Fundamentals of Programming
Finite State Machines

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant’Anna – Pisa

April 12, 2012
Outline

1. Finite State Machines (FSMs)
   - Introduction
   - Moore and Mealy machines
   - State Diagrams
   - Mealy machines

2. Non deterministic FSMs
   - Non determinism
   - Exercise

3. Implementing FSM in C

4. Regular Expressions

5. Hierarchical Finite State Machines
   - H-FSM specification

6. The Elevator Example
   - Simple FSM
   - Improved design
1 Finite State Machines (FSMs)
   - Introduction
   - Moore and Mealy machines
   - State Diagrams
   - Mealy machines

2 Non deterministic FSMs
   - Non determinism
   - Exercise

3 Implementing FSM in C

4 Regular Expressions

5 Hierarchical Finite State Machines
   - H-FSM specification

6 The Elevator Example
   - Simple FSM
   - Improved design
Outline

1. Finite State Machines (FSMs)
   - Introduction
     - Moore and Mealy machines
     - State Diagrams
     - Mealy machines

2. Non deterministic FSMs
   - Non determinism
   - Exercise

3. Implementing FSM in C

4. Regular Expressions

5. Hierarchical Finite State Machines
   - H-FSM specification

6. The Elevator Example
   - Simple FSM
   - Improved design
State machines are basic building blocks for computing theory.

- very important in theoretical computer science
- many applications in practical systems
- There are many slightly different definitions, depending on the application area

A state machine is a Discrete Event Discrete State system

- transitions from one state to another only happen on specific events
- events do not need to occur at specific times
- we only need a temporal order between events (events occur one after the other), not the exact time at which they occur
A deterministic finite state machine (DFSM) is a 5-tuple:

- \( S \) (finite) set of states
- \( I \) set of possible input symbols (also called input alphabet)
- \( s_0 \) initial state
- \( \phi \) transitions: a function from (state,input) to a new state
  \[ \phi : S \times I \to S \]
- \( \omega \) output function (see later)

An event is a new input symbol presented to the machine.

- In response, the machine will react by updating its state and possibly producing an output. This reaction is instantaneous (synchronous assumption).
Outline

1. Finite State Machines (FSMs)
   - Introduction
   - Moore and Mealy machines
     - State Diagrams
     - Mealy machines

2. Non deterministic FSMs
   - Non determinism
   - Exercise

3. Implementing FSM in C

4. Regular Expressions

5. Hierarchical Finite State Machines
   - H-FSM specification

6. The Elevator Example
   - Simple FSM
   - Improved design
Two types of machines:

- **Moore** output only depends on state:

  \[ \omega_{mr} : S \rightarrow \Omega \]

  Where \( \Omega \) is the set of output symbols. In this case, the output only depends on the state, and it is produced upon entrance on a new state.

- **Mealy** output depends on state and input:

  \[ \omega_{ml} : S \times I \rightarrow \Omega \]

  In this case, the output is produced upon occurrence of a certain transaction.
Moore machines are the simplest ones
If $\Omega = \{\text{yes, no}\}$, the machine is a recognizer
A recognizer is able to accept or reject sequences of input symbols
The set of sequences accepted by a recognizer is a regular language
Outline

1. Finite State Machines (FSMs)
   - Introduction
   - Moore and Mealy machines
   - State Diagrams
     - Mealy machines

2. Non deterministic FSMs
   - Non determinism
   - Exercise

3. Implementing FSM in C

4. Regular Expressions

5. Hierarchical Finite State Machines
   - H-FSM specification

6. The Elevator Example
   - Simple FSM
   - Improved design
FSM can be represented by State Diagrams

final states are identified by a double circle
In this example $l = \{a, b\}$. The following state machine recognizes string $aba$. 

![State Machine Diagram]

- $S0$ to $S1$ with input $a$
- $S1$ to $S2$ with input $b$
- $S2$ to $S3$ with input $a$
- $S3$ to $S4$ with input $b$
- $S4$ to $S0$ with input $a, b$

G. Lipari (Scuola Superiore Sant'Anna)
Example: recognizer II

- Recognize string $a^n b^m$ with $n$ even and $m$ odd (i.e. $aabbb$, $b$, $aab$ are all legal sequences, while $a$, $aabb$, are non legal)

- $S_4$ is an error state. It is not possible to go out from an error state (for every input, no transaction out of the state)

- $S_2$ is an accepting state, however we do not know the length of the input string, so it is possible to exit from the accepting state if the input continues

- If we want to present a new string we have to reset the machine to its initial state
Non regular language

