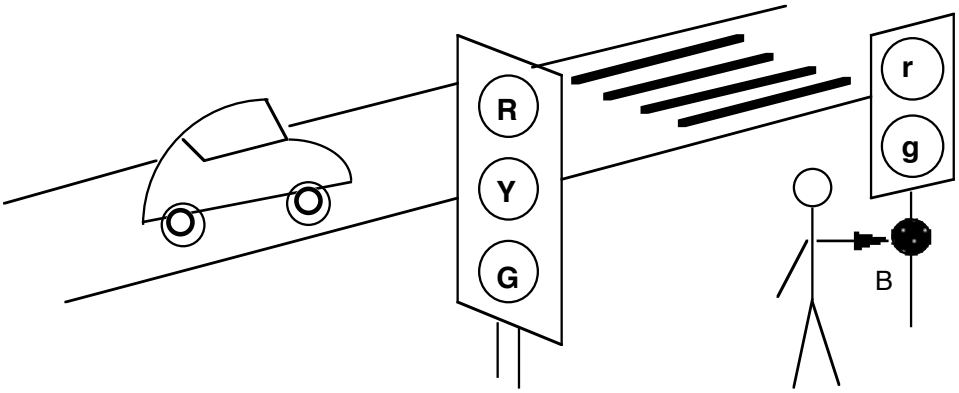
2. Finite state machines (fsm, sequential machines): examples and applications

Goal of this chapter: fsm's are everywhere in our technical world! Learn how to work with them.

2.1 Example: Design a finite state controler to synchronize traffic lights

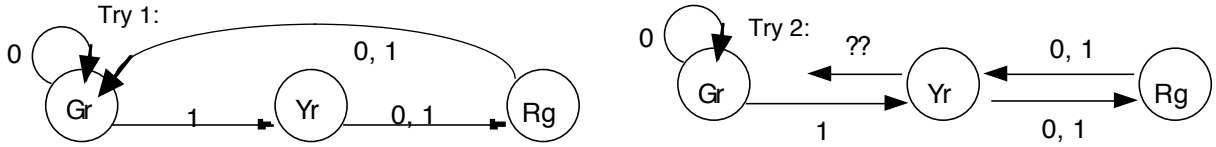
Finite state machines are the most common controlers of machines we use in daily life. In the example illustrated by the figure, the intersection of a main road with a pedestrian crossing is controled by two traffic lights. The traffic light for cars has 3 possible values: red R, yellow Y, and green G. The traffic light for pedestrians has 2 possible values: red r and green g. Your task is to design a finite state controler that turns the various lights on and off in a reasonable and safe manner.



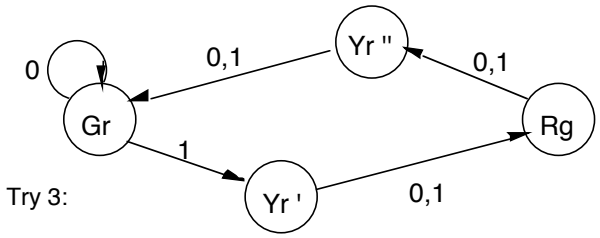
The controler reacts to an input signal B that has 2 possible values, 1 (pressed) and 0 (not pressed), depending on whether or not a pedestrian has pressed the button B in the preceding time interval. This signal is sampled at regular time intervals of, say, 10 seconds, and the controler executes a transition once every 10 seconds. The time interval of 10 seconds is assumed to be sufficient to cross the road.

When first confronted with the task of describing the behavior of such a system, most people resort to scenarios involving time sequences: "if the button is pressed, after 10 seconds, this or that happens, and then ...". This approach is unmanageable for a realistic controler whose behavior depends on a variety of sensors, as the number of scenarios grows exponentially with the number of sensor readings. We need a static description that captures all conceivable scenarios, in order to prove invariants that state certain configurations or events must, or must never, occur. The concept of a state space of the system is a standard technique that compresses all possible sequences of states (in general, infinitely many and of unbounded length) into a description of fixed size. Let us design a state space for controling the traffic light system shown above. There is no unique best solution to this problem - different designs might have different advantages and disadvantages.

