Introduction to Finite State Automata and Machines

Finite state automata are discrete system models foundational to computer science and increasingly adopted by the dynamics and control community for the purpose of higher-level control in which information is most naturally represented by finite sets of discrete values or modes.

Deterministic Finite State Automata (DFSA)

A DFSA receives input strings over alphabet Σ and returns true (acceptance) if the input string is in L^* (selectively defined with or without the empty string ε). In other words, if there is at least one sequence of "words" each in language L that can be concatenated to form the input string that input string is accepted (the DFSA returns true). Else the DFSA returns false.

In ***deterministic*** finite state automata, the input string must always return the same true/false answer. This requires that there must be exactly one unique path through the automaton graph for each input string. As will be shown below, ***nondeterministic*** finite state automata can have zero or multiple paths through the graph for any input string.

The deterministic finite state automaton (DFSA) is defined as the tuple $D = (Q, \Sigma, \delta, q_0, F)$ where:

- Q is the finite non-empty set of states (graph nodes)
- Σ is the finite non-empty alphabet
- $q_0 \in Q$ is the unique initial state
- $F \subseteq Q$ is the finite and possibly empty set of final states[1]
- $\delta = Q \times \Sigma \rightarrow Q$ is the total transition function mapping single input characters and the 'current' state to a new state. The transition function δ defines the DFSA graph edges.

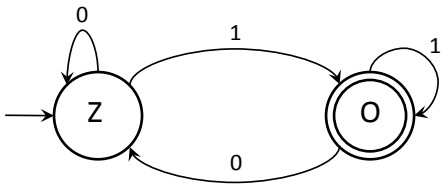
Input to a DFSA is parsed left to right, one alphabet character at a time. The system starts in state q_0. The first (leftmost) input symbol is read then removed from the input string, then the system transitions along the graph edge defined by δ to the next state. This procedure continues until the remaining input string is empty, at which time the DFSA returns "true" if it is in a state contained in final state set F and "false" otherwise. A DFSA function (implementation) takes the machine D and input string $s \in \Sigma^*$ as input and returns true (string accepted) or false (string not accepted).

Three simple DFSA examples are shown below. Formal definitions of each DFSA are accompanied by figures showing the same DFSA. In each figure, the initial state is given by an arrow (from nowhere) leading into the state (typically from the left), transitions are labeled by the input character to which they correspond, and each final state is depicted by a double-circle. Note

[1] While F could technically be Ø or the full set Q, such a machine design wouldn't be very useful as it would always return true (when F = Q) or false (when F is the empty set).

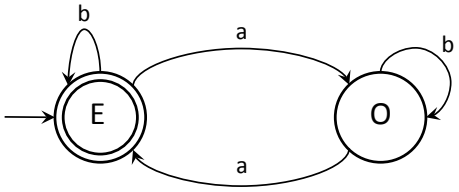
that for a DFSA each state must have exactly one outgoing transition for each character in Σ to ensure there is exactly one unique path to follow for each possible input string.

Example 1: Define a DFSA accepting all string ending in 1.



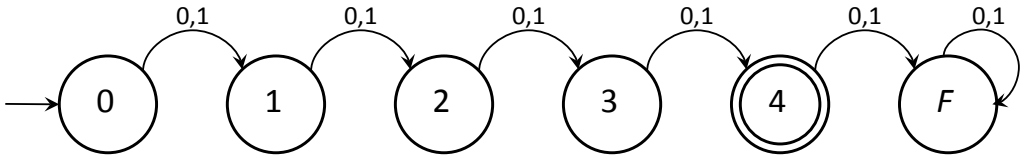
This DFSA requires two states, Q={Z,O}. State Z (zero) represents strings that either have 0 as the last character or the empty string. State O (one) "remembers" that the last character in the string parsed thusfar was 1. It is useful to give the states defined in Q intuitive names. The DFSA is then formally defined by: Q={Z,O}, Σ={0,1}, q_0=Z, F={O}, δ = {<Z,0,Z>, <Z,1,O>, <O,0,Z>,<O,1,O>}.

Example 2: Define a DFSA accepting all strings with an even number of *a*'s; assume 0 is an even number.



The DFSA for this example is formally defined by: Q={E,O}, Σ={*a,b*}, q_0=E, F={E}, δ = {<E,*a*,O>, <E,*b*,E>, <O,*a*,E>,<O,*b*,O>}.

Example 3: Define a DFSA that accepts only 4-bit sequences from Σ={0,1}.



