Finite State Machine (Mealy)

Inputs = \{\text{cof-but, tea-but, coin}\}
Outputs = \{\text{cof, tea}\}
States: \{q_1, q_2, q_3\}
Initial state = q_1
Transitions = \{
    (q_1, \text{coin}, -, q_2),
    (q_2, \text{coin}, -, q_3),
    (q_3, \text{cof-but, cof}, q_1),
    (q_3, \text{tea-but, tea}, q_1)
\}

<table>
<thead>
<tr>
<th>condition current state</th>
<th>input</th>
<th>output</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>(q_1) coin</td>
<td>-</td>
<td></td>
<td>(q_2)</td>
</tr>
<tr>
<td>(q_2) coin</td>
<td>-</td>
<td></td>
<td>(q_3)</td>
</tr>
<tr>
<td>(q_3) cof-but</td>
<td>cof</td>
<td></td>
<td>(q_1)</td>
</tr>
<tr>
<td>(q_3) tea-but</td>
<td>tea</td>
<td></td>
<td>(q_1)</td>
</tr>
</tbody>
</table>

Sample run:

- Transition from \(q_1\) to \(q_2\): coin/
- Transition from \(q_2\) to \(q_3\): coin/ 
- Transition from \(q_3\) to \(q_1\): cof-but / cof

- Transition from \(q_2\) to \(q_3\): coin/
- Transition from \(q_3\) to \(q_1\): cof-but / cof
Fully Specified FSM

Self-loop completion

<table>
<thead>
<tr>
<th>condition</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>current state</td>
<td>input</td>
</tr>
<tr>
<td>q₁</td>
<td>coin</td>
</tr>
<tr>
<td>q₂</td>
<td>coin</td>
</tr>
<tr>
<td>q₃</td>
<td>cof-but</td>
</tr>
<tr>
<td>q₃</td>
<td>tea-but</td>
</tr>
<tr>
<td>q₁</td>
<td>cof-but</td>
</tr>
<tr>
<td>q₁</td>
<td>tea-but</td>
</tr>
<tr>
<td>q₂</td>
<td>cof-but</td>
</tr>
<tr>
<td>q₂</td>
<td>tea-but</td>
</tr>
<tr>
<td>q₃</td>
<td>coin</td>
</tr>
</tbody>
</table>
enum currentState {q1,q2,q3};
enum input {coin, cof_but, tea_but};
int nextStateTable[3][3] = {
    q2, q1, q1,
    q3, q2, q2,
    q3, q1, q1
};

int outputTable[3][3] = {
    0, 0, 0,
    0, 0, 0,
    coin, cof, tea
};

While(Input=waitForInput()) {
    OUTPUT(outputTable[currentState,input])
    currentState=nextStateTable[currentState,input];
}

enum currentState {q1,q2,q3};
enum input {coin,cof,tea_but,cof_but};

While(input=waitForInput){
    Switch(currentState){
        case q1: {
            switch (input) {
                case coin: currentState=q2; break;
                case cof_but:
                case tea_but: break;
                default: ERROR("Unexpected Input");
            }
            break;
        }
        case q3: {
            switch(input) {
                case cof_buf: {currentState=q3;
                    OUTPUT(cof);
                    break;}
                ...
                default: ERROR("unknown currentState")
            }
            break;
        }
    }
}
Concepts

- Two states $s$ and $t$ are (language) equivalent iff
  - $s$ and $t$ accepts same language
  - has same traces: $tr(s) = tr(t)$
- Two Machines $M_0$ and $M_1$ are equivalent iff initial states are equivalent
- A minimized / reduced $M$ is one that has no equivalent states
  - for no two states $s, t$, $s \neq t$, $s$ equivalent $t$
Conformance Testing

Given a specification FSM $M_S$

a (black box) implementation FSM $M_I$
determine whether $M_I$ conforms to $M_S$.
i.e., $M_I$ behaves in accordance with $M_S$
i.e., whether outputs of $M_I$ are the same as of $M_S$
i.e., whether the reduced $M_I$ is equivalent to $M_S$
Possible Errors

- output fault
- extra or missing states
- transition fault
  - to other state
  - to new state
Class Door{
    Private:
        // state variables
        // methods
    Public:
        Door();
        ~Door();
        Lock();
        Unlock();
        Move(Angle a) throws ErrorExc;
        // test Helpers?
        State getState();
        void setState(State);
        void reset();
}

Class PuckSupply{
    int _count=3;
    Public:
        int remaining() const;
        puck * get();
    }

Diagram of state transitions:
## Object State Tests

<table>
<thead>
<tr>
<th>current state</th>
<th>action</th>
<th>output</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open, angle=90</td>
<td>Move(45)</td>
<td>-</td>
<td>angle=45</td>
</tr>
<tr>
<td>OpenLocked, angle=45</td>
<td>Move(0)</td>
<td>ErrExc()</td>
<td>undefined, angle=45</td>
</tr>
</tbody>
</table>
Client/Server

Client Machine

Server Machine

RPC/RMI

Encapsulated State
FSM - Finite State Machine - or *Mealy Machine* is 5-tuple

\[ M = (S, I, O, \delta, \lambda) \]

- **S**: finite set of states
- **I**: finite set of inputs
- **O**: finite set of outputs
- \( \delta : S \times I \rightarrow S \): transfer function
- \( \lambda : S \times I \rightarrow O \): output function

Natural extension to sequences:

\[ \delta : S \times I^* \rightarrow S \]
\[ \lambda : S \times I^* \rightarrow O^* \]
Restrictions