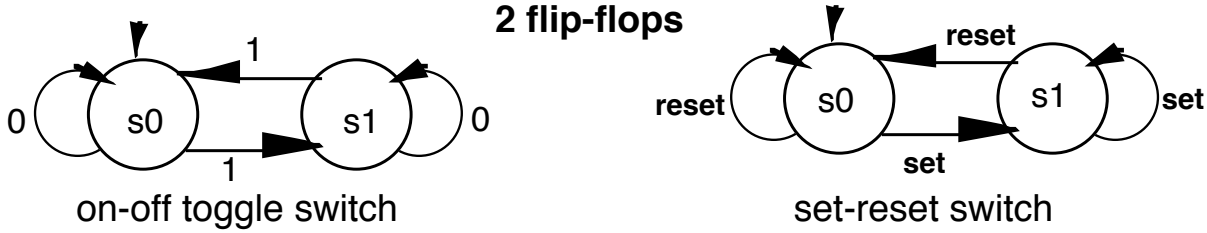
First, we list all the states that this traffic light system is potentially capable of signaling to an observer, and comment on the desirability and role of each such visible state. Among the six distinct visible states, namely: Rr, Rg, Yr, Yg, Gr, Gg, we can immediately rule out Gg, a killer, and Yg, as too dangerous. Rg and Gr ar the two alternating states that assign the right of way to one or the other participant. Rr might make sense if it never lasts very long, but is unnecessary since we introduced the signal Y, yellow, to give cars time to break before pedestrians see green g. Thus, we introduce a state Yr, as shown in the figure below.



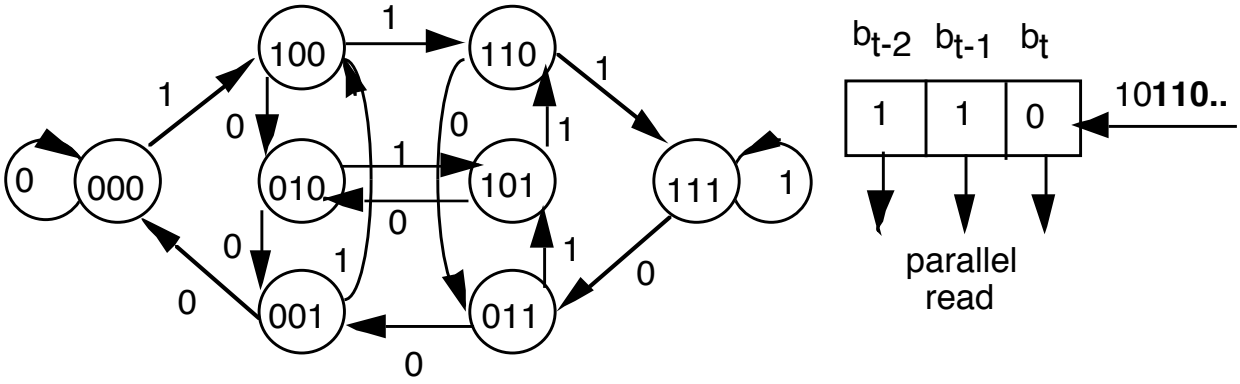
The solution at left (Try 1) has the problem that it relies on the courtesy of drivers to delay take-off until all pedestrians complete their crossing. If we try to use the delay state Yr in both phases (Try 2), in the transition from Rg back to Gr as we did on the way from Gr to Rg, we run into a technical problem: on both input values 0 and 1, the transitions out of Yr have already been set to lead to Rg. The solution below (Try 3) recognizes the fact that the total state of the system is not necessarily limited to to the visible state. When the lights show Yr, we need an additional bit of memory to remember whether the previous state was Gr or Rg. Thus, we introduce 2 delay states, which look identical to an observer, but are distinguished internally.



