Automated Model Based Conformance Testing

Does the behavior of the (blackbox) implementation comply to that of the specification?
Mealy FSM

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}, \text{cof}, q_1),
    (q_3, \text{tea-but}, \text{tea}, q_1)
\}

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

Sample run:
q_1 \xrightarrow{\text{coin/-}} q_2 \xrightarrow{\text{coin/-}} q_3 \xrightarrow{\text{cof-but / cof}} q_1 \xrightarrow{\text{coin/-}} q_2 \xrightarrow{\text{coin/-}} q_3 \xrightarrow{\text{cof-but / cof}} q_1
Finite State Machine: FSM Model

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:

- \( \delta : S \times I^* \rightarrow S \)
- \( \lambda : S \times I^* \rightarrow O^* \)
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 M0 and M1 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 \( t \)
Fundamental Results

- Every FSM may be determinized accepting the same language (potential explosion in size).

- For each FSM there exist a language-equivalent minimal deterministic FSM.

- FSM’s are closed under $\cap$ and $\cup$

- FSM’s may be described as regular expressions (and vise versa)
Conformance Testing

Given a specification FSM $M_S$ and 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 $M_S$

Today:

• Deterministic Specifications
• SUT is an (unknown) deterministic FSM (testing hypothesis)
Restrictions

FSM restrictions:

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

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

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

- **reduced**
  there are no equivalent states
Possible Errors

- output fault (wrong or missing)
- extra or missing states
- transition fault
  - to other state
  - to new state
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
FSM  Transition Testing

- 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

States:
- 0
- 5
- 10

Transitions:
- Coffee? / - from 0 to 0
- Token? / coin! from 10 to 0
- Coin? / coin! from 10 to 10
- Coffee? / - from 5 to 5
- Token? / coin! from 0 to 5
- Coin? / coin! from 0 to 10
- Coffee? / - from 0 to 5
- Token? / - from 5 to 10
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| * | I | 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
Transition Testing-4

State check:

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

- **Distinguishing sequence (identification)**
  - sequence $x$ that produces different output for every state:
    - for all pairs $t, s$ 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:
    - for all pairs $t \neq s$ 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

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!

- 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* \( M_S \).

  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.
Class PuckSupply{
    int _count=10;
    Public:

        puck * get(){
            if(_count>0) {
                _count--;return get_puck();
            } else {
                return NULL;
            }
        }
};

•A Test case

s=new PuckSupply;
Assert(s.get()== aPuck);
Assert(s.get()== aPuck);
Assert(s.get()== aPuck);
Assert(s.get()== aPuck);
Assert(s.get()== aPuck);
Assert(s.get()== aPuck);
Assert(s.get()== aPuck);
Assert(s.get()== aPuck);
Assert(s.get()== aPuck);
Assert(s.get()== aPuck);
Assert(s.get()== aPuck);
Assert(s.get()== aPuck);
Assert(s.get()== aPuck);
Assert(s.get()== aPuck);
Assert(s.get()== aPuck);
Assert(s.get()== aPuck);
Assert(s.get()== aPuck);
Assert(s.get()== NULL);
Assert(s.get()== NULL);

s10 get() / puck
s9 get() / puck
s8 ... s1 get() / puck
s0 get() / NULL
Class PuckSupply{
    int _count=10;

    Public:
        //test helpers
        int getState() {return _count;}
        void setState(int count) {
            _count=count;
            alloc_puks();
        }
        puck * get(){
            if(_count>0) {
                _count--;return new puck();
            } else {
                return NULL;
            }
        }
};

Assert(s.get()==aPuck);
Assert(s.getState()==9);
s.setState(1);
Assert(s.get()==aPuck);
Assert(s.getState()==0);
Assert(s.get()==NULL);
Assert(s.getState()==0);
Assert(s.get()==NULL);
Assert(s.getState()==0);
Object Testing: Abstraction

Class PuckSupply{
    int _count=10;

    public:

    puck * get(){
        if(_count>0) {
            _count--; return new puck();
        } else {
            return NULL;
        }
    }
};

