### Class FSM
- **Current State**
- **AcceptTransition(t)**
- **ApplyTActions(s)**
- **ApplyEntryActions(s)**
- **ApplyExitActions(s)**

### State
- **ID**
- **TransitionList**
- **EntryActionList**
- **ExitActionList**
- **ActivityList**

### Transition
- **ID**
- **Target State**
- **ActionList**

### Action
- **F()**

---

Bruce Powel Douglass, Ph.D.

i-Logix
Summary

- Objects have behavior
  - Simple
  - Continuous
  - State-driven
- Modeling objects as Finite State Machines simplifies the behavior
- States apply to objects
- FSM Objects spend all their time in exactly 1 state (which may contain concurrent substates)
Summary

- States are *disjoint ontological conditions that persist for a significant period of time*.

- States are defined by one of the following:
  - The values of all attributes of the object
  - The values of specific attributes of the object
  - Disjoint behaviors
    - Events accepted
    - Actions performed
Summary

- Transitions are the representation of responses to events within FSMs
- Transitions take an insignificant amount of time
- Actions are functions which may be associated with
  - Transitions
  - State Entry
  - State Exit
- Activities are processing that continues as long as a state is active
Summary

- Harel Statecharts provide
  - Nested States
  - Concurrency
  - Propagated and Broadcast Transitions
  - Orthogonal Components
  - Guards on transitions
  - Flexible action model
  - Activities within states
  - History
  - Inherited state behavior
Summary

- Statecharts show static structural view
- State tables show missing transitions
- State specifications are good for defining testable, traceable requirements
- Sequence diagrams show scenarios
- Timing diagrams show overall timing in scenarios
- Petri nets are more general and show static structural view