State machines

L. Palopoli

email:palopoli@dit.unitn.it

URL:
http://www.dit.unitn.it/~palopoli

Tel. +39 0461 883967

General concepts

The concept of SM

- SM are systems operating on sequences:
  - *F: InputSignals -> OuputSignals*
    where both input signals and output signals have the form:
    - EventStream: Naturals_0 -> Symbols

{0,1,2,...}

  - This definition captures the concept of *ordering* between events
  - Reference to time (discrete or continuous) is neither explicit nor implied
- Crucial is the definition of a *state* that summarises the past story of the system

Example

x={x(0), x(1), …}
$x \in InputSignals$ → **Code Recognizer** → *y={y(0), y(1), …}*
$y \in OutputSignals$

$x \in InputSignals = [Nats_0 \to \{0,1\}]$
$y \in OutputSignals = [Nats_0 \to \{True, False\}]$

$$Recognizer(x)(n) = \begin{cases} True \text{ if } x(n)=0 \wedge x(n-1)=1 \wedge x(n-2)=0 \\ False \text{ otherwise} \end{cases}$$

Example:

$x=\{0,0,0,1,0,0 \ldots\} \to y=\{-, -, False, False, True, False\}$

A formal definition

A state machine has several components:

States Set of possible states (called state space)

Inputs Set of possible input elements (called input alphabet)

Outputs Set of possible output elements (called output alphabet)

update: States × Inputs –> States × Outputs the update function

The update function defines the new state and output given the current state and input.

initialState The initial state

Thus, a state machine can be described as a 5-tuple:

(*States, Inputs, Outputs, update, initialState*)

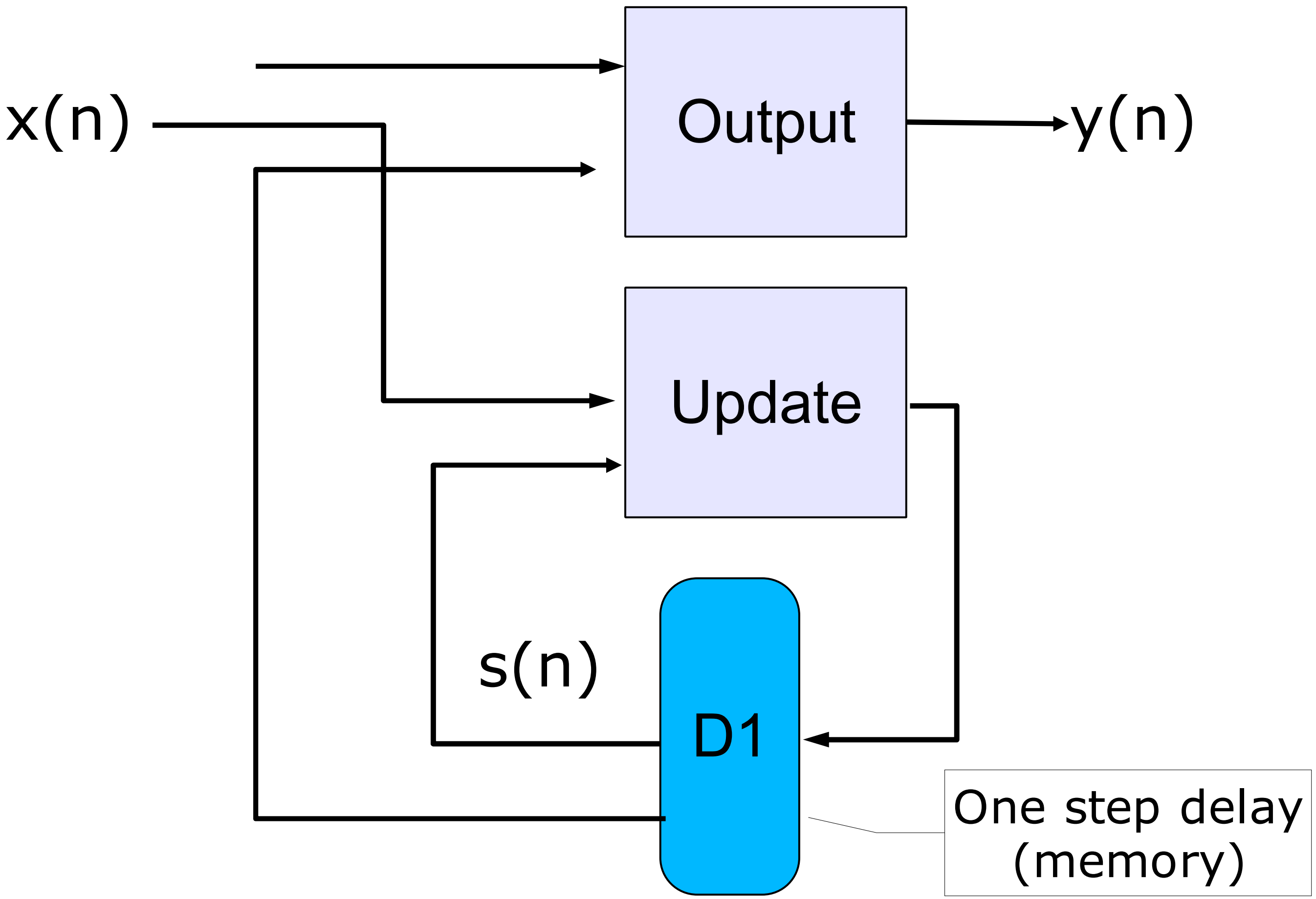
Update

- SM are causal: output depends only on the current state (that summarises the past history of the system) and on the current inputs
- The update dunction is often split into two functions:

update = (nextState, output),
nextState: States x Inputs → States
output: States x Inputs → Outputs

so that
$(s(n+1), y(n)) = \text{update}(s(n), x(n))$
$s(n+1) = \text{nextState}(s(n), x(n))$
$y(n) = \text{output}(s(n), x(n))$

Conceptual scheme



Stuttering

We define a special “do nothing” input called *absent*.