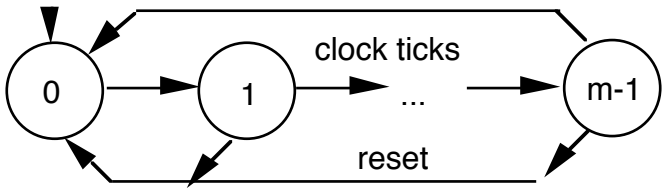
2.2 Simple finite state machines encountered in light switches, wrist watches, ticket vending machines, computer user interfaces, etc



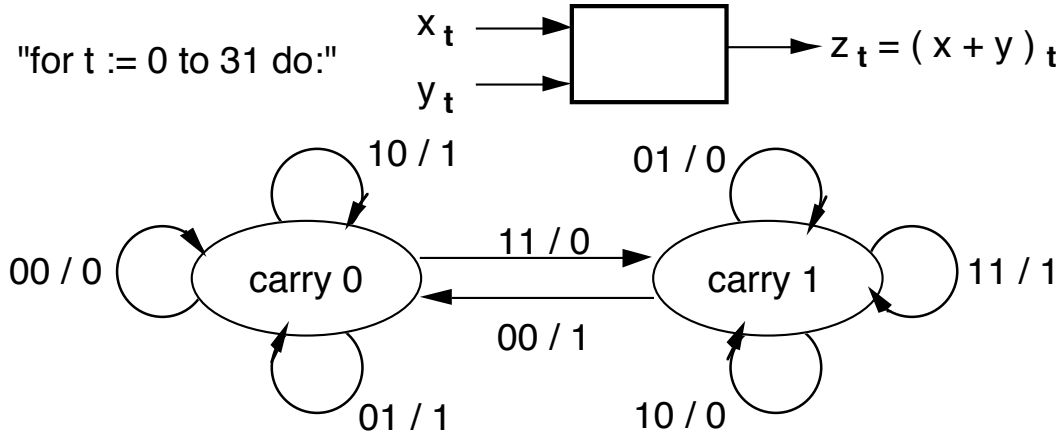
Shift register: save the last 3 bits of a data stream:



Counter mod m with reset (e.g. clock):

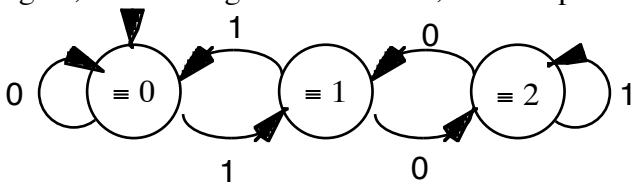


Serial binary adder:



FSMs are standard system components for performing arithmetic on **numbers of fixed size**, say 32 bits, but they cannot do much more than addition and subtraction on numbers of arbitrary size.
Exercise: Show that there is no fsm multiplier for numbers of arbitrary size.

Mod 3 divider:
Read a binary integer "left to right", i.e. most significant bit first, and compute its remainder mod 3.

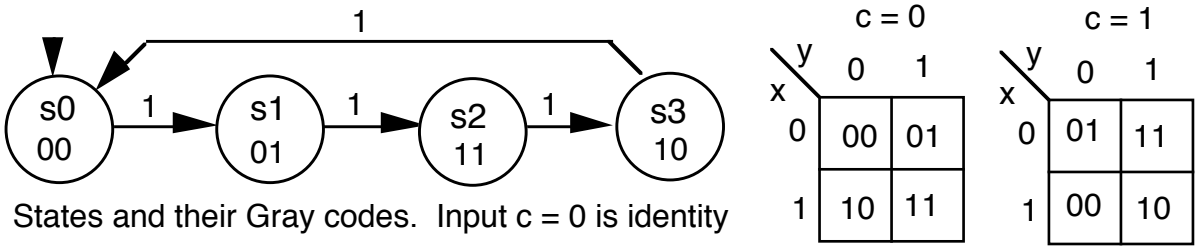


Justification. Consider an integer with the binary representation L b, where L is a bitstring and b is a single bit. Let |L| denote the integer represented by L, and similarly for |L b| . Assume L has been read, most significant bit first, and, by way of example, the fsm is now in state '≡1', meaning that |L| mod 3 = 1. Then L 0 represents the integer |L 0| = 2 |L|, where |L 0| mod 3 = 2; and L 1 represents |L 1| = 2 |L| + 1, where |L 1| mod 3 = 0. This justifies the two transitions out of state '≡1'. We can turn this fsm into an acceptor by designating some state(s) as accepting, and thus recognize any language that is a union of residue classes mod 3. E.g. L0 = { x | |x| mod 3 = 0}, or L12 = { x | |x| mod 3 ≠ 0}, where |x| is the integer denoted by x.