The DFSA for this example is formally defined by: Q={0,1,2,3,4,*F*}, Σ={0,1}, q_0=0, F={4}, δ = {<0,0,1>, <0,1,1>, <1,0,2>, <1,1,2>, <2,0,3>, <2,1,3>, <3,0,4>, <3,1,4>, <4,0,*F*>, <4,1,*F*>, <*F*,0,*F*>, <*F*,1,*F*>}. Note that in this example, state F is an "absorbing state" in that there is no path out of this state once reached. Absorbing states that are not also final states are called "failure states" since, once reached, the string will not be accepted regardless of additional input received.

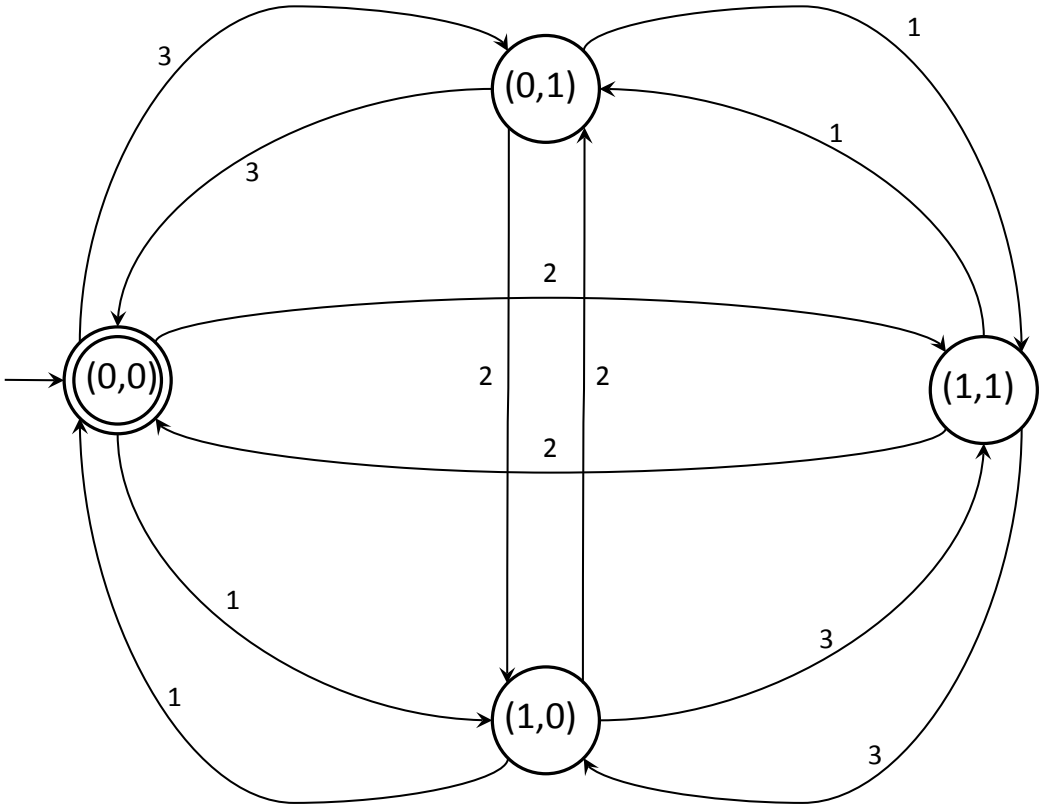
Example 4: The Three-Light Switch (with Σ={1,2,3} indicating "switch flip" actions).

Consider a system in which three light switches denoted 1, 2, and 3 are used to control two lights denoted N (North light) and S (South light). The switches are connected to the lights such that:

1. Switch 1 toggles light N on/off.
2. Switch 2 toggles both lights on/off.
3. Switch 3 toggles light S on/off.

Assume both lights are initially off.

For this system, we describe each state by a pair (n,s) denoting whether the north (n) and south (s) lights are on (1) or off (0). For example, state (0,0) would have both lights off. Transitions are then defined per the above rules.