When x(n) = *absent*,

- the state does not change, so s(n+1) = s(n);
- the output is also “nothing”, i. e. y(n) = *absent*.
- We can always add absent symbols to input sequences without changing non absent outputs

This symbol, *absent*, is called the stuttering symbol.

Note that this symbol is always a possible input and output, so

$$absent \in Inputs,\ absent \in Outputs$$

FSM

- We will restrict to the case of machines having a finite state: Finite State Machines
- We will also restrict to the case of finite input and output alphabet

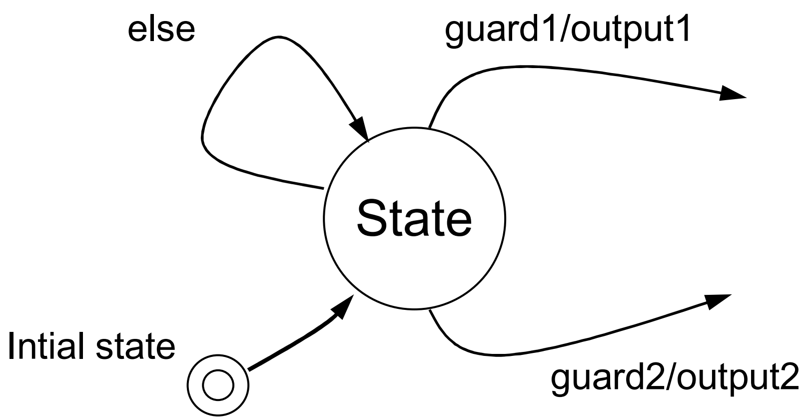
Update tables

- Update functions can be specificied using tables
- Parity example:

	$(s(n+1), y(n)) = update(s(n), x(n))$	
	$x(n) = True$	$x(n) = False$
$s(n) = even$	(odd, *True*)	(even, *False*)
$s(n) = odd$	(even, *False*)	(odd, *True*)

State transition diagrams

- State transition diagrams are a popular way for representing state machines
  - Bubbles represent states
  - Directed Arcs represent transitions
    - Transition are enabled by a guard
      - Guards are collections of input symbols
    - Transitions can also be associated to outputs



Excercise

Draw the transition diagram of a state machine with

Inputs = Bools

Outputs = Bools

At all times t, the output is true iff the inputs at times t-2, t-1, and t are all true .

Exercise - (continued)

State after i-th input :

0, if i-th input is false (or i = 0)

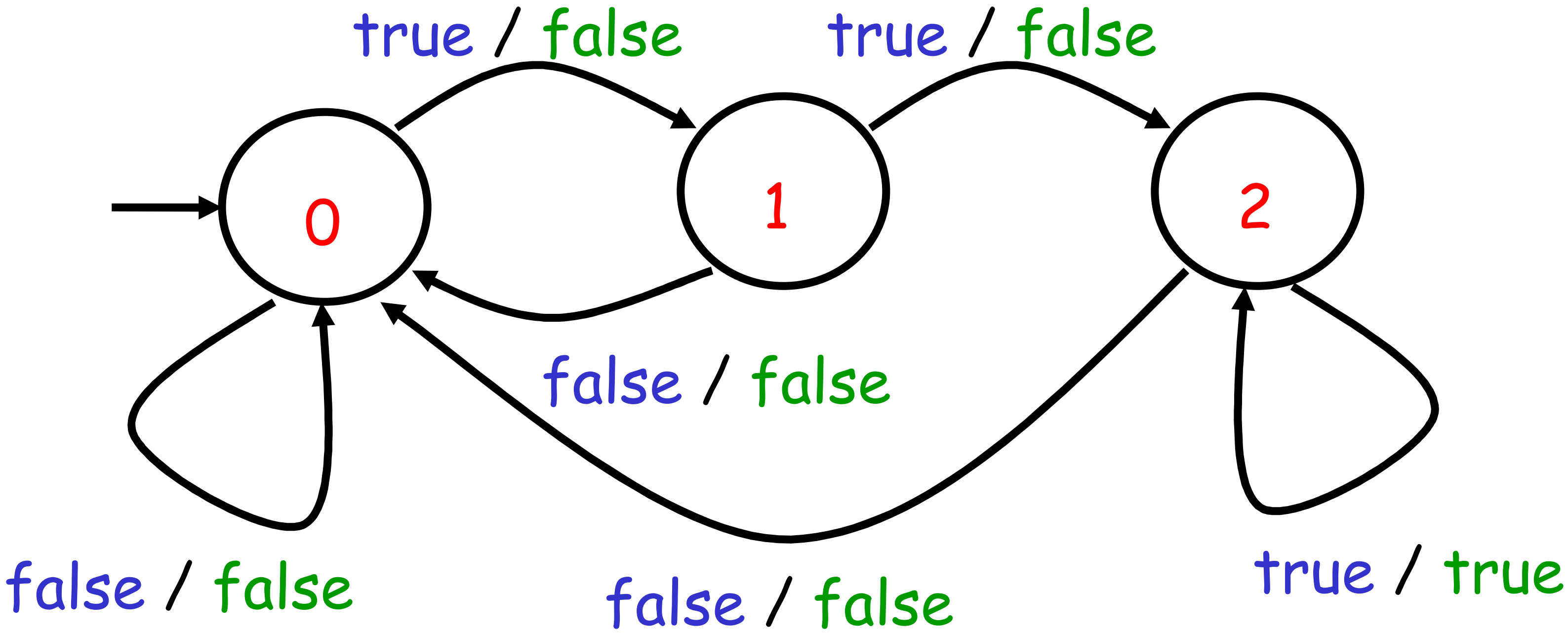
1, if i-th input is true and (i-1)-th input is false (or i = 1)

2, if both i-th and (i-1)-th inputs are true

Three states

Exercise - (Continued)

Transition Diagram



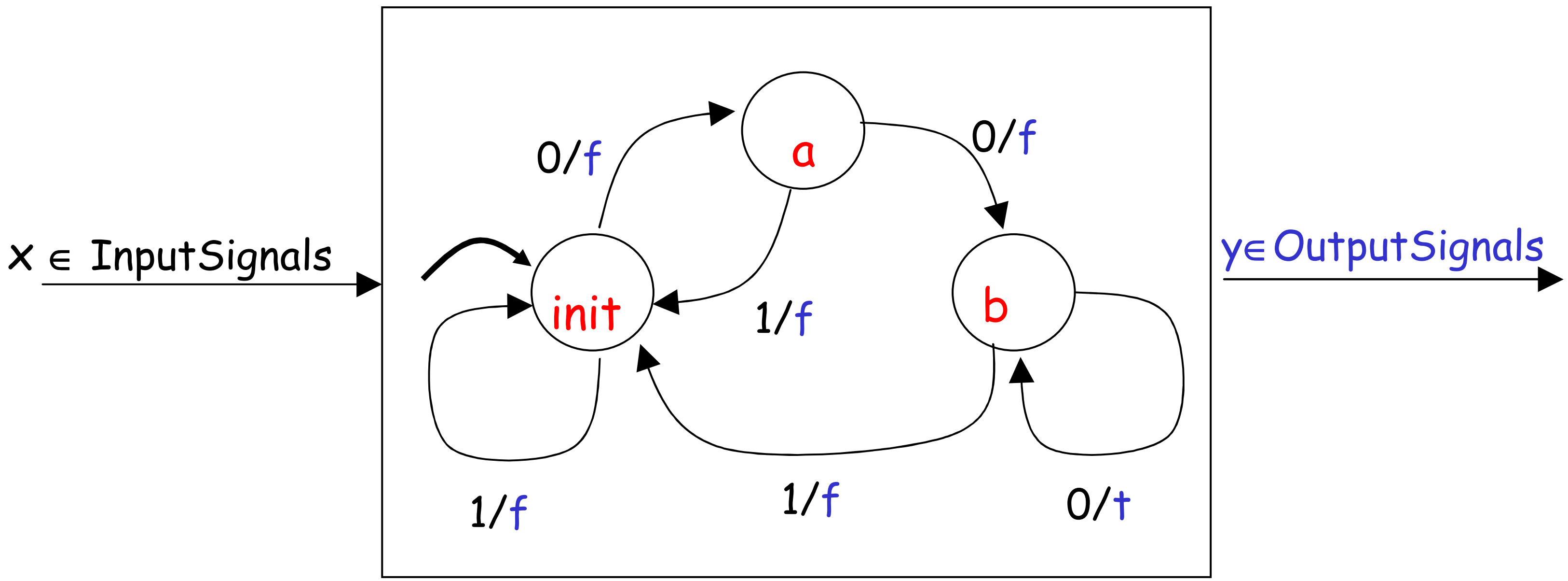
States = { 0, 1, 2 }

Inputs = Bools

Outputs = Bools

State transition diagrams

- An example



x = (0, 1, 0, 0, 0, 0, 1, 1, ...)
s = (init, a, init, a, b, b, b, init, ...)
y = (f, f, f, f, t, t, f, f, ...)

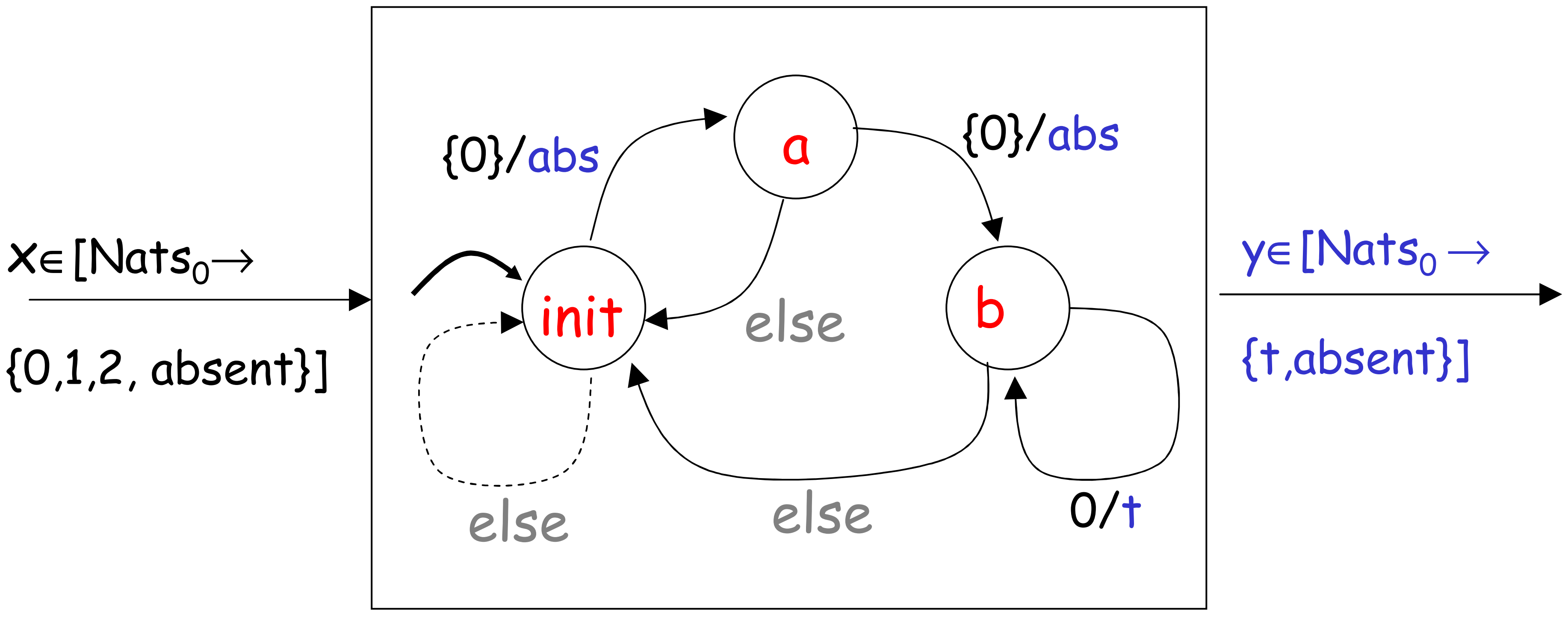
s(0) = init
(s(n+1), y(n)) = update (s(n), x(n)), n = 0,1,...

Shorthand

- If no guard is specified the transition is always taken (except for the stuttering symbol)
- If no output symbol is specified then the output is the stuttering symbol
- If no else transition is specified then we assume a self loop
- If inputs are {a,b,c,d,...,z} we can use {not a} in a guard instead of {b,c,...,z}

Shorthand

- The else loop



Some terminology

- **Receptiveness:** our SM always react to a symbol (there is an outgoing arc specified, or an else arc either specified or implied)
- **Determinism:** a SM is said deterministic if every input symbols is contained in one and only one outgoing arc
- State transitions triggered by a sequence of inputs are called **State response**
- A finite machine without output is called *finite automata*

Finite automata

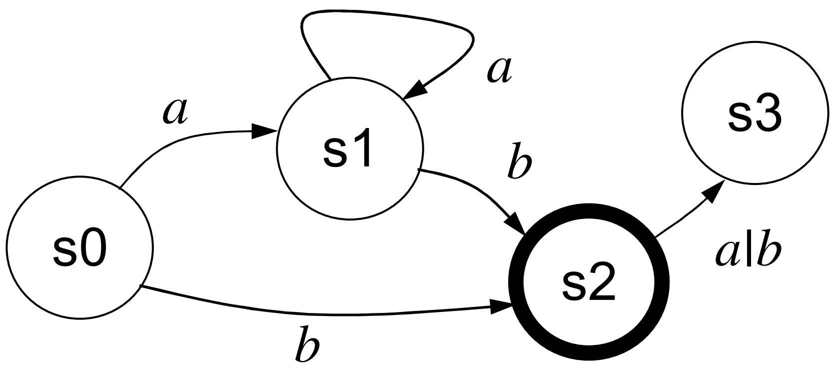
- Finite automata can be used to recognize patterns expressed by regular expressions
- In this case it is useful to introduce other markers (finite state) where the system arrives when it recognises a correct expression
- In this case we may consider unreceptive machine (i.e., the machine may not accept all events in all state)
  - as a consequence we may have deadlock

Finite automata - Example

- Consider the language composed of the expression:

$$b\,,ab\,,aab\,,aaab\,,aaaab\,,\ldots$$

- It is recognised by the following automaton



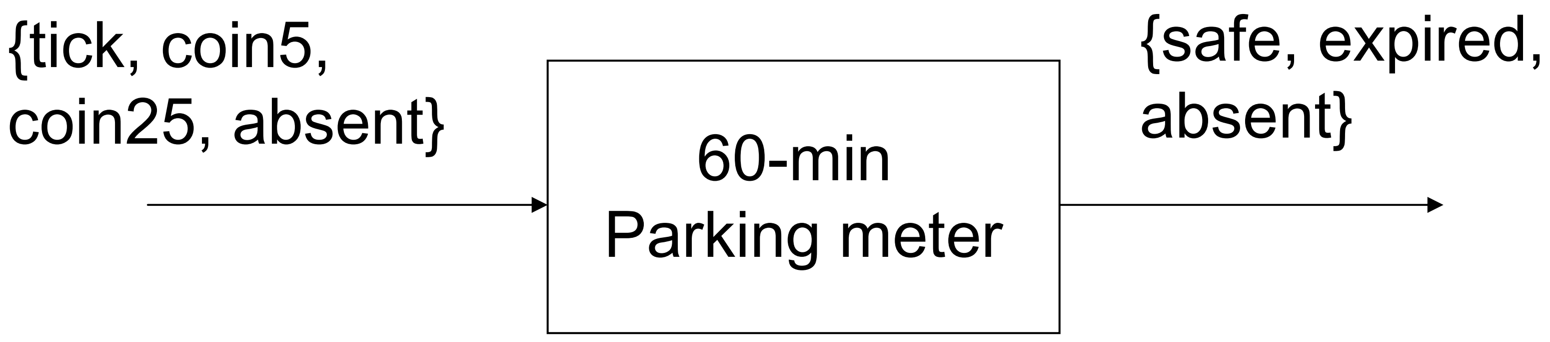
Finite automata - Example

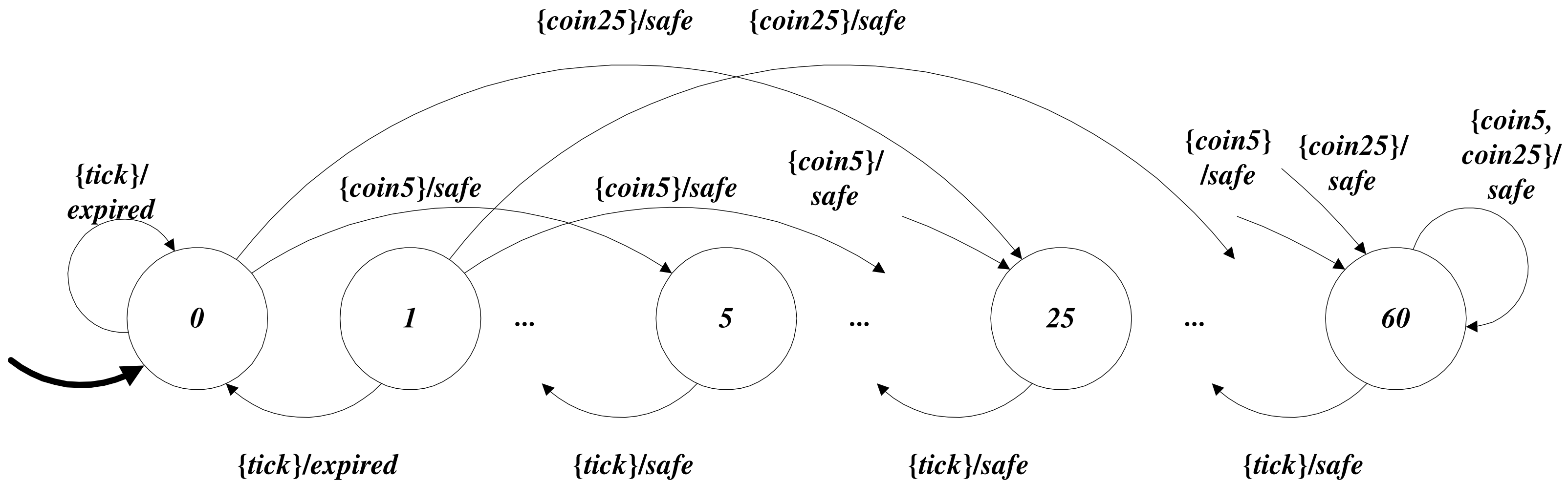
- Consider the language:

$$ab\,,\,aabb\,,\,aaabbb\,,\,aaaabbbb\,,\,,\,...$$

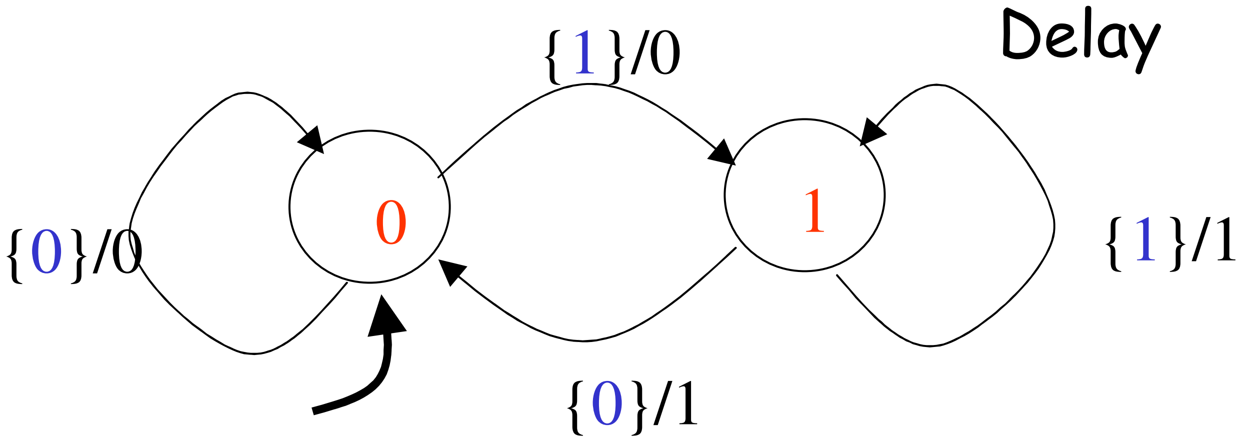
- It cannot be regognised by any finite automata. Why?

Finite automata – Parking meter example (from Lee-Varaija book)





Finite automata – delay example (from Lee-Varaija book)



InputSignals = OutputSignals = [$\text{Nats}_0 \rightarrow$ {0,1, absent}]

x = 0 0 1 1 0 …

s = 0 0 0 1 1 0 …

y = 0 0 0 1 1 0 …

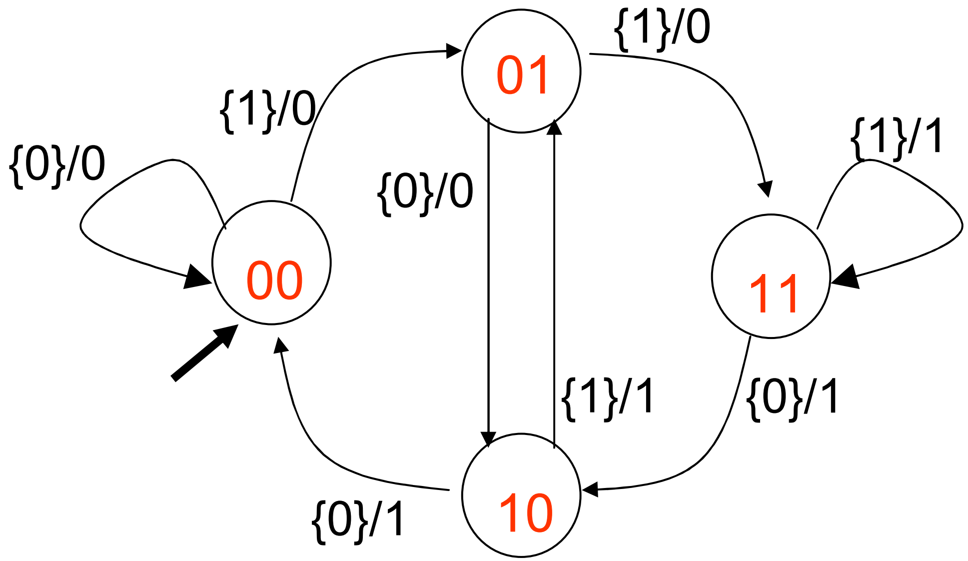
$\forall x \in$ InputSignals, $\forall n \in \text{Nats}_0$, Delay(x)(n)=0, n=0;
= x(n-1), n > 0

Finite automata – 2-delay example (from Lee-Varaija book)

$\forall x \in$ InputSignals, $\forall n \in Nats_0$,

$$\begin{aligned} Delay_2(x)(n) &= 0, & n = 0,1; \\ &= x(n-2), & n = 2,3,\ldots \end{aligned}$$

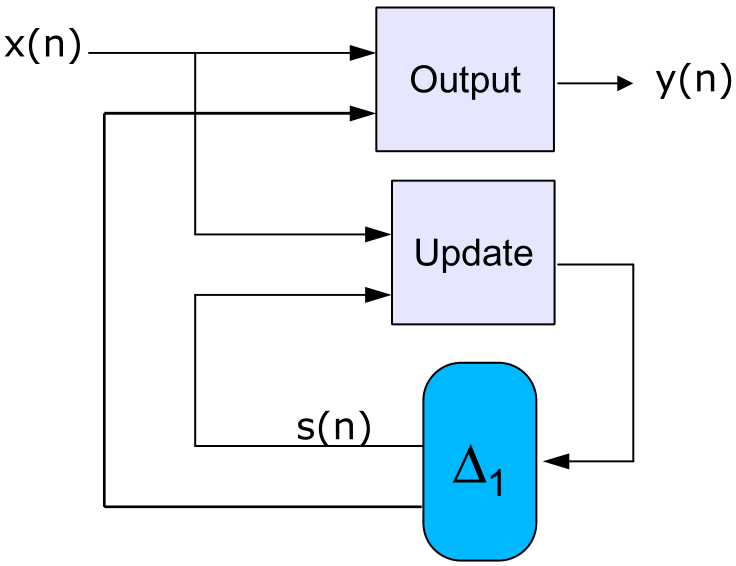
Implement $Delay_2$ as state machine



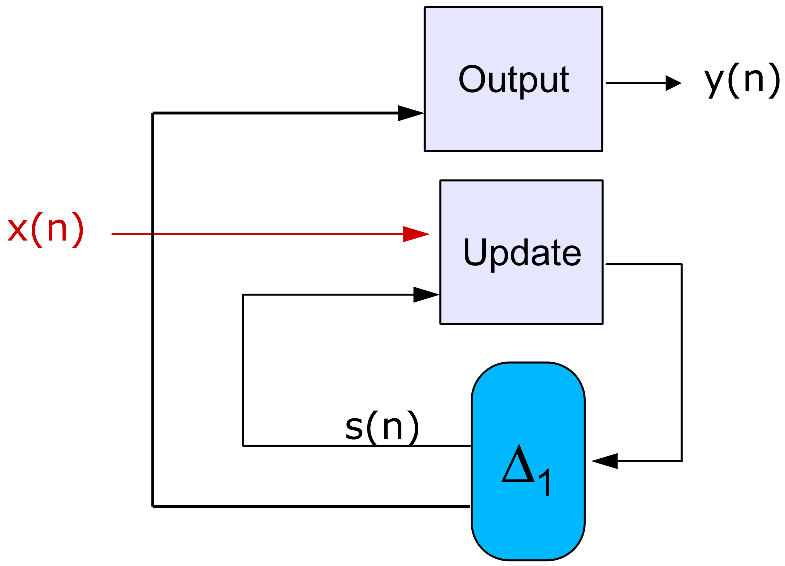
We will see later that $Delay_2 \sim Delay_1 \circ Delay_1$

Some definitions - (continued)

- The state machines addressed here are called Mealy machines. Mealy machines generate outputs during state transitions.
- Moore machines generate output while the system is in a particular state (output depends on state only).



Mealy



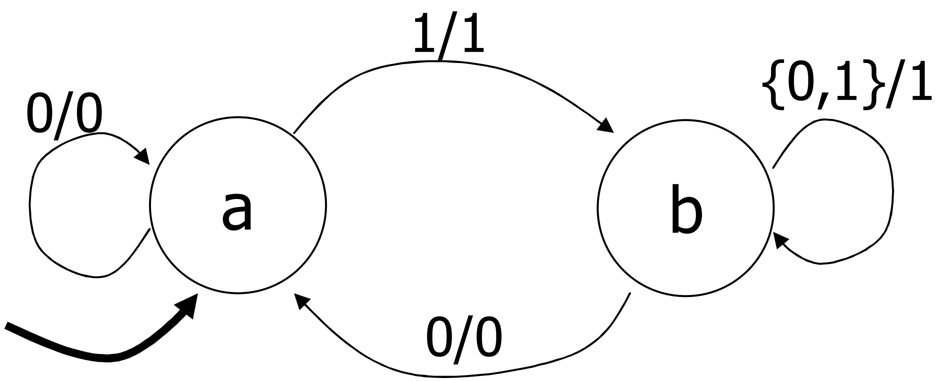
Moore

Some facts

- **A SM is state-determined:** Each transition and output depends only on the current state and current input.
  - Previous input elements only affect the transitions and output insofar as they determine the current state.
- A representation of a SM may not be unique
- Representations of a SM
  - Update tables (useful in computer implementations)
  - State transition diagrams (human friendly)
  - Sets and Functions (useful for mathematical purposes)

Nondeterministic State Machines

Nondeterministic state machines are like the deterministic state machines, except there may be more than one possible transition for a given current state and input.



When in state s(n)=b, if x(n) = 0, the next state s(n+1) can be either a or b.

The output y(n) can be either 0 or 1.

In a nondeterministic state machine, there is more than one possible state response and output sequence.

Nodeterministic State Machines: Parts

Nondeterministic state machines are described with the 5-tuple:

(*States, Inputs, Outputs,* ***possibleUpdates,*** *initialState*)

The update function is now called ***possibleUpdates***.

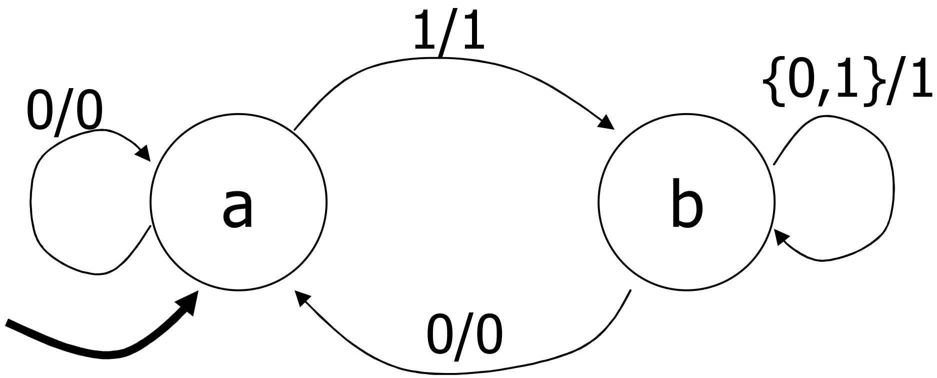
For a given input x(n) and current state s(n), *possibleUpdates* provides the **set** of possible next states s(n+1) and outputs y(n).

possibleUpdates: *States* × *Inputs* –> P(*States* × *Outputs*)

Here, the notation P() denotes the power set. For some set S, P(S) is the set of all subsets of S.

Nondeterministic State Machines: Example

Provide the 5-tuple definition for the following state machine:



States = {a, b}

Inputs = {0, 1, *absent*}

Outputs = {0, 1, *absent*}

initialState = a

	(s(n+1), y(n)) = *possibleUpdates*(s(n), x(n))	
	x(n) = 0	x(n) = 1
s(n) = a	(a, 0)	(b, 1)
s(n) = b	{(b, 1), (a, 0)}	(b, 1)

Why nondeterminism?

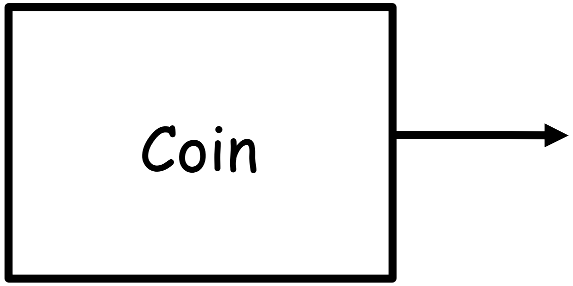
-modeling randomness

-modeling abstraction

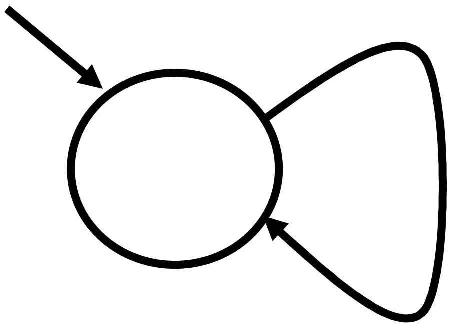
-...

