Finite-State Machines

- A finite state machine (FSM)
  - An abstract representation of behavior exhibited by some systems
  - Derived from application requirements.
    - But, not all aspects of requirements are specified by an FSM. (e.g., real time requirements, performance requirements)
  - A part of UML 2.x
- Example (Moore machine- actions are associated with states)

Mealy vs. Moore Machine

Source: “State machines and Statecharts” by Bruce Powel Douglass
# Embedded Systems and FSMs

- An embedded system often
  - Receives inputs from its environment and
  - Responds with appropriate actions.
  - While doing so, it moves from one state to another.
    - The response depends on its current state.

- Behavior of an embedded system is often modeled by a finite state machine (FSM).

---

## Finite-State Machines

- Example with initial state, final state, and action
  - Mealy machine (actions are associated with transitions)

- Is FSM requirements spec or design spec?
  - Can be useful for both

- Application domains?
  - GUIs, network protocols, pacemakers, Teller machines, WEB applications, safety software modeling in nuclear plants, and many more.

---

## FSM and Statecharts

- Note that FSMs are different from statecharts.
  - While FSMs can be modeled using statecharts, the reverse is not true.
  - Techniques for generating tests from FSMs are different from those for generating tests from statecharts.

- The term “state diagram”, a directed graph, is often used to denote a graphical representation of an FSM or a statechart.

---

## FSM (Mealy machine): Formal definition

**Deterministic**

*A finite-state machine is a six-tuple* $(X, Y, Q, q_0, \delta, O)$, *where*

- $X$ is a finite set of input symbols also known as the input alphabet.
- $Y$ is a finite set of output symbols also known as the output alphabet.
- $Q$ is a finite set of states.
- $q_0 \in Q$ is the initial state.
- $\delta : Q \times X \rightarrow Q$ is a next state or state transition function.
- $O : Q \times X \rightarrow Y$ is an output function.

**Non-deterministic**

$\delta : Q \times X \rightarrow 2^Q$
Formal Description Examples

<table>
<thead>
<tr>
<th>X</th>
<th>[Turn-CW]</th>
<th>[Turn-CW, Turn-CCW] [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, *]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Q</td>
<td>[OFF, ON_DIM, ON_BRIGHT]</td>
<td>[OFF, ON_DIM, ON_BRIGHT] [q0, q1, q2]</td>
</tr>
<tr>
<td>q0</td>
<td>[OFF]</td>
<td>q0</td>
</tr>
<tr>
<td>F</td>
<td>None</td>
<td>q1</td>
</tr>
<tr>
<td>δ</td>
<td>See Figure</td>
<td>See Figure</td>
</tr>
<tr>
<td>O</td>
<td>Not applicable</td>
<td>See Figure</td>
</tr>
</tbody>
</table>

Test Generation from FSMs

Blue: Generated data

Our focus

Test generation algorithm

Requirements ----- FSM ----- Test generation algorithm

Test generation for application

Application Test inputs

FSM based Test inputs

Test inputs

Test driver

Application

Pass/fail

Oracle

Observed behavior

©Aditya P Mathur 2009

Tabular Representation

<table>
<thead>
<tr>
<th>Current state</th>
<th>Action</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>q0</td>
<td>INIT (num, d)</td>
<td>q1</td>
</tr>
<tr>
<td>q1</td>
<td>ADD (num, d)</td>
<td>OUT (num)</td>
</tr>
<tr>
<td>q2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FSM Properties

- **Completely specified**
  - An FSM M is said to be completely specified if from each state in M there exists a transition for each input symbol.