- FSM are not so powerful. They can only recognize simple languages.
- Example:
  - strings of the form $a^n b^n$ for all $n \geq 0$ cannot be recognized by a FSM (because they only have a finite number of states).
  - they could if we put a limit on $n$. For example, $0 \leq n \leq 10$. 
Outline

1. Finite State Machines (FSMs)
   - Introduction
   - Moore and Mealy machines
   - State Diagrams
   - Mealy machines

2. Non deterministic FSMs
   - Non determinism
   - Exercise

3. Implementing FSM in C

4. Regular Expressions

5. Hierarchical Finite State Machines
   - H-FSM specification

6. The Elevator Example
   - Simple FSM
   - Improved design
In Mealy machines, output is related to both state and input.

In practice, output can be associated to a transition.

Given the synchronous assumption, the Moore’s model is equivalent to the Mealy’s model: for every Moore machine, it is possible to derive an equivalent Mealy machine, and viceversa.
In this example, we have a Mealy machine that
- outputs 1 if the number of symbols 1 in input so far is odd;
- it outputs 0 otherwise.

Usually, Mealy machines have a more compact representation than Moore machines (i.e. they perform the same task with a number of states that is no less than the equivalent Moore machine).
A FSM can be represented through a table

The table shown below corresponds to the parity-check Mealy FSM shown just before.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0$</td>
<td>$S_0 / 0$</td>
<td>$S_1 / 1$</td>
</tr>
<tr>
<td>$S_1$</td>
<td>$S_1 / 1$</td>
<td>$S_0 / 0$</td>
</tr>
</tbody>
</table>
Stuttering symbol

- Input and output alphabets include the **absent symbol** $\epsilon$
- It correspond to a null input or output
- When the input is **absent**, the state remains the same, and the output is **absent**
- Any sequence of inputs can be interleaved or extended with an arbitrary number of absent symbols without changing the behavior of the machine
- the absent symbol is also called the **stuttering symbol**
If no guard is specified for a transition, the transition is taken for every possible input (except the absent symbol $\epsilon$)

If no output is specified for a transition, the output is $\epsilon$

given a state $S_0$, if a symbol $\alpha$ is not used as guard of any transition going out of $S_0$, then an implicit transition from $S_0$ to itself is defined with $\alpha$ as guard and $\epsilon$ as output
Exercise

- Draw the state diagram of a FSM with $I = \{0, 1\}$, $\Omega = \{0, 1\}$, with the following specification:
  - let $x(k)$ be the sequence of inputs
  - the output $\omega(k) = 1$ iff $x(k - 2) = x(k - 1) = x(k) = 1$
three states: $S_0$ is the initial state, $S_1$ if last input was 1, $S_2$ if last two inputs were 1
Outline

1. Finite State Machines (FSMs)
   - Introduction
   - Moore and Mealy machines
   - State Diagrams
   - Mealy machines

2. Non deterministic FSMs
   - Non determinism
   - Exercise

3. Implementing FSM in C

4. Regular Expressions

5. Hierarchical Finite State Machines
   - H-FSM specification

6. The Elevator Example
   - Simple FSM
   - Improved design
Transitions are associated with:
- a source state
- a **guard** (i.e. a input value)
- a destination state
- a **output**

In *deterministic* FSM, a transition is uniquely identified by the first two.

In other words, given a source state and a input, the destination and the output are uniquely defined.
A non deterministic finite state machine is identified by a 5-tuple:

- $I$ set of input symbols
- $\Omega$ set of output symbols
- $S$ set of states
- $S_0$ set of initial states
- $\phi$ transition function:

  $$\phi : S \times I \rightarrow (S \times \Omega)^*$$

  where $S^*$ denotes the power set of $S$, i.e. the set of all possible subsets of $S$.

In other words, given a state and an input, the transition returns a set of possible pairs (new state, output).
Non determinism is used in many cases:
- to model *randomness*
- to build *more compact* automata
Non determinism

- Non determinism is used in many cases:
  - to model *randomness*
  - to build *more compact* automata

- Randomness is when there is more than one possible behaviour and the system follows one specific behavior at random
Non determinism

Non determinism is used in many cases:

- to model randomness
- to build more compact automata

Randomness is when there is more than one possible behaviour and the system follows one specific behavior at random

Randomness has nothing to do with probability! we do not know the probability of occurrence of every behavior, we only know that they are possible
Non determinism

- Non determinism is used in many cases:
  - to model randomness
  - to build more compact automata
- Randomness is when there is more than one possible behaviour and the system follows one specific behavior at random
- Randomness has nothing to do with probability! we do not know the probability of occurrence of every behavior, we only know that they are possible
- A more abstract model of a system hides *unnecessary* details, and it is more compact (less states)
We now build an automata to recognize all input strings (of any length) that end with a 01
Equivalence between D-FSM and N-FSM

- It is possible to show that Deterministic FSMs (D-FSMs) are equivalent to non deterministic ones (N-FSMs)
- Proof sketch
  - Given a N-FSM $A$, we build an equivalent D-FSM $B$ (i.e. that recognizes the same strings recognized by the N-FSM). For every subset of states of the $A$, we make a state of $B$. Therefore, the maximum number of states of $B$ is $2^{|S|}$. The start state of $B$ is the one corresponding to the $A$. For every subset of states that are reachable from the start state of state of $A$ with a certain symbol, we make one transition in $B$ to the state corresponding to the sub-set. The procedure is iterated until all transitions have been covered.
Outline

1 Finite State Machines (FSMs)
   - Introduction
   - Moore and Mealy machines
   - State Diagrams
   - Mealy machines

2 Non deterministic FSMs
   - Non determinism
   - Exercise

3 Implementing FSM in C

4 Regular Expressions

5 Hierarchical Finite State Machines
   - H-FSM specification

6 The Elevator Example
   - Simple FSM
   - Improved design
As an exercise, build the D-FSM equivalent to the previous example of N-FSM.

Figure: The N-FSM
**Solution**

![Diagram of N-FSM]

**Figure:** The N-FSM

- **Initial state:** \{S0\}

<table>
<thead>
<tr>
<th>state name</th>
<th>subset</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>q0</td>
<td>{S0}</td>
<td>{S0, S1}</td>
<td>{S0}</td>
</tr>
<tr>
<td>q1</td>
<td>{S0, S1}</td>
<td>{S0, S1}</td>
<td>{S0, S2}</td>
</tr>
<tr>
<td>q2</td>
<td>{S0, S2}</td>
<td>{S0, S1}</td>
<td>{S0}</td>
</tr>
</tbody>
</table>
Figure: The equivalent D-FSM
The first thing to do is to encode states and event symbols in the state machine. States can be simply and enumerated, so they can be represented by an integer variable.

```
#define STATE_1 1
#define STATE_2 2
#define STATE_3 3
...
int current_state;
...
```

Same thing can be done for the events.

```
#define EVENT_1 1
#define EVENT_2 2
#define EVENT_3 3
...
int event;
...
In C, an enumerated type can also be defined with the keyword `enum`

```c
enum states { STATE_1, STATE_2, STATE_3, MAX_STATES } current_state;
enum events { EVENT_1, EVENT_2, MAX_EVENTS } new_event;
```

- The C compiler maps those variables into `int`
- Therefore, it is just a notation, no new added feature
The main cycle is the following:

- When an event arrives
  - check the current state
  - depending on the event, perform an action and change state

A simple way to perform this is through a sequence of

```
if-then-else or switch-case
```

```java
switch(current_state) {
    case STATE_1 :
        if (new_event == EVENT_1) {
            // action for EVENT_1
            // change state
        } else if (new_event == EVENT_2) {
            // new action for EVENT_2
            // change state
        }
    }
    ... 

    case STATE_2 :
        if (new_event == EVENT_1) {
            // action for EVENT_1
            // change state
        }
    }
    ...
}
```
The previous implementation does not scale for large number of states and events.

A more modular implementation consists in having one separate function per action.

```c
void action_s1_e1 ();
void action_s1_e2 ();
void action_s2_e1 ();
void action_s2_e2 ();
void action_s3_e1 ();
void action_s3_e2 ();
```

In this way, functions can go in separate files.

```c
void action_s1_e1 ()
{
    /* do some processing here */
    current_state = STATE_2; /* set new state, if necessary */
}
```
All functions can be stores in a table of states-events

```c
typedef void (*ACTION)();

ACTION [MAX_STATES][MAX_EVENTS] = {
    { action_s1_e1, action_s1_e2 }, /* procedures for state 1 */
    { action_s2_e1, action_s2_e2 }, /* procedures for state 2 */
    { action_s3_e1, action_s3_e2 }  /* procedures for state 3 */
};
```

Of course, not all transitions are possible

In this case, you can define empty functions, or functions that return an error
The main program cycle is then:

```c
new_event = get_new_event (); /* get the next event to process */

if (((new_event >= 0) && (new_event < MAX_EVENTS))
    && ((current_state >= 0) && (current_state < MAX_STATES))) {
    /* call the action procedure */
    state_table[current_state][new_event] ();
}
else {
    /* invalid event/state – handle appropriately */
}
```
Consideration

- In the previous implementation, all functions act on a global variable `current_state` to modify the state.
- To make the implementation less dependent on a global variable, it is possible to pass the state and event, and return the new state variable;

```c
typedef int (*ACTION)(int, int);

int action_s1_e1(int state, int event) {
    /* do something */
    return STATE_2; /* returns the new state */
}
```

- In this way, it is also possible to write more functions that can be reused for different states and events.
Outline

1. Finite State Machines (FSMs)
   - Introduction
   - Moore and Mealy machines
   - State Diagrams
   - Mealy machines

2. Non deterministic FSMs
   - Non determinism
   - Exercise

3. Implementing FSM in C

4. Regular Expressions

5. Hierarchical Finite State Machines
   - H-FSM specification

6. The Elevator Example
   - Simple FSM
   - Improved design
Regular expressions and automata

- Regular expressions are equivalent to Finite State Automata
  - In fact, a regular expression can be translated to an automaton, by means of an appropriate parser
  - This is exactly what the `grep` program does

- Regular expression syntax (POSIX)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>matches any single character</td>
</tr>
<tr>
<td>[abc]</td>
<td>matches any of the characters within the brackets. <code>[a-z]</code> specifies a range which matches any lowercase letter from “a” to “z”</td>
</tr>
<tr>
<td>[ ^ abc]</td>
<td>matches any of the characters not within the brackets</td>
</tr>
<tr>
<td>^</td>
<td>matches the starting position of the input line</td>
</tr>
<tr>
<td>$</td>
<td>matches the ending position of the input line</td>
</tr>
<tr>
<td>*</td>
<td>matches the preceding character or expression any number of times (including 0)</td>
</tr>
<tr>
<td>+</td>
<td>matches the preceding character or expression one or more number of times</td>
</tr>
<tr>
<td>?</td>
<td>matches the preceding character or expression zero or one times</td>
</tr>
<tr>
<td></td>
<td>Or between two expressions</td>
</tr>
</tbody>
</table>
*cat* any string containing the substring *cat*

\[ [a-z]^*=[1-9][0-9]^* \] An assignment (for example \( x = 50 \))

(Here I am assuming that characters not in the event list will abort the machine)
Outline

1 finite state machines (FSMs)
   • introduction
   • moore and mealy machines
   • state diagrams
   • mealy machines

2 non deterministic FSMs
   • non determinism
   • exercise

3 implementing FSM in C

4 regular expressions

5 hierarchical finite state machines
   • H-FSM specification

6 The Elevator Example
   • Simple FSM
   • improved design
Problems with FSMs

- FSM are *flat* and *global*
- All states stay on the same level, and a transition can go from one state to another
  - It is not possible to *group* states and transitions
- Replicated transition problem:
Another problem is related to the cartesian product of two FSMs. Suppose we have two distinct FSMs that we want to combine into a single one.
- The result is a state machine where each state corresponds to a pair of states of the original machines.
- Also, each transition in corresponds to one transition in either of the two original state machines.

![Diagram showing the state machine with states S0-Q0, S0-Q1, S0-Q2, S1-Q0, S1-Q1, S1-Q2, and transitions labeled α, β, γ, δ.](image)
All these problems have to do with complexity of dealing with states.

In particular, the latter problem is very important, because we often need to combine different simple state machines.

However, the resulting diagram (or table specification) can become very large.

We need a different specification mechanism to deal with such complexity.

In this course, we will study Statecharts (similar to Matlab StateFlow), first proposed by Harel.
Outline

1. Finite State Machines (FSMs)
   - Introduction
   - Moore and Mealy machines
   - State Diagrams
   - Mealy machines

2. Non deterministic FSMs
   - Non determinism
   - Exercise

3. Implementing FSM in C

4. Regular Expressions

5. Hierarchical Finite State Machines
   - H-FSM specification

6. The Elevator Example
   - Simple FSM
   - Improved design
In H-FSMs, a state can be final or composite
A state consists of:
- An *entry* action, executed once when the system enters the state.
- An *exit* action, executed once before leaving the state.
- A *do* action, executed *while* in the state (the semantic is not very clear).

They are all optional.

**Figure:** Entry, exit and do behaviors
A transition can have:

- A *triggering event*, which activates the transition
- A *guard*, a boolean expression that *enables* the transition. If not specified, the transition is always enabled
- An *action* to be performed if the transition is activated and enabled, just after the exit operation of the leaving state, and before the entry operation of the entering state

Only the triggering event specification is mandatory, the other two are optional

**Figure:** Transition, with event, guard and action specified
A state can be decomposed into substates. When the machine enters state *Composite*, it goes into state *Comp1*. Then, if event *e2* it goes in *Comp2*, if event *e3* it goes in *Comp3*, else if event *e4* it exits from *Composite*. 
When the machine exits from a composite state, normally it 
*forgets* in which states it was, and when it enters again, it starts 
from the starting state.

To “remember” the state, so that when entering again it will go in 
the same state it had before exiting, we must use the *history* 
symbol.
AND decomposition

- A state can be decomposed in orthogonal regions, each one contains a different sub-machine
- When entering the state, the machine goes into one substate for each sub-machine

Figure: Orthogonal states for a keyboard
Outline

1. Finite State Machines (FSMs)
   - Introduction
   - Moore and Mealy machines
   - State Diagrams
   - Mealy machines

2. Non deterministic FSMs
   - Non determinism
   - Exercise

3. Implementing FSM in C

4. Regular Expressions

5. Hierarchical Finite State Machines
   - H-FSM specification

6. The Elevator Example
   - Simple FSM
   - Improved design
Outline

1. Finite State Machines (FSMs)
   - Introduction
   - Moore and Mealy machines
   - State Diagrams
   - Mealy machines

2. Non deterministic FSMs
   - Non determinism
   - Exercise

3. Implementing FSM in C

4. Regular Expressions

5. Hierarchical Finite State Machines
   - H-FSM specification

6. The Elevator Example
   - Simple FSM
   - Improved design
Let’s define an “intelligent” elevator

- For a 5-stores building (ground floor, and four additional floors)
- Users can “reserve” the elevator
- The elevator serves all people in order of reservation

We assume at most one user (or group of users) per each “trip”, and they all need to go to the same floor
Design considerations

How do you encode at which floor the elevator is?

1. One different state per each floor
   - Does not scale well; for 100 floors building, we need 100 states!

2. The floor is encoded as an *extended* state, i.e. a variable \( cf \)
   - It scales, but more difficult to design

3. It always depends on what we want to describe!

Which events do we have?

- An user press a button to “reserve” the elevator, setting variable \( rf \)
- An user inside the elevator presses the button to change floor, setting variable \( df \)
First design

elevator_machine

<<machine>>

Idle

Destination reached

Move To Reserve

Move to destination

Ready to Load

reserve [cf = rf]

reserve [cf == rf]

motor stop

press button

timeout

motor stop

timeout
Outline

1. Finite State Machines (FSMs)
   - Introduction
   - Moore and Mealy machines
   - State Diagrams
   - Mealy machines

2. Non deterministic FSMs
   - Non determinism
   - Exercise

3. Implementing FSM in C

4. Regular Expressions

5. Hierarchical Finite State Machines
   - H-FSM specification

6. The Elevator Example
   - Simple FSM
   - Improved design
Doors

- The previous design does not capture all aspects of our systems.
- Let's start to add details by adding the description of how the doors behave.
- Abstraction level
  - The level of details of a design depends on what the designer is more interested in describing with the specification.
  - In the previous design, we were not interested in describing all aspects, but only on giving a few high-level details.
  - The design can be refined by adding details when needed.
The doors submachine

doors_closed

opening

closing

doors_open

close_end

open_end

close_doors

open_doors

<<machine>>

doors_closed

doors_open
The elevator, second design

elevator_machine
<<machine>>

Idle
entry / close doors

Destination reached
entry / open doors

Move To Reserve
reserve [cf != rf]
reserve [cf == rf]
motor stop

elevator

Ready to Load
entry / open doors

Motor stop

go

timeout

Ready to Load
entry / open doors

Move to destination
entry / close doors

motor stop

Destination reached
entry / open doors

timeout

Idle
entry / close doors

G. Lipari (Scuola Superiore Sant'Anna)
Tree and Heap
April 12, 2012 65 / 66
Putting everything together

Global Elevator

<<machine>>

Idle

entry / close doors

Destination reached

entry / open doors

timeout

Move To Reserve

motor stop

reserve [cf != rf]

Ready to Load

entry / open doors

timeout

go

Move to destination

entry / close doors

motor stop

reserve [cf == rf]

doors_closed

close_end

opening

open doors

doors_open

close doors

open_end