The DFSA for this example is formally defined by Q={(0,0), (0,1), (1,0), (1,1)}, Σ={1,2,3}, q_0=(0,0), F={(0,0)}, δ = {<(0,0),1,(1,0)>, <(0,0),2,(1,1)>, <(0,0),3,(0,1)>, <(0,1),1,(1,1)>, <(0,1),2,(1,0)>, <(0,1),3,(0,0)>, <(1,0),1,(0,0)>, <(1,0),2,(0,1)>, <(1,0),3,(1,1)>, <(1,1),1,(0,1)>, <(1,1),2,(0,0)>, <(1,1),3,(1,0)>}

Nondeterministic Finite State Automata (NDFSA)

A nondeterministic finite state automaton (NDFSA) is defined as:

M = (K, Σ, S, F, Δ) where

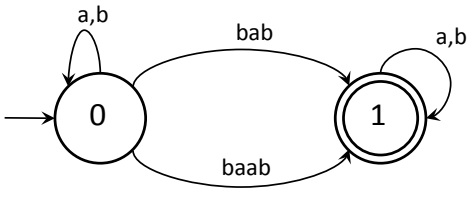
- K is the finite non-empty set of states
- Σ is the finite non-empty alphabet
- S ⊆ K is the finite and non-empty set of initial state(s)
- F ⊆ K is the finite and possibly empty set of final states
- Δ = K × Σ* → K is the total transition function mapping zero or more input characters and the 'current' state to a new state. The transition function Δ defines the NDFSA graph edges.

The NDFSA conceptually is the same as a DFSA except that zero or multiple paths can exist through the NDFSA for any string received. This in turn allows the transition map to include zero, one, or multiple paths out of each state when a particularly alphabet character or character sequence is received. Transitions can also be labeled as empty ($\mathcal{E}$) meaning the system could remain in the current state or transition to a new state via the empty string transition without reading any new character(s) from the input string.

An NDFSA takes the machine M and input string $s \in \Sigma^*$ as input and returns true (string accepted) or false (string not accepted). To parse a string through an NDFSA on a computer, all possible paths through the NDFSA must be explored; if at least one of the possible paths through the NDFSA returns acceptance (true) the entire NDFSA is said to accept the string (return true). A recursive implementation of the NDFSA is recommended because it enables a compact source code.

Example:

Design a NDFSA M accepting the language L = {*x* | *x* ∈ {*a,b*} ∧ *x* contains *bab* or *baab*}



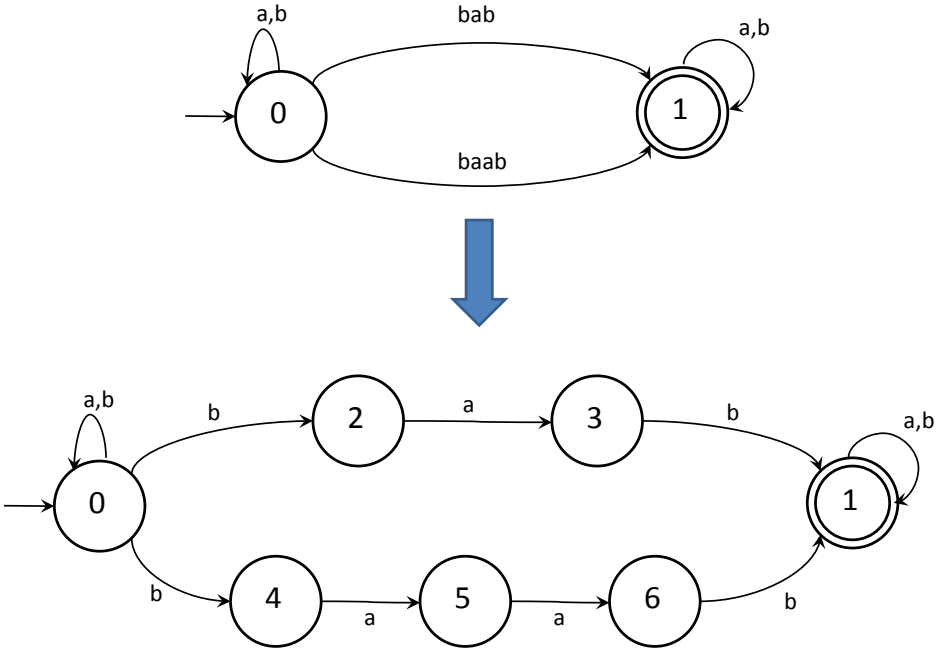
The NDFSA for this example is formally defined by K={0, 1}, Σ={*a, b*}, S={0}, F={1}, Δ = {<0,a,0>, <0,b,0>, <1,a,1>, <1,b,1>, <0,bab,1>, <0,baab,1>}.

All NDFSA can be reformulated as a DFSA, although the DFSA may be require more states and transitions to represent the same language. Below is an algorithm to convert from NDFSA to DFSA. The NDFSA example listed above is carried through the algorithm to show the process.

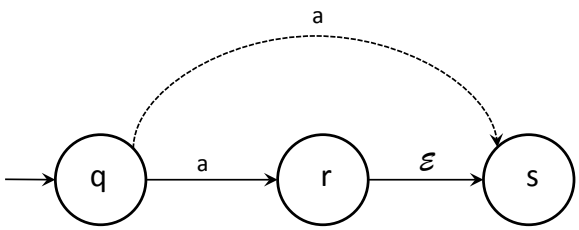
Algorithm for Conversion from NDFSA to DFSA

Step 1: "Normalize" machine M. A normalized FSA extracts exactly one input character per transition.

a. Eliminate multi-character transitions.



b. Eliminate empty string state transitions (a general case is shown here since our specific example did not contain empty string transitions). In the example illustrated below, since ($<q, a, r> \wedge <r, \varepsilon, s>$)) $\in \Delta$ we add $<q, a, s>$ to the transition set then can remove the transition $<r, \varepsilon, s>$ without loss of generality.



Step 2: Construct a "power set" DFSA from the NDFSA starting with the normalized (M') version of NDFSA M (from Step 1). This construction process does not guarantee a minimum number of states for the DFSA but it does guarantee the DFSA will accept the same language as did the

NDFSA if the construction process is performed correctly. This process is shown below for our NDFSA example. This example is quite involved because of the number of states but illustrates the full construction process which ideally would be performed by a computer (not by hand).

Example: Convert the NDFSA for the language L = $\{x \mid x \in \{a,b\} \wedge x$ contains *bab* or *baab*$\}$ (shown above) to a DFSA using the above conversion algorithm.

Per step 1, M normalizes to form NDFSA M' with states K' = {0,1,2,3,4,5,6}.

The following narrative describes DFSA construction per step 2 of the above algorithm:

For our example machine M', the full power set of states is given by:

$$2^{K'} = \{ \emptyset,\{0\},\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\{0,1\},\{0,2\},\ldots,\{0,1,2,3,4,5,6\}\}$$

The power set of K' represents all possible states that may be needed for the DFSA. However, since the transition map is sparse in our example, we will not expect to need all these states in our final DFSA. The initial state in the new DFSA is the [unique] power set state containing all possible initial states in S' of NDFSA M'. The empty set state $\emptyset$ in the new DFSA represents an absorbing failure state since reaching this state indicates there was no valid path through the NDFSA for that input string. Transitions in the new DFSA go from each current state to the new power set state that collectively represents all possible final states in the NDFSA for the single input string character being parsed. While the power set may itself be quite large, as is the case in our example, the final DFSA generated must keep members of power set state set only if there is some path from the initial state to that power set element.

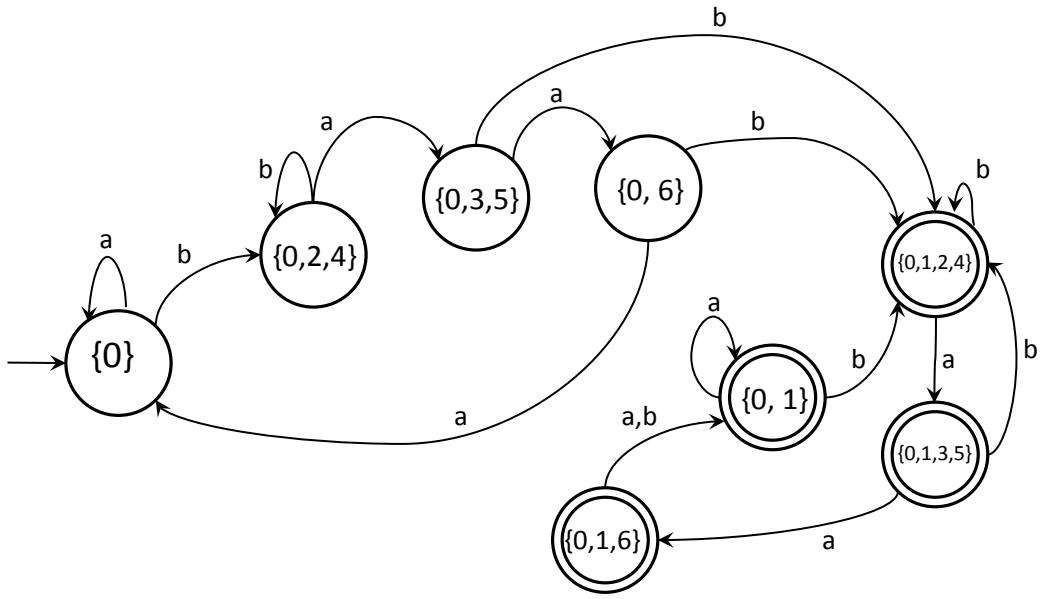
The new DFSA is shown in the figure below, following a narrative of the transition function construction process.