- **Strongly connected**
  - An FSM M is considered strongly connected if for each pair of states (q_i, q_j) there exists an input sequence that takes M from state q_i to q_j.
FSM Properties

- **V-Equivalence**
  - Let $M_1 = (X, Y, Q_1, m^1_0, T_1, O_1)$ and $M_2 = (X, Y, Q_2, m^2_0, T_2, O_2)$ be two FSMs.
  - Let $V$ denote a set of non-empty strings over the input alphabet $X$, i.e., $V \subseteq X^*$.
  - Let $q_i$ and $q_j$ be two different states of machines $M_1$ and $M_2$, respectively.
  - $q_i$ and $q_j$ are considered \textit{V-equivalent} if $O_1(q_i, s) = O_2(q_j, s)$ for all $s \in V$.
  - $q_i$ and $q_j$ are considered \textit{equivalent} if $O_1(q_i, s) = O_2(q_j, s)$ for any set $V$.
  - Otherwise, they are \textit{distinguishable}.

- **k-equivalence**
  - Let $M_1 = (X, Y, Q_1, m^1_0, T_1, O_1)$ and $M_2 = (X, Y, Q_2, m^2_0, T_2, O_2)$ be two FSMs.
  - States $q_i \in Q_1$ and $q_j \in Q_2$ are considered \textit{k-equivalent} if, when excited by any input of length $k$, yield identical output sequences.
  - States that are not $k$-equivalent are considered \textit{k-distinguishable}.
  - It is also easy to see that if two states are $k$-distinguishable for any $k > 0$ then they are also distinguishable for any $n \geq k$.
  - If $M_1$ and $M_2$ are not $k$-distinguishable then they are said to be $k$-equivalent.
  - In other words, there is no $s \in X^k$ such that $O(q_i, s) \neq O(q_j, s)$

- **Machine equivalence**
  - Machines $M_1$ and $M_2$ are said to be equivalent if
  - (a) for each state $\sigma$ in $M_1$ there exists a state $\sigma'$ in $M_2$ such that $\sigma$ and $\sigma'$ are equivalent and,
  - (b) for each state $\sigma$ in $M_2$ there exists a state $\sigma'$ in $M_1$ such that $\sigma$ and $\sigma'$ are equivalent.

- **Minimal machine**
  - An FSM $M$ is considered minimal if the number of states in $M$ is less than or equal to any other FSM equivalent to $M$.

Fault Model

- An FSM serves to specify the correct requirement or design of an application. Hence tests generated from an FSM target faults related to the FSM itself.

- Fault model defines a small set of possible faults that can occur in the implementation.

\textit{What faults are targeted by the tests generated using an FSM?}
Fault Model (Sequencing Faults)

- Correct design
- Operation error
- Transfer error

Fault Model (Sequencing Faults)

- Correct design
- Extra state error
- Missing state error

Mutants of FSMs

- Mutants of the model $M_d$
  - Possible implementations of the model
  - Obtained by introducing $1^*$ faults $1^*$ times
    - Use the fault model introduced

Mutants of FSMs

- Some mutants may be equivalent to $M_d$
  - Output behaviors of $M_d$ and the mutants are identical on all possible inputs
  - Note that this is different to program mutation

- Example first-order mutants
  - Find a test that distinguish $M$ and $M_1$
  - Find a test that distinguish $M$ and $M_2$
Fault Coverage

- How to evaluate the goodness of a test set?
  - Count how many faults it reveals in an implementation $M_i$

- Formal definition of fault coverage, given fault model
  - $N_i$: total number of first-order mutants of the machine $M$ (i.e., the number of all possible implementations)
  - $N_e$: number of mutants that are equivalent to $M$
  - $N_d$: number of mutants distinguished by a test set $T$ generated using a test-generation method.
  - $N_f$: number of mutants not distinguished by $T$

$$FC(T, M) = \frac{\text{Number of mutants not distinguished by } T}{\text{Number of mutants that are not equivalent to } M} = \frac{N_f - N_e - N_d}{N_i - N_e}$$

Test Generation using Chow’s Method

- Assumptions
  - Minimal, connected, deterministic
  - Completely specified

- Algorithm Sketch (derive test set from a given FSM $M$)
  - Estimate the max. number of states ($m$) in the correct implementation of $M$.
  - Construct the characterization set $W$ for $M$.
  - Construct the testing tree for $M$ and generate the transition cover set $P$ from the testing tree.
  - Construct set $Z$ from $W$ and $m$.
  - Desired test set is $P-Z$

Step 1: Estimation of $m$

- We do not have access to the correct design or the correct implementation. So we need to estimate.
- This is based on a knowledge of the implementation. In the absence of any such knowledge, let $m=|Q|$.