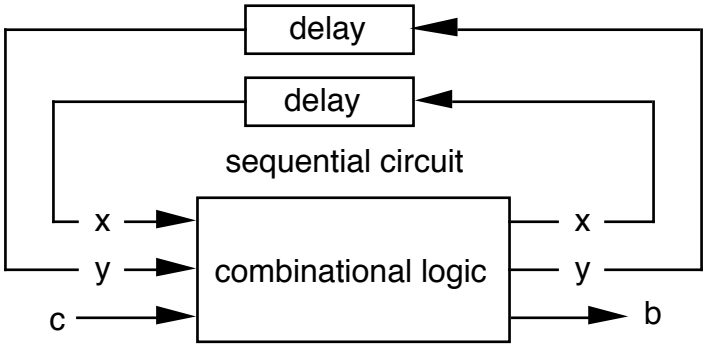
Exercise: Design a mod 3 divider that reads binary integer "right to left", i.e. least significant bit first.

Sequential circuit design

Logic design for a mod 4 counter. State assignment using Gray codes (each transition changes 1 single bit).

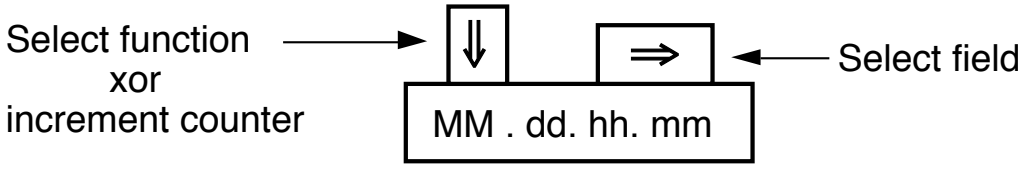


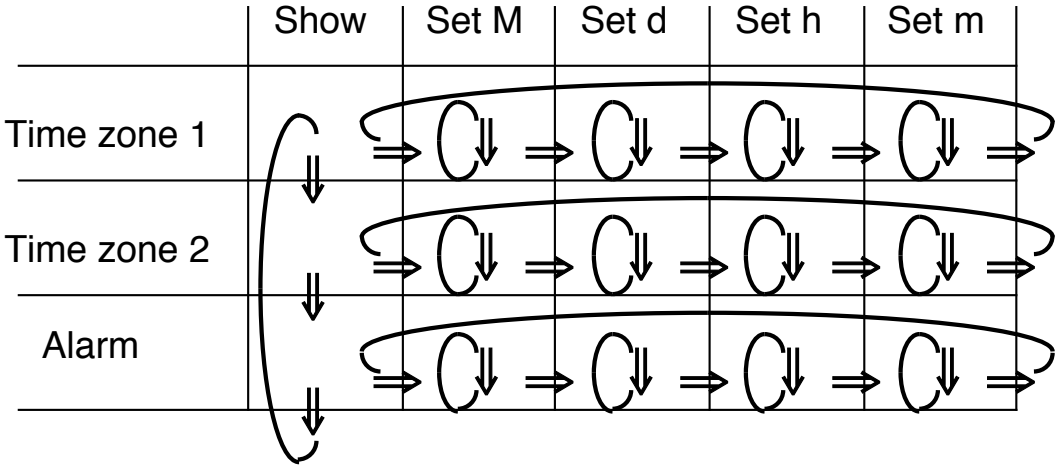
State assignement: 2 boolean variables x, y code the states: s0: 00, s1: 01, s2: 11, s3: 10.
Input c: count the number of 1's, ignore the 0s. Ouput b: b = 1 in s3, b = 0 otherwise.
Combinational logic: $x := \neg c \wedge x \vee c \wedge y$, $y := \neg c \wedge y \vee c \wedge \neg x$, $b := x \wedge \neg y$



2.3 Design a digital watch interface: State space as a Cartesian product

Goal: 2 buttons suffice, no operating manual is needed.