The initial state in the new DFSA is uniquely specified by power set state {0} since state 0 was the only initial state in M'. If an *a* is received in {0}, the only transition option is to remain in state {0}. If a *b* is received in {0}, the system may remain in state 0 or transition to state 2 or 4 in M'. This is represented in the new DFSA with a transition from {0} to power set state {0, 2, 4}. From state {0, 2, 4}, receiving an *a* can lead to state 0 (when transitioning from state 0), it can transition to state 3 (from state 2), or it can transition to state 5 (from state 4). From state {0, 2, 4}, receiving a *b* can only lead to state {0} (from state 0) as states 2 and 4 in M' have no outgoing transition when *b* is received. From state {0, 3, 5}, receiving an *a* can lead to state 0 (when transitioning from state 0) or it can transition to state 6 (from state 5); state 3 has no outgoing transition when *a* is received. From state {0, 3, 5}, receiving a *b* can lead to state 0, 2, or 4 (from state 0) or state 1 (from state 3); state 5 has no outgoing transition when *b* is received. Because 1 is a final state in M', the state {0, 1, 2, 4} will also be a final state in the new DFSA, i.e., reaching state {0, 1, 2, 4} indicates that at least one path in M' can reach this state with a state value of 1 from M'. From state {0, 6}, receiving an *a* leads to state 0 (from state 0); state 6 has no outgoing transition when a is

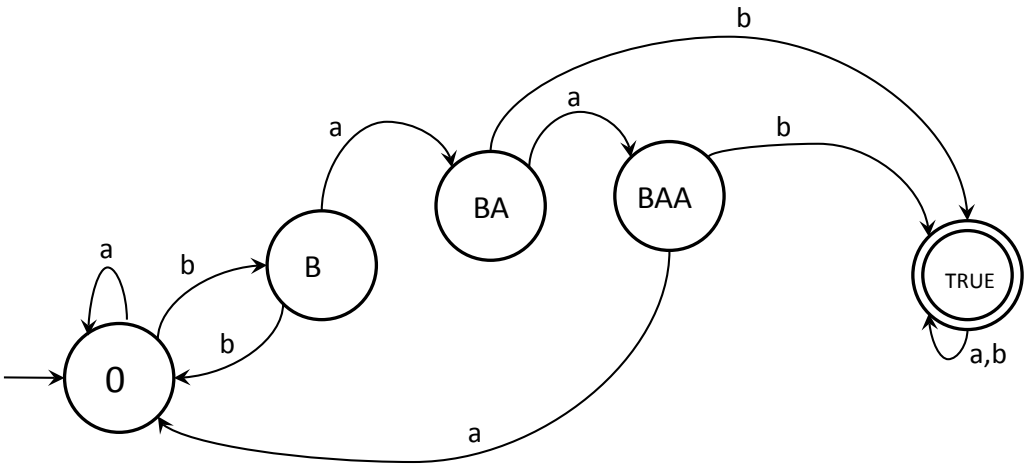
received. From state {0, 6} receiving a *b* can lead from state 0 to state {0, 2, 4}, and receiving a b from state 6 leads to state 1, so we add a transition from {0,6} to {0, 1, 2, 4}. From state {0, 1, 2, 4}, receiving an *a* can lead to state 0 from state 0, state 1 from state 1, state 3 from state 2, and state 5 from state 4, creating a transition to a new power set state {0, 1, 3, 5} which is also a final state. From state {0, 1, 2, 4}, receiving a *b* can lead to state 0 from state 0; state 1 from state 1; no outgoing transitions are present when a *b* is received in states 2 or 4.

From state {0,1}, receiving an *a* can lead from state 0 to state 0 and from state 1 to state 1, defining a reflexive transition back to state {0,1}. From state {0,1} receiving a *b* can lead from state 0 can lead to state 0, 2, or 4 from state 0 or lead to state 1 from state 1, defining a transition to existing power set state {0, 1, 2, 4}. From state {0, 1, 3, 5}, receiving an *a* can lead from state 0 to state 0, from state to state 1, from state 5 to state 6; no transition exists out of state 3 when *a* is received. A new state {0, 1, 6} must therefore be created. From state {0, 1, 3, 5}, receiving a *b* can lead from state 0 to state 0, 2, or 4, from state 1 to state 1, and from state 3 to state 1; no transition exists out of state 5 when *b* is received. From state {0, 1, 6}, receiving an *a* can lead from state 0 to state 0 and from state 1 to state 1; no transition exists out of state 6 when *a* is received. From state {0, 1, 6}, receiving a *b* can lead from state 0 to state 0, from state 1 to state 1, and from state 6 to state 1.

The DFSA shown below has a total of 8 states. A computer can generate this DFSA D from NDFSA M using the two-step process. The result is a DFSA that correctly accepts the same language L but that is not guaranteed to have the minimum number of states. For this example, we generated four final states and transition between them per the above algorithm. However, we can see by inspection that all four of these final states could indeed be collapsed into a single absorbing final state since once states {0, 1, 2, 4} is reached the DFSA will always return a value of "true". The subsequent figure illustrates the simplified machine generated by combining the four final states. Further analysis of graphs including algorithms for their simplification and reduction is beyond the scope of this chapter; there is significant literature on this topic, however, as efficiency in graph-based algorithms is critically dependent on managing and minimizing graph size.



DFSA D created by application of the NDFSA-to-DFSA conversion algorithm.



Simplified DFSA D' created by combining the four final states in D. The state names have been replaced by names indicating the data each state "remembers".

Automata Configurations

This section applies to DFSA and NDFSA and defines terminology useful for recording the steps taken as an automaton processes input string data. Machine symbols used below are for DFSA.

A ***configuration*** of a finite state machine is defined as the pair $(q, w) \in Q \times \Sigma^*$, where q is the current state and w is the remaining input string to process (from left-to-right). Configurations a

The ***turnstile*** relation ($\vdash$), "yields in one step", is defined as follows:

$(q, w) \vdash (q', w')$ if and only if, for some $\sigma \in \Sigma$, $w = \sigma w'$ and $\delta(q, \sigma) = q'$.

Below is a turnstile operator example using the above DFSA with states Q = {0, B, BA, BAA, TRUE} shown above accepting language L = $\{x \mid x \in \{a,b\} \wedge x$ contains *bab* or *baab*$\}$:

(0, *baab*) $\vdash$ (B, *aab*).

Definition: $\vdash^*$, "yields in zero or more steps", is the transitive closure of $\vdash$.

The $\vdash^*$ operator, Kleene star applied to the turnstile operator, combines a string of zero or more individual turnstile steps into "transitive" relations over multiple steps.

Examples using the same DFSA:

if ((0, *baab*) $\vdash$ (B, *aab*) $\vdash$ (BA, *ab*) $\vdash$ (BAA, *b*)), then (0, *baab*) $\vdash^*$ (BAA, b).

if ((0, *babb*) $\vdash$ (B, *abb*) $\vdash$ (BA, *bb*) $\vdash$ (TRUE, *b*) $\vdash$ (TRUE, ε)), then (0, *babb*) $\vdash^*$ (TRUE, ε).

Definition:

Let D represent a DFSA and L(D) be the language accepted by DFSA D. Then

$$L(D) = \{x: x \in \Sigma^* \text{ and } (\exists q_f \in F)((q_{0s}, x) \vdash^* (q_f, e))\}.$$

Example: *baab* $\in$ L(D) if (0, *baab*) $\vdash^*$(TRUE, ε) , 0 is an initial state, and TRUE is a final state.

Deterministic Finite State MACHINES (Transducers)[2]

Recall that the sole output of an automaton is whether the input sequence is good (accepted) or bad (not accepted). A DFSA is effectively a computational abstraction of a sequential logic circuit that outputs good (1) or bad (0). A deterministic finite state machine (DFSM) or transducer is a DFSA that generates output "actions" associated with each state. The DFSM is analogous to the feedback control system, where the sequential input characters represent either open loop (user) commands or closed loop feedback (observations). The DFSM reacts to this input by updating (transitioning between) internal states plus updating (outputting) a command each time an input character is received/parsed.

For control applications, two types of DFSMs have been formulated: the Moore Machine and the Mealy Machine. In the ***Moore Machine***, output depends only on the current DFSM state. This is the simplest possible DFSM formulation. Consider a simple "elevator door" example. This state

[2] Substantial content from this section is taken from: http://en.wikipedia.org/wiki/Finite-state_machine

machine recognizes two user inputs: "open door" (O) and "close door" (C) which trigger corresponding state transitions. Upon entering state "DOOR OPEN" the system starts a motor to open the door, while entering state "DOOR CLOSED" causes the system to drive the motor to close the door. Timings for door opening and closing actions are not modeled (timed automata will be discussed below and in [Alur and Dill] as a further finite state automaton augmentation). The Fig. 1 two-state model presumes that each "action" will be self-contained, e.g., that a "closing" action will only drive the door motor until the door is actually closed.

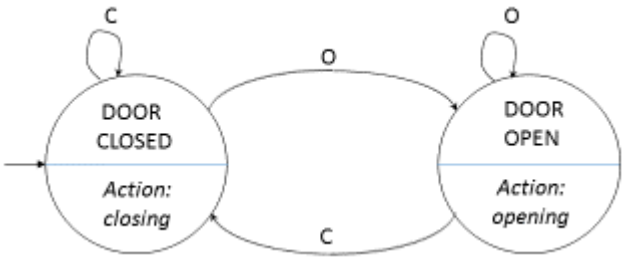


Figure 1: Two-state Simplified Elevator Moore Machine.

In the case where distinction needs to be made between the states in which an elevator is open or closed versus in transition (door opening or closing), a four-state machine is required, as shown in Fig. 2. In this case, user button-pushing (UO, UC) and input from sensors that detect whether a door is fully-open or fully-closed (SO, SC respectively) are required. In the case where no motor action is required, we define a "NOOP" (no operation) action. The NOOP action is generically used to define a system that is not actively operating/acting on itself or its environment. Note that any transition not shown in this chapter implies transition to an absorbing "failure" or "error" state (e.g., a sensor indicating the door has just closed (SC) while executing the "opening" action).

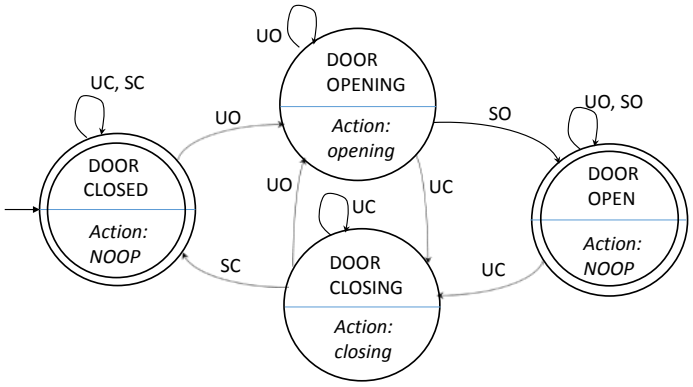


Figure 2: Four-state Elevator Moore Machine.

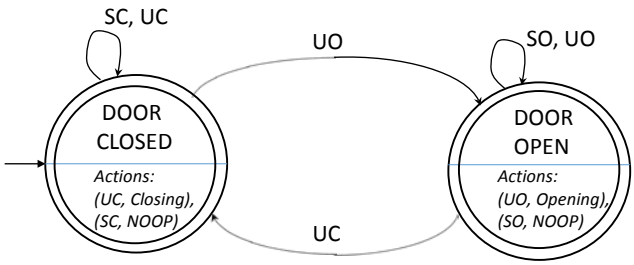


Figure 3: Two-state Elevator Mealy Machine.

A ***Mealy Machine*** generates control outputs that depend on the last input character and current state. The use of a Mealy FSM may lead to a reduction of the number of states (but not always). The elevator example with sensor distinguishing action completion can be reduced to a two-state Mealy Machine. As shown in Figure 3 above, the action executed in each state depends on which input was last received as well as the current state, allowing "NOOP" to be recognized as appropriate if the sensor indicates completion of a door opening or closing action.

A *deterministic finite state machine (DFSM) or transducer* is described by $(\Sigma, \Gamma, S, s_0, \delta, \omega)$, where:

- Σ is the input alphabet (a finite non-empty set of symbols).
- Γ is the output alphabet (a finite, non-empty set of symbols).
- S is a finite, non-empty set of states.
- s_0 is the initial state, an element of S.
- δ is the state-transition function: $\delta : S \times \Sigma \rightarrow S$.
- ω is the output function.

If the output function is a function of a state and input alphabet ($\omega : S \times \Sigma \rightarrow \Gamma$) that definition corresponds to a Mealy machine. If the output function depends only on a state ($\omega : S \rightarrow \Gamma$) that definition corresponds to a Moore machine. If we disregard the first output symbol of a Moore machine, $\omega(s_0)$, then it can be readily converted to an output-equivalent Mealy machine by setting the output function of every Mealy transition (i.e. labeling every edge) with the output symbol given of the destination Moore state. The converse transformation is less straightforward because a Mealy machine state may have different output labels on its incoming transitions (edges). Every such state needs to be split in multiple Moore machine states, one for every incident output symbol.

Deterministic Finite State Automata / Machine Optimization

Optimizing a DFSA/DFSM means finding the automaton or machine with the minimum number of states that accepts the desired language (performs the desired function). The fastest known algorithm

for doing this is the Hopcroft minimization algorithm that merges "non-distinguishable" states. Note that the following text and algorithm pseudocode are from http://en.wikipedia.org/wiki/DFA_minimization#Hopcroft.27s_algorithm.

The Hopcraft algorithm is based on partitioning the DFA states into groups by their behavior. These groups represent equivalence classes of the Myhill–Nerode equivalence relation, whereby every two states of the same partition are equivalent if they have the same behavior for all the input sequences. That is, for every two states p_1 and p_2 that belong to the same equivalence class within the partition P, it will be the case that for every input word w, if one follows the transitions determined by w from the two states p_1 and p_2 one will either be led to accepting states in both cases or be led to rejecting states in both cases; it should not be possible for w to take p_1 to an accepting state and p_2 to a rejecting state or vice versa. The following pseudocode describes the algorithm:

Algorithm 1: Hopcraft Minimization Algorithm

```
P := {F, Q \ F};  // F = set of final states, Q = set of all states.
W := {F};
while (W is not empty) do
    choose and remove a set A from W
    for each c in ∑ do
        let X be the set of states for which a transition on c leads to a state in A
        for each set Y in P for which X ∩ Y is nonempty and Y \ X is nonempty do
            replace Y in P by the two sets X ∩ Y and Y \ X
            if Y is in W
                replace Y in W by the same two sets
            else
                if |X ∩ Y| <= |Y \ X|
                    add X ∩ Y to W
                else
                    add Y \ X to W
        end;
    end;
end;
```

The algorithm starts with a partition that is too coarse: every pair of states that are equivalent according to the Myhill–Nerode relation belong to the same set in the partition, but pairs that are inequivalent might also belong to the same set. It gradually refines the partition into a larger number of smaller sets, at each step splitting sets of states into pairs of subsets that are necessarily inequivalent. The initial partition is a separation of the states into two subsets of states that clearly do not have the same behavior as each other: the accepting states and the rejecting states. The algorithm then repeatedly chooses a set A from the current partition and an input symbol c, and splits each of

the sets of the partition into two (possibly empty) subsets: the subset of states that lead to A on input symbol c, and the subset of states that do not lead to A. Since A is already known to have different behavior than the other sets of the partition, the subsets that lead to A also have different behavior than the subsets that do not lead to A. When no more splits of this type can be found, the algorithm terminates. The above Wikipedia page also contains algorithms to define unreachable and nondistinguishable states:

The state p of DFSA $M=(Q, \Sigma, \delta, q_0, F)$ is unreachable if no such string w in $\sum^*$ exists for which $p=\delta(q_0, w)$. Reachable states can be obtained with the algorithm shown below. Unreachable states can be removed from the DFSA without affecting the language that it accepts.

Algorithm 2: Algorithm to identify the Reachable State Set of a DFSA

```
let reachable_states:= {q0};
let new_states:= {q0};
do {
   temp := the empty set;
   for each q in new_states do
      for all c in ∑ do
         temp := temp ∪ {p such that p=δ(q,c)};
      end;
   end;
   new_states := temp \ reachable_states;
   reachable_states := reachable_states ∪ new_states;
} while(new_states ≠ the empty set);
unreachable_states := Q \ reachable_states;
```

The above algorithms collectively can be used to ensure a DFSA has the minimal number of states and that all states are actually reachable from the single initial state. Note that the DFSA or DFSM originally input to these algorithms must be correct, i.e., accept the desired language (DFSA) or implement the desired function (DFSM), for the optimized DFSA/DFSM to be correct.