Introduction to FSM-modelling

Brian Nielsen bnielsen@cs.aau.dk
Department of Computer Science,
Aalborg University, Denmark
Basic Definitions
System Structure

• How do we model components?
• How does components interact?
• How do we specify environment assumptions?
• How do we ensure correct behaviour
Behavior

Unified Model = **State Machine!**
Finite State Machine (Mealy)

Inputs = {cof-but, tea-but, coin}
Outputs = {cof, tea}
States: {q1, q2, q3}
Initial state = q1
Transitions = 
{(q1, coin, -, q2),
(q2, coin, -, q3),
(q3, cof-but, cof, q1),
(q3, tea-but, tea, q1)}

<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>
</tbody>
</table>

Sample run:

q1 → coin/ - → q2 → coin/ - → q3 → cof-but / cof → q1 → coin/ -
q2 → coin/ - → q3 → cof-but / cof → q1
**Finite State Machine (Moore)**

**Input sequence:** coin.coin.cof-but.coin.coin.cof-but

**Output sequence:** need2.need1.select.cof. need2.need1.select.cof

*need2=display shows “insert two coins”*
IO-FSM

Inputs = {cof-but, tea-but, coin}
Outputs = {cof, tea}
States: {q1, q2, q3}
Initial state = q1
Transitions = {
(q1, coin, q2),
(q2, coin, q3),
(q3, cof-but, q5),
(q3, tea-but, q4),
(q4, tea, q1),
(q5, cof, q1)
}

Sample run:
q1 → q2 → q3 → q5 → cof

action trace: coin?.coin?.coin!-coin?.coin?.coin?.coin!.
input sequence: coin.coin.coin.coin
Output sequence: cof.cof
FSM as program 1

```c
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");
        }
    ...
}
Spontaneous Transitions

<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>q₃</td>
<td>-</td>
</tr>
<tr>
<td>q₄</td>
<td>fix</td>
</tr>
</tbody>
</table>

alias internal transitions alias unobservable transitions
Non-deterministic FSM

<table>
<thead>
<tr>
<th>current state</th>
<th>input</th>
<th>output</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>q₁</td>
<td>coin</td>
<td>-</td>
<td>q₂</td>
</tr>
<tr>
<td>q₁</td>
<td>coin</td>
<td>-</td>
<td>q₁</td>
</tr>
<tr>
<td>q₂</td>
<td>coin</td>
<td>-</td>
<td>q₃</td>
</tr>
<tr>
<td>q₃</td>
<td>tea-but</td>
<td>tea</td>
<td>q₁</td>
</tr>
<tr>
<td>q₃</td>
<td>cof-but</td>
<td>cof</td>
<td>q₁</td>
</tr>
<tr>
<td>q₃</td>
<td>cof-but</td>
<td>mocca</td>
<td>q₁</td>
</tr>
</tbody>
</table>
Extended FSM

- EFSM = FSMs + variables + enabling conditions + assignments
- Easier way of expressing an FSM
- Can be translated into FSM if variables have bounded domain
- State: control location + variable states: \((q, total, capacity)\)

\[
\begin{align*}
(q_1, 0, 10) & \xrightarrow{\text{coin / -}} (q_1, 1, 10) & \xrightarrow{\text{coin / -}} (q_1, 2, 10) & \xrightarrow{\text{cof-but / cof}} (q_1, 0, 9)
\end{align*}
\]
Parallel Composition (independent)

\[ P \parallel Q \]

\[ (p_1, q_1) \]

\[ (p_2, q_1) \]

\[ (p_3, q_1) \]

\[ (p_2, q_2) \]

\[ (p_3, q_2) \]

\[ (p_1, q_2) \]

\[ (p_3, q_2) \]

\[ (p_1, q_3) \]

\[ (p_2, q_3) \]

\[ (p_3, q_3) \]
State Explosion Problem

- n parallel FSMs
- Each with $k$ states
- In Parallel they have $k^n$ states
- EXPONENTIAL!
  - $10^2 = 100$
  - $10^3 = 1000$
  - $10^4 = 10000$
  - $10^{10} = 10000000000$
Bank-box code

To open a bank box
the code must contain at least 2

To open a bank box
the code must end with

To open a bank box
the code must end with a palindrome.

To open a bank box
the code must end with

or with

To open a bank box
the code must end with a palindrome

e.g.:

!!
Determinization + minimization
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 $t$
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)
High-level FSM languages
UML-Statecharts
Textual Notations for FSM

Promela/Spin