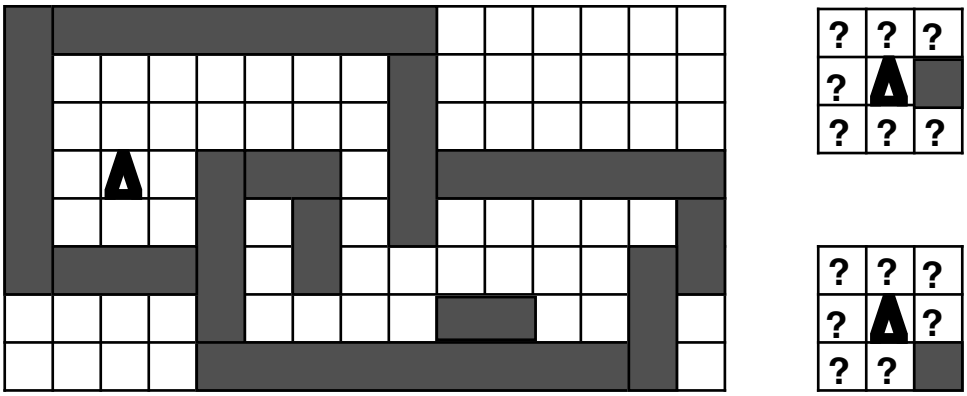


This is the state space (that should be) visible to the user, e.g. by displaying the function (e.g. "TZ2") and by flashing the active field. The machine's internal state space of course is much more complex. Each field is a counter: mod 12, mod 31, mod 24, mod 60.

Design principle: the state space visible to the user is a homomorphic image of the internal state space.

2.4 The wall-hugging robot

Program a toy robot to follow a wall. The figure below illustrates the assumptions. In a simplified version of "turtle geometry" [Papert], the robot's world is a grid, any square is either free or occupied by a "wall". The robot, shown as a triangle, is placed on a free square facing in one of four possible directions. The robot has two binary sensors: **h** ("head", a touch sensitive bumper) signals whether the square in front of the robot is free (0) or occupied by a wall (1); **r** (right) provides the same information for the square to the right of the robot. The robot is capable of two primitive actions: **R** turns right by 90° while remaining on the same square; **F** (forward) advances one square in the robot's current direction.



a) The city wall and b) the predicate "wall to the rear-right"

The robot must be programmed to find a piece of wall, then endlessly cycle along the inside of the wall, hugging it with his right side. The following finite state machine, presented in tabular form, solves the problem.

Seek: precondition = true, postcondition = "wall to the rear-right"

r	**h**	**Actions**	**Next state**
1	-		Track
-	1	RRR	Track
0	0	F	Seek

Track: precondition = postcondition = "wall to the rear-right"

r	**h**	**Actions**	**Next state**
0	-	RF	Track
1	0	F	Track
1	1	RRR	Track

The robot starts in state Seek, not knowing anything about its position, as expressed by the vacuous assertion "precondition = true". The function of this state is to bring about the postcondition "wall to the rear-right", illustrated in Fig.b: There is a piece of wall to the right of the robot, or diagonally to the rear-right, or both. As long as the robot has not sensed any piece of wall (**r** = 0, **h** = 0) it moves forward with F. If it senses a wall, either ahead or to the right, it positions itself so as to fulfill the postcondition "wall to the rear-right". With the "don't care" notation "-" we follow the design principle "specify behavior only as far as needed" and make the robot non-deterministic.

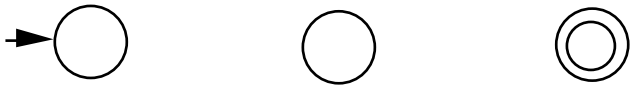
The mathematically inclined reader may find it instructive to prove the program correct by checking that the state Track maintains the invariant "wall to the rear-right", and that, in each transition, the robot progresses in his march along the wall. Here we merely point out how several fundamental ideas of the theory of computation, of algorithms and programs can be illustrated in a simple, playful setting whose rules are quickly understood.

2.5 Varieties of finite state machines and automata: definitions

Notation: Alphabet A = {a, b, ..} or A = {0, 1, ..}. A* = { w, w', ...}. Nullstring ε. Empty set {} or Ø.
"Language" L ⊆ A*. Set S = {s, s', .., s0, s1, ..}. Cardinality |S|. Power set $\mathbf{2^S}$. "¬" not or complement.