Step 2: Construction of $W$. What is $W$?

- Let $M=(X, Y, Q, q_1, \delta, O)$ be a minimal and complete FSM.
- $W$ is a finite set of input sequences that distinguish the behavior of any pair of states in $M$.
- Given states $q_i$ and $q_j$ in $Q$, $W$ contains a string $s$ such that: $O(q_i, s) \neq O(q_j, s)$
- Example
  - $W=\{\text{baaa,aa,aaa}\}$
  - e.g., $\text{baaa}$ distinguishes state $q_1$ from $q_2$ as $O(\text{baaa},q_1) \neq O(\text{baaa},q_2)$
Steps to Construct W

- Step 1: Construct a sequence of \( k \)-equivalence partitions of \( Q \) denoted as \( P_1, P_2, \ldots, P_m \), \( m > 0 \).

- Step 2: Traverse the \( k \)-equivalence partitions in reverse order to obtain distinguishing sequence for each pair of states.

\[ k \]-equivalence Partition of \( Q \)?

- A \( k \)-equivalence partition of \( Q \), denoted as \( P_k \), is a collection of \( n \) finite sets \( \Sigma_{k1}, \Sigma_{k2}, \ldots, \Sigma_{kn} \) such that
  
  \[ \bigcup_{i=1}^{n} \Sigma_{ki} = Q \]
  
  - States in \( \Sigma_{ki} \) are \( k \)-equivalent.
  
  - If state \( v \) is in \( \Sigma_{ki} \) and \( v \) in \( \Sigma_{kj} \) for \( i \neq j \), then \( u \) and \( v \) are \( k \)-distinguishable.

\( k \)-equivalence

- Let \( M_1 = (X, Y, Q_1, m_1, T_1, O_1) \) and \( M_2 = (X, Y, Q_2, m_2, T_2, O_2) \) be two FSMs.

- States \( q_i \in Q_1 \) and \( q_j \in Q_2 \) are considered \( k \)-equivalent if, when excited by any input of length \( k \), yield identical output sequences.

How to Construct a \( k \)-equivalence Partition?

- Given an FSM \( M \), construct a 1-equivalence partition, start with a tabular representation of \( M \).

<table>
<thead>
<tr>
<th>Current state</th>
<th>Output</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_1 )</td>
<td>0</td>
<td>( q_1 )</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>0</td>
<td>( q_2 )</td>
</tr>
<tr>
<td>( q_3 )</td>
<td>0</td>
<td>( q_3 )</td>
</tr>
<tr>
<td>( q_4 )</td>
<td>1</td>
<td>( q_4 )</td>
</tr>
<tr>
<td>( q_5 )</td>
<td>1</td>
<td>( q_5 )</td>
</tr>
</tbody>
</table>

Construct 1-equivalence Partition

- Group states identical in their Output entries. This gives us 1-partition \( P_1 \) consisting of \( \Sigma_1 = \{q_1, q_2, q_3\} \) and \( \Sigma_2 = \{q_4, q_5\} \).

<table>
<thead>
<tr>
<th>( \Sigma )</th>
<th>Current state</th>
<th>Output</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( q_1 )</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>( q_2 )</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>( q_3 )</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>( q_4 )</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>( q_5 )</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**Construct 2-equivalence Partition**

- Rewrite $P_1$ table.
  - Remove the output columns.
  - Replace a state entry $q_i$ by $q_{ij}$ where $j$ is the group number in which lies state $q_i$.