```
int x;
proctype P(){
  do
  :: x<200 --> x=x+1
  od}

proctype Q(){
  do
  :: x>0 --> x=x-1
  od}

proctype R(){
  do
  :: x==200 --> x=0
  od}

init
{run P(); run Q(); run R()}
```

FSP/LTSA

```
SERVERv2 = (accept.request
->service>accept.reply->SERVERv2).
CLIENTv2 = (call.request
->call.reply->continue->CLIENTv2).
||CLIENT_SERVERv2 = (CLIENTv2 || SERVERv2)
/{call/accept}.
```
UppAal Navigator
UppAal

Modelling and verifying
untimed systems
Tool Support (model checking)

System Description

Requirement

Tools: UPPAAL, visualSTATE, ESTEREL, SPIN, Statemate, FormalCheck, VeriSoft, Java Pathfinder, Telelogic…
Home-Banking?

int accountA, accountB; //Shared global variables
//Two concurrent bank costumers

Thread costumer1 () {
    int a,b; //local tmp copy
    a=accountA;
b=accountB;
a=a-10; b=b+10;
accountA=a;
accountB=b;
}

Thread costumer2 () {
    int a,b;
a=accountA;
b=accountB;
a=a-20; b=b+20;
accountA=a;
accountB=b;
}

• Are the accounts in balance after the transactions?
A[] (pc1.finished and pc2.finished) imply (accountA+accountB==200)?
int accountA, accountB; //Shared global variables
Semaphore A,B;            //Protected by sem A,B
//Two concurrent bank costumers

Thread costumer1 () {
        int a,b; //local tmp copy
        wait(A);
        wait(B);
        a=accountA;
        b=accountB;
        a=a-10; b=b+10;
        accountA=a;
        accountB=b;
        signal(A);
        signal(B);
    }

Thread costumer2 () {
        int a,b;
        wait(B);
        wait(A);
        a=accountA;
        b=accountB;
        a=a-20; b=b+20;
        accountA=a;
        accountB=b;
        signal(B);
        signal(A);
    }
Semaphore FSM Model

Binary Semaphore

Counting Semaphore
Composition

IO Automata (2-way synchronization)
Composition

IO Automater
Semaphore Solution?

1. A[] (mc1.finished and mc2.finished) imply (accountA+accountB==200)
2. E<> mc1.critical_section and mc2.critical_section
3. A[] not (mc1.finished and mc2.finished) imply not deadlock

1. Consistency? (Balance)
2. Race conditions?
3. Deadlock?

1. A[] (mc1.finished and mc2.finished) imply (accountA+accountB==200) ✓
2. E<> mc1.critical_section and mc2.critical_section ✓
3. A[] not (mc1.finished and mc2.finished) imply not deadlock ✗
Global shared variables:
int accountA, accountB;

Local variables:
int a, b;

Local control:
• system state = snapshot of each machines control location
  + local variables + global variables
mc1.control=requestB, mc1.a=0, mc1.b=0,
mc2.control=requestB, mc2.a=0, mc2.b=0,
bsem1.control=closed, bsem2.control=open,
accountA=100, accountB=100
Process Interaction

- ! = Output, ? = Input
- Handshake communication
- Two-way

University = Coffee Machine || Lecturer
  - LTS?
  - How many states?
  - Traces ?

Coffee Machine

4 states

Lecturer

4 states

synchronization results in internal actions

4 states: Interaction constrain overall behavior
Broad-casts

- chan coin, cof, cofBut;
- broadcast chan join;
  - sending: output join!
  - every automaton that listens to join moves
  - i.e. every automaton with enabled “join?” transition moves in one step
  - may be zero!
Committed Locations

- Locations marked C
  - *No delay* in committed location.
  - Next transition must involve automata in committed location.

- Handy to model atomic sequences
- The use of committed locations reduces the number of states in a model, and allows for more space and time efficient analysis.

- S0 to s5 executed atomically
The Cruise Controller

User

Controller

CruiseControl

SpeedControl

Engine

engineOff, engineOn, acc, brake on, off, resume

enableControl, disableControl, recordSpeed

setThrottle

speed

enableControl, disableControl, recordSpeed

setThrottle

speed
END