Modeling randonmess

Coin Tossing



$Nats_0 \rightarrow \{\ heads, tails\ \}$



toss / heads

toss / tails

Modeling randonmess

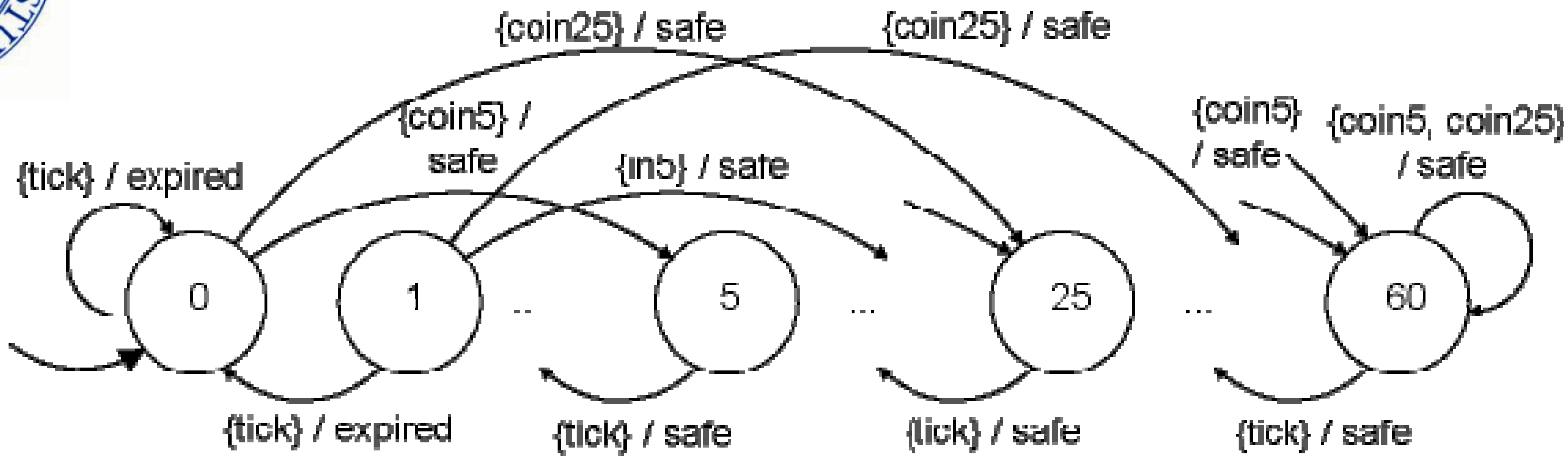
One possible behavior

Time	0	1	2	3	4
Input	toss	toss	toss	toss	toss
Output	heads	heads	tails	heads	tails

Another possible behavior

Time	0	1	2	3	4
Input	toss	toss	toss	toss	toss
Output	tails	heads	tails	heads	heads

Nondeterministic State Machines: Abstraction



This example is a deterministic model of a parking meter.

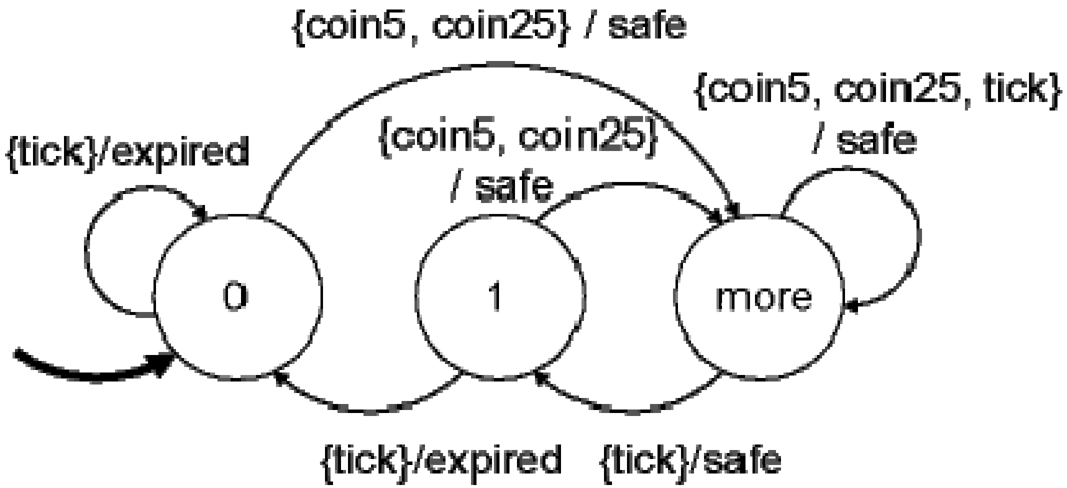
The inputs are coin5 (5 minutes) coin25 (25 minutes) and tick.

The outputs are safe and expired.

The states represent the number of minutes left until expiration.

This model of meter operation is accurate, but complicated!

Nondeterministic State Machines: Abstraction



Here is a nondeterministic model of the parking meter.

It contains less detail – it is an abstraction of the previous model.

But, some outputs are possible in this model that are impossible in the previous model:

x = coin5, tick, tick   can result in   y = safe, safe, expired

The meter expires 2 minutes after inserting a 5 minute coin!