Deterministic Finite Automaton (FA, DFA), **Finite State Machine** (FSM): M = (S, A, f, s0, ...)
Set of states S, alphabet A, transition function f: S x A -> S, initial state s0.
Other components of M, indicated above by dots "...", vary according to the purpose of M.

Acceptor (the standard model in theory)**:** M = (S, A, f, s0, F), where F ⊆ S is a set of accepting or final states.



Notation: starting state (arrow), non-accepting state, accepting state

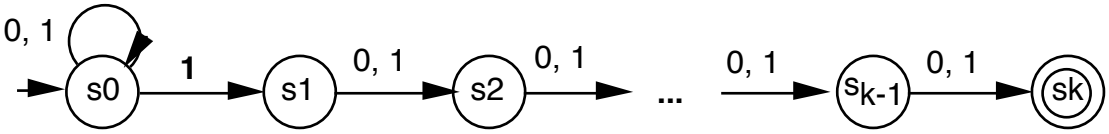
Extend f from S x A -> S to f: S x A* -> S as follows: f(s, ε) = s, f(s, wa) = f(f(s, w), a) for w ∈ A*
Df: **M accepts w** ∈ A* iff f(s0, w) ∈ F. Set L ⊆ A* accepted by M: **L(M)** = { w | f(s0, w) ∈ F}.

Transducer (fsm's used in applications)**:** M = (S, A, f, g, s0), with a function g that produces an output string over an alphabet **B**: g: S -> B (Moore machine), h: S x A -> B (Mealy machine)
An acceptor is the special case of a transducer where F(s) = 1 for s ∈ F, F(s) = 0 for s ∉ F.

Non-deterministic Finite Automaton (NFA) **with ε-transitions: f: S x (A ∪ {ε}) -> $\mathbf{2^S}$.**
Special case: NFA without ε-transitions: **f: S x A -> $\mathbf{2^S}$.**
Variation: **Probabilistic FA**: An NFA whose transitions are assigned probabilities.

Extend f: S x A* -> 2^S: f(s, ε) = ε-hull of s = all states reachable from s via ε-transitions (including s);
f(s, wa) = ∪ f(s', a) for s' ∈ f(s, w).
Extend f further f: 2^S x A* -> 2^S as follows: f(s1, .., sk, a) = ∪ f(si, a) for i = 1, .., k.
Df: **M accepts w** ∈ A* iff f(s0, w) ∩ F ≠ {}. Notice: w is accepted iff ∃ some w-path from s0 to F.
Set L ⊆ A* accepted by M: L(M) = { w | f(s0, w) ∩ F ≠ {}}.

A non-deterministic machine spawns multiple copies of itself, each one tracing its own root-to-leaf path of the tree of all possible choices. Non-determinism yields an **exponential increase in computing power**!
Ex 1: L = (0 ∪ 1)* 1 (0 ∪ 1)$^{k-1}$ i.e. all strings whose k-th last bit is 1. A DFA that accepts L must contain a shift register k bits long, with 2^k states as shown in 2.2 for k=3. A NFA accepts L using only k+1 states, by "guessing" where the tail-end k bits start. This example shows that simulating a NFA by a DFA may require an **exponential increase in the size of the state space.**