<table>
<thead>
<tr>
<th>$\Sigma$</th>
<th>Current state</th>
<th>Next state</th>
<th>Group number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$q_1$</td>
<td>$q_{11}$</td>
<td>$q_{42}$</td>
</tr>
<tr>
<td></td>
<td>$q_2$</td>
<td>$q_{11}$</td>
<td>$q_{52}$</td>
</tr>
<tr>
<td></td>
<td>$q_3$</td>
<td>$q_{52}$</td>
<td>$q_{11}$</td>
</tr>
<tr>
<td>2</td>
<td>$q_4$</td>
<td>$q_{31}$</td>
<td>$q_{42}$</td>
</tr>
<tr>
<td></td>
<td>$q_5$</td>
<td>$q_{21}$</td>
<td>$q_{52}$</td>
</tr>
</tbody>
</table>

**Construct 2-equivalence Partition (cont.)**

- Construct $P_2$ table
  - Group all entries with identical **second subscripts** under the next state column.
  - Relabel the groups

<table>
<thead>
<tr>
<th>$\Sigma$</th>
<th>Current state</th>
<th>Next state</th>
<th>$P_2$ Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$q_1$</td>
<td>$q_{11}$</td>
<td>$q_{43}$</td>
</tr>
<tr>
<td></td>
<td>$q_2$</td>
<td>$q_{11}$</td>
<td>$q_{53}$</td>
</tr>
<tr>
<td>2</td>
<td>$q_3$</td>
<td>$q_{53}$</td>
<td>$q_{11}$</td>
</tr>
<tr>
<td>3</td>
<td>$q_4$</td>
<td>$q_{32}$</td>
<td>$q_{43}$</td>
</tr>
<tr>
<td></td>
<td>$q_5$</td>
<td>$q_{21}$</td>
<td>$q_{53}$</td>
</tr>
</tbody>
</table>

**Construct 3-equivalence Partition (cont.)**

- Construct $P_3$ table
  - Group all entries with identical **second subscripts** under the next state column.
  - Relabel the groups

<table>
<thead>
<tr>
<th>$\Sigma$</th>
<th>Current state</th>
<th>Next state</th>
<th>$P_3$ Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$q_1$</td>
<td>$q_{11}$</td>
<td>$q_{44}$</td>
</tr>
<tr>
<td></td>
<td>$q_2$</td>
<td>$q_{11}$</td>
<td>$q_{54}$</td>
</tr>
<tr>
<td>2</td>
<td>$q_3$</td>
<td>$q_{54}$</td>
<td>$q_{11}$</td>
</tr>
<tr>
<td>3</td>
<td>$q_4$</td>
<td>$q_{32}$</td>
<td>$q_{43}$</td>
</tr>
<tr>
<td>4</td>
<td>$q_5$</td>
<td>$q_{21}$</td>
<td>$q_{54}$</td>
</tr>
</tbody>
</table>

**Construct 4-equivalence Partition (cont.)**

- Construct $P_4$ table
  - Continue the regrouping and relabeling
  - Repeat constructing $k$-eq. partitions until converge

<table>
<thead>
<tr>
<th>$\Sigma$</th>
<th>Current state</th>
<th>Next state</th>
<th>$P_4$ Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$q_1$</td>
<td>$q_{11}$</td>
<td>$q_{44}$</td>
</tr>
<tr>
<td>2</td>
<td>$q_2$</td>
<td>$q_{11}$</td>
<td>$q_{55}$</td>
</tr>
<tr>
<td>3</td>
<td>$q_3$</td>
<td>$q_{55}$</td>
<td>$q_{11}$</td>
</tr>
<tr>
<td>4</td>
<td>$q_4$</td>
<td>$q_{33}$</td>
<td>$q_{44}$</td>
</tr>
<tr>
<td>5</td>
<td>$q_5$</td>
<td>$q_{22}$</td>
<td>$q_{55}$</td>
</tr>
<tr>
<td>k-Equivalence Partition: Convergence</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- The process is guaranteed to converge. ( P_n = P_{n+1} )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- When the process converges, and the machine is minimal, each state will be in a separate group.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- The next step is to obtain the distinguishing strings for each state.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>