How many states in corresponding FSM?

getPuck() / puck

full

!Empty

get() / NULL

Emp

•⇒ Generate tests systematically from abstract descriptions to select reasonably number of tests
Object Tests

```java
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();
}

• D = new Door();
```
Object Tests

**Test Purpose:** A specific test objective (or observation) the tester wants to make on SUT

**TP1:** check that door can be open and locked?
- $E<> door.\text{OpenLocked}$
- Shortest Trace: $\text{Door()?.move(1)?.lock()}$?
Coverage Based Test Generation

- Multi purpose testing
- Cover measurement
- Examples:
  - Location coverage,
  - Edge coverage,
  - Definition/use pair coverage
Coverage Based Test Generation

- Multi purpose testing
- Cover measurement
- Examples:
  - Location coverage,
  - Edge coverage,
  - Definition/use pair coverage
Coverage Based Test Generation

- Multi purpose testing
- Cover measurement
- Examples:
  - Location coverage,
  - Edge coverage,
  - Definition/use pair coverage
Coverage Based Test Generation

- Multi purpose testing
- Cover measurement
- Examples:
  - Location Coverage,
  - Edge Coverage,
  - Definition/Use Pair Coverage
Location Coverage

- Test sequence traversing all locations
- Encoding:
  - Enumerate locations $l_0, \ldots, l_n$
  - Add an auxiliary variable $l_i$ for each location
  - Label each ingoing edge to location $i$ $l_i := \text{true}$
  - Mark initial visited $l_0 := \text{true}$
- Check: $\text{E}<\!( l_0 = \text{true} \land \ldots \land l_n = \text{true} )$
Edge Coverage

- Test sequence traversing all edges
- Encoding:
  - Enumerate edges \( e_0, \ldots, e_n \)
  - Add auxiliary variable \( e_i \) for each edge
  - Label each edge \( e_i := \text{true} \)
- Check: \( E<> ( e_0 = \text{true} \land \ldots \land e_n = \text{true} ) \)
Definition/Use Pair Coverage

- Dataflow coverage technique
- Def/use pair of variable $x$:

  ![Diagram](image)

  - $x := 0$
  - definition
  - no defs
  - $x \geq 4$
  - use

- Encoding:
  - $v_d \in \{\text{false}\} \cup \{e_0, \ldots, e_n\}$, initially false
  - Boolean array $du$ of size $|E| \times |E|
  - At definition on edge $i$: $v_d := e_i$
  - At use on edge $j$: if($v_d$) then $du[v_d, e_j] := true$
Definition/Use Pair Coverage

- Dataflow coverage technique
- Def/use pair of variable $x$:

  
  \[ x := 0 \]
  \[ x \geq 4 \]

  definition  no defs  use

- Encoding:
  - $v_d \in \{\text{false}\} \cup \{e_0, ..., e_n\}$, initially false
  - Boolean array $du$ of size $|E| \times |E|
  - At definition on edge $i$: $v_d := e_i$
  - At use on edge $j$: if($v_d$) then $du[v_d, e_j] := \text{true}$

- Check:
  - $E<> (\text{all } du[i,j] = \text{true} )$
Test Suite Generation

- In general a set of test cases is needed to cover a test criteria
- Add global reset of SUT and environment model and associate a cost (of system reset)

\[ \sigma = \varepsilon_0, i_0, \ldots, \varepsilon_1, i_1, \underbrace{\text{reset } \varepsilon_2, i_2, \ldots, \varepsilon_0, i_0}_{\sigma_i}, \varepsilon_1, i_1, \varepsilon_2, i_2, \ldots} \]

- Same encodings and min-cost reachability
- Test suite \( T = \{ \sigma_1, \ldots, \sigma_n \} \) with minimum cost
Optimal Tests

- **Shortest** test for max light??
- **Fastest** test for max light??
- **Fastest** edge-covering test suite??
- Least *power* consuming test??