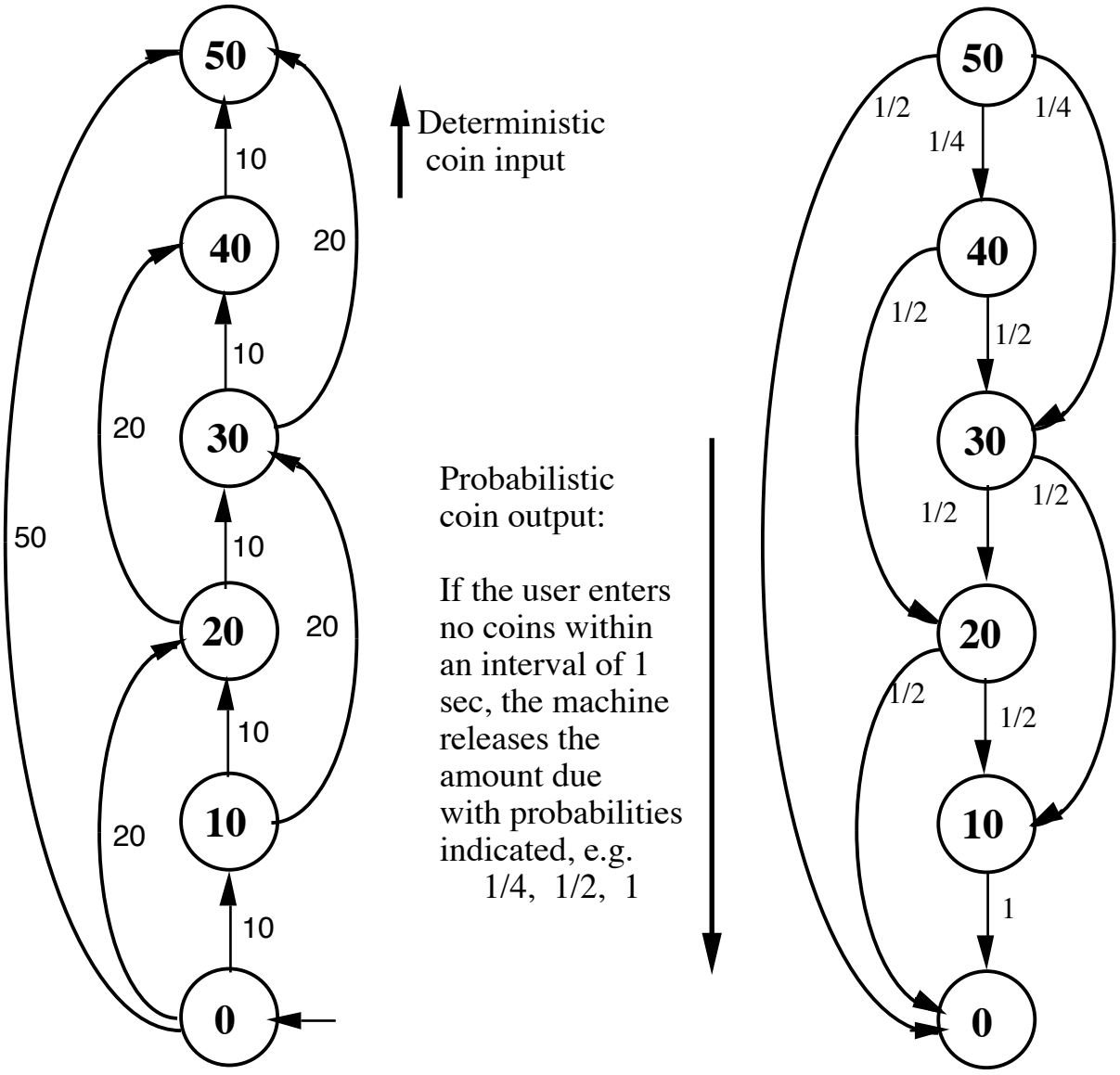


NFA accepts the language of strings whose k-th last bit is 1 - by "guessing"

2.6 Probabilistic coin changer

Many utility machines (vending or ticket machine, watch, video cassette recorder. etc.) are controled by an fsm. Understanding their behavior (user interface) frequently requires a manual, which we usually don't have at hand. A diagram of the fsm, perhaps animated, would often help the novice user to trace the machine's behavior. As an example, imagine a coin changer modeled after gambling machines, whose display looks as shown in the following figure.

The states correspond to the amount of money the machine owes you, and the current state lights up. As long as you enter coins rapidly, the machine accumulates them up to a total of 50 cents. If you pause for a clock interval, the machine starts emitting coins randomly, to the correct total. After a few tries you are likely to get useful change: either breaking a big coin into smaller ones, or vice- versa.



Exercise: Mathematically analyze, or estimate by simulation, the average number of tries (times you need to enter a coin) in order to obtain a desired change. E.g., to change a 50 cent coin into five 10 cent coins.

Exercise: Logic design for the fsm "binary integer mod 3" shown in 2.2.

Hw2.1: Design an fsm that solves the problem "binary integer mod 3" when the integer is read least significant bit first (as opposed to most significant bit first as shown in 2.2).

Hw2.2: Analyze the behavior of the ticket machines used by Zurich's system of public transportation, VBZ. Draw a state diagram that explains the machines' behavior. Evaluate the design from the user's point of view.

End of Ch2