FSM restrictions:

• *deterministic*
  \[ \delta : S \times I \rightarrow S \quad \text{and} \quad \lambda : S \times I \rightarrow O \quad \text{are functions} \]

• *completely specified*
  \[ \delta : S \times I \rightarrow S \quad \text{and} \quad \lambda : S \times I \rightarrow O \quad \text{are complete functions} \]
  (empty output is allowed; sometimes implicit completeness)

• *strongly connected*
  from any state any other state can be reached

• *reduced*
  there are no equivalent states
Desired Properties

- **Nice, but rare / problematic**
  - **status messages:** Assume that tester can ask implementation for its current state (reliably!!) without changing state
  - **reset:** reliably bring SUT to initial state
  - **set-state:** reliably bring SUT to any given state

```
SUT
FSM \text{M}_i
```

status?

reset?

set-state S10?

currentState=S10!
• Make test case for every transition in spec separately:

Test transition:
1. Go to state S1
2. Apply input a?
3. Check output x!
4. Verify state S2 (optionally)

Test purpose: “Test whether the system, when in state S1, produces output x! on input a? and goes to state S2”
Coffee Machine FSM Model

current state: 0

- Coffee? / -
- Token? / -
- Coffee? / Coffee!

Next states:
- Coffee? / - (to state 5)
- Coin? / - (to state 10)

- Token? / Token! (to state 10)
- Coin? / Coin! (to state 5)

current state: 5

- Coffee? / -
- Token? / Coin!

Next states:
- Coffee? / - (to state 0)
- Coin? / - (to state 10)

- Token? / Token! (to state 0)
- Coin? / Coin! (to state 10)

current state: 10

- Token? / Token!
- Coin? / Coin!

Next states:
- Token? / Token! (to state 5)
- Coin? / Coin! (to state 0)
Transition Testing –1

To test token? / coin!:
  go to state 5: set-state 5
  give input token? check output coin!
  verify state: send status? check status=10

|S| * | 1 | test cases remaining
• Make Transition Tour that covers every transition (in spec)

Test input sequence:

+ check expected outputs and target state by status message
Transition Testing -1

• Go to state S5:
• No Set-state property???
  • use reset property if available
  • go from S0 to S5
    ( always possible because of determinism and completeness )
• or:
  • synchronizing sequence brings machine to particular known state, say S0, from any state
  • ( but synchronizing sequence may not exist )
Transition Testing -1

synchronizing sequence: token? coffee?

To test token? / coin!: go to state 5 by: token? coffee? coin?
Transition Testing –2,3

• To test `token? / coin!`:
  1. go to state 5 by: `token? coffee? coin?`
  2. give input `token?`
  3. check output `coin!`
  4. verify that machine is in state 10
Transition Testing-4

- No Status Messages??
- State identification: What state am I in??
- State verification: Am I in state s?
  - Apply sequence of inputs in the current state of the FSM such that from the outputs we can
    - identify that state where we started; or
    - verify that we were in a particular start state
  - Different kinds of sequences
    - UIO sequences (Unique Input Output sequence, SIOS)
    - Distinguishing sequence (DS)
    - W-set (characterizing set of sequences)
    - UIOv
    - SUIO
    - MUIO
    - Overlapping UIO
State check:

- **UIO sequences** (verification)
  - sequence $x_s$ that distinguishes state $s$ from all other states:
    
    \[
    \forall t \neq s: \lambda(s, x_s) \neq \lambda(t, x_s)
    \]
  - each state has its own UIO sequence
  - UIO sequences may not exist

- **Distinguishing sequence** (identification)
  - sequence $x$ that produces different output for every state:
    
    \[
    \forall t, s \text{ with } t \neq s: \lambda(s, x) \neq \lambda(t, x)
    \]
  - a distinguishing sequence may not exist

- **$W$ - set of sequences** (identification)
  - set of sequences $W$ which can distinguish any pair of states:
    
    \[
    \forall t \neq s \text{ there is } x \in W: \lambda(s, x) \neq \lambda(t, x)
    \]
  - $W$ - set always exists for reduced FSM
Transition Testing-4: UIO

UIO sequences

state 0: coin? / - coffee? / -
state 5: token? / coin!
state 10: coffee? / coffee!
Transition Testing-4: DS

DS sequence:

- coffee? / -
- token? / -
- coffee? / coffee!
- token? / token!
- coin? / coin!
- token? / coin!
- coffee? / -

DS sequence: token? output state 0: -
output state 5: coin!
output state 10: token!
Transition Testing – 4 done

• To test token? / coin!:
  
  go to state 5: token? coffee? coin?
  give input token? check output coin!
  Apply UIO of state 10: coffee? / coffee!

Transition Testing - done

- 9 transitions / test cases for coffee machine
- if end-state of one corresponds with start-state of next then concatenate
- different ways to optimize and remove overlapping / redundant parts
- there are (academic) tools to support this
FSM Transition Testing

• Test transition :
  • Go to state S1
  • Apply input a?
  • Check output x!
  • Verify state S2

• Checks every output fault and transfer fault (to existing state)

• If we assume that
  
  the number of states of the implementation machine $M_I$
  is less than or equal to
  
  the number of states of the specification machine to $M_S$.

then testing all transitions in this way
leads to equivalence of reduced machines,
i.e., complete conformance

• If not: exponential growth in test length in number of extra states.
State Coverage

- Make *State Tour* that covers every state (in spec!)

Test sequence: coin? token? coffee?
Transition Coverage

- Make *Transition Tour* that covers every transition (in spec)

Test input sequence: