Deterministic Finite Automata

Definition: A deterministic finite automaton (DFA) consists of

1. a finite set of *states* (often denoted Q)
2. a finite set Σ of *symbols* (alphabet)
3. a *transition function* that takes as argument a state and a symbol and returns a state (often denoted δ)
4. a *start state* often denoted q_0
5. a set of *final* or *accepting* states (often denoted F)

We have $q_0 \in Q$ and $F \subseteq Q$

Deterministic Finite Automata

So a DFA is mathematically represented as a 5-uple

$(Q, \Sigma, \delta, q_0, F)$

The transition function δ is a function in

$Q \times \Sigma \to Q$

$Q \times \Sigma$ is the set of 2-tuples (q, a) with $q \in Q$ and $a \in \Sigma$

Deterministic Finite Automata

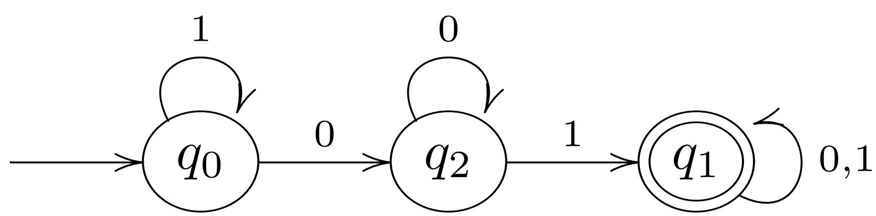
How to present a DFA? With a *transition table*

	0	1
$\rightarrow q_0$	q_2	q_0
$*q_1$	q_1	q_1
q_2	q_2	q_1

The $\rightarrow$ indicates the *start* state: here q_0

The $*$ indicates the final state(s) (here only one final state q_1)

This defines the following *transition diagram*



Deterministic Finite Automata

For this example

$Q = \{q_0, q_1, q_2\}$

start state q_0

$F = \{q_1\}$

$\Sigma = \{0, 1\}$

δ is a *function* from $Q \times \Sigma$ to Q

$\delta : Q \times \Sigma \rightarrow Q$

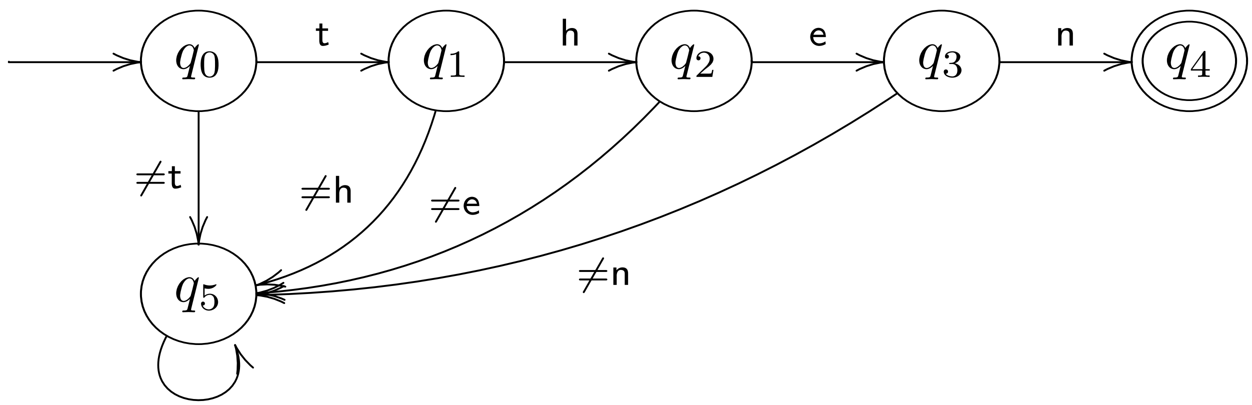
$\delta(q_0, 1) = q_0$

$\delta(q_0, 0) = q_2$

Example: password

When does the automaton accepts a word??

It reads the word and accepts it if it stops in an accepting state



Only the word then is accepted

Here $Q = \{q_0, q_1, q_2, q_3, q_4\}$

Σ is the set of all characters

$F = \{q_4\}$

We have a "stop" or "dead" state q_5, *not* accepting

How a DFA Processes Strings

Let us build an automaton that accepts the words that contain 01 as a subword

$\Sigma = \{0, 1\}$

$L = \{x01y \mid x, y \in \Sigma^*\}$

We use the following states

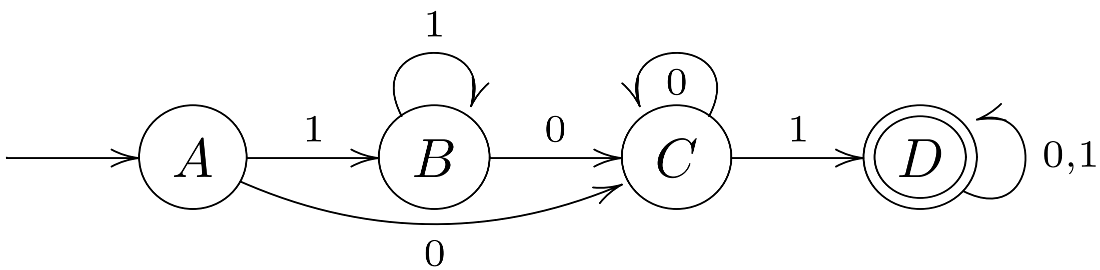
A: start

B: the most recent input was 1 (but not 01 yet)

C: the most recent input was 0 (so if we get a 1 next we should go to the accepting state D)

D: we have encountered 01 (accepting state)

We get the following automaton



Transition table

	0	1
→A	C	B
B	C	B
C	C	D
∗D	D	D

Q = {A,B,C,D}, Σ = {0,1}, start state A, final state(s) {D}

Extending the Transition Function to Strings

In the previous example, what happens if we get 011? 100? 10101?

We define $\hat{\delta}(q, x)$ by induction

$\hat{\delta} : Q \times \Sigma^* \to Q$

BASIS $\hat{\delta}(q, \epsilon) = q$ for $|x| = 0$

INDUCTION suppose $x = ay$ (y is a string, a is a symbol)

$\hat{\delta}(q, ay) = \hat{\delta}(\delta(q, a), y)$

Notice that if $x = a$ we have

$\hat{\delta}(q, a) = \delta(q, a)$ since $a = a\epsilon$ and $\hat{\delta}(\delta(q, a), \epsilon) = \delta(q, a)$

Extending the Transition Function to Strings

$\hat{\delta} : Q \times \Sigma^* \to Q$

We write $q.x$ instead of $\hat{\delta}(q, x)$

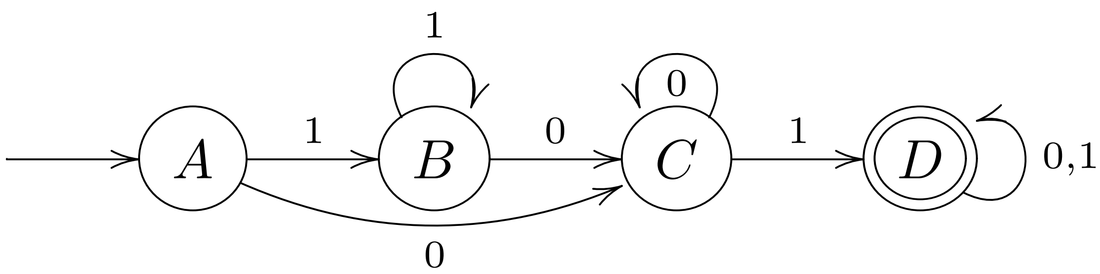
We can now define mathematically the *language* accepted by a given automaton $Q, \Sigma, \delta, q_0, F$

$L = \{x \in \Sigma^* \mid q_0.x \in F\}$

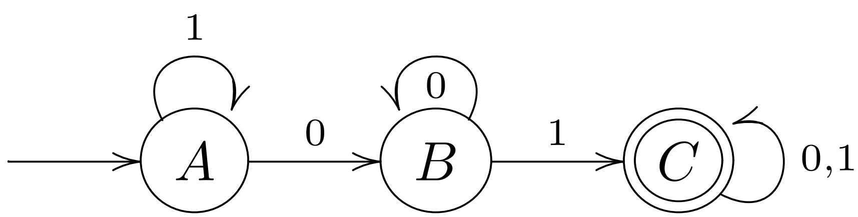
On the previous example 100 is not accepted and 10101 is accepted

Minimalisation

The same language may be represented by different DFA



and



Minimalisation

Later in the course we shall show that there is only *one* machine with the minimum number of states (up to renaming of states)

Furthermore, there is a (clever) algorithm which can find this minimal automaton given an automaton for a language

Example

M_n the "cyclic" automaton with n states on $\Sigma = \{1\}$ such that

$$L(M_n) = \{1^l \mid n \; divides \; l\}$$

Functional representation: Version 1

$Q = A|B|C$ and $E = 0|1$ and $W = [E]$

One function $next : Q \times E \to Q$

$next\ (A, 1) = A,\ next\ (A, 0) = B$

$next\ (B, 1) = C,\ next\ (B, 0) = B$

$next\ (C, b) = C$

One function $run : Q \times W \to Q$

$run\ (q, b : x) = run\ (next\ (q, b), x), \quad run\ (q, []) = q$

$accept\ x = final\ (run\ (A, x))$ where

$final\ A = final\ B = False, \ \ final\ C = True$

Functional representation: Version 2

$E = 0|1, \qquad W = [E]$

Three functions $F_A, F_B, F_C : W \rightarrow Bool$

$F_A\ (1 : x) = F_A\ x, \qquad F_A\ (0 : x) = F_B\ x, \qquad F_A\ [] = False$

$F_B\ (1 : x) = F_C\ x, \qquad F_B\ (0 : x) = F_B\ x, \qquad F_B\ [] = False$

$F_C\ (1 : x) = F_C\ x, \qquad F_C\ (0 : x) = F_C\ x, \qquad F_C\ [] = True$

We have a *mutual recursive definition* of 3 functions

Functional representation: Version 3

```
data Q = A | B | C
data E = O | I

next :: Q -> E -> Q
next A I = A
next A O = B
next B I = C
next B O = B
next C _ = C

run :: Q -> [E] -> Q
run q (b:x) = run (next q b) x
run q [] = q
```

Functional representation: Version 3

```
accept :: [E] -> Bool
accept x = final (run A x)

final :: Q -> Bool
final A = False
final B = False
final C = True
```

Functional representation: Version 4

We have

```
Q -> E -> Q     ~     Q x E   -> Q

                ~     E -> (Q -> Q)
```

Functional representation: Version 4

```
data Q = A | B | C
data E = O | I

next :: E -> Q -> Q
next I A = A
next O A = B
next I B = C
next O B = B
next _ C = C

run :: Q -> [E] -> Q
run q (b:x) = run (next b q) x
run q [] = q
```

Functional representation: Version 4

```
-- run q [b1,...,bn] is
-- next bn (next b(n-1) (... (next b1 q)...))
-- run = foldl next
```

A proof by induction

A very important result, quite intuitive, is the following.

Theorem: *for any state* q *and any word* x *and* y *we have* $q.(xy) = (q.x).y$

Proof by induction on x. We prove that: *for all* q *we have* $q.(xy) = (q.x).y$ (notice that y is fixed)

Basis: $x = \epsilon$ then $q.(xy) = q.y = (q.x).y$

Induction step: we have $x = az$ and we assume $q'.(zy) = (q'.z).y$ *for all* q'

The other definition of $\hat{\delta}$

Recall that $a(b(cd)) = ((ab)c)d$; we have two descriptions of words

We define $\hat{\delta}'(q, \epsilon) = q$ and

$\hat{\delta}'(q, xa) = \delta(\hat{\delta}'(q, x), a)$

Theorem: *We have* $q.x = \hat{\delta}(q, x) = \hat{\delta}'(q, x)$ *for all* x

The other definition of $\hat{\delta}$

Indeed we have proved

$q.\epsilon = q$ and $q.(xy) = (q.x).y$

As a special case we have $q.(xa) = (q.x).a$

This means that we have two functions $f(x) = q.x$ and $g(x) = \hat{\delta}'(q, x)$ which satisfy

$f(\epsilon) = g(\epsilon) = q$ and

$f(xa) = f(x).a \qquad g(xa) = g(x).a$

Hence $f(x) = g(x)$ for all x that is $q.x = \hat{\delta}'(q, x)$

Automatic Theorem Proving

$$f(0) = h(0) = 0, \quad g(0) = 1$$

$$f(n+1) = g(n),\ g(n+1) = f(n),\ h(n+1) = 1 - h(n)$$

We have $f(n) = h(n)$

We can prove this automatically using DFA

Automatic Theorem Proving

We have 8 states: $Q = \{0,1\} \times \{0,1\} \times \{0,1\}$

We have only one action $\Sigma = \{1\}$ and $\delta((a,b,c),s) = (b,a,1-c)$

The initial state is $(0,1,0) = (f(0),g(0),h(0))$

Then we have $(0,1,0).1^n = (f(n),g(n),h(n))$

We check that all accessible states satisfy $a = c$ (that is, the property $a = c$ is an invariant for each transition of the automata)

Automatic Theorem Proving

A more complex example

$f(0) = 0 \qquad f(1) = 1 \qquad f(n+2) = f(n) + f(n+1) - f(n)f(n+1)$

$f(2) = 1 \qquad f(3) = 0 \qquad f(4) = 1 \qquad f(5) = 1 \qquad \ldots$

Show that $f(n+3) = f(n)$ by using $Q = \{0,1\} \times \{0,1\} \times \{0,1\}$ and the transition function $(a, b, c) \longmapsto (b, c, b + c - bc)$ with the initial state $(0, 1, 1)$

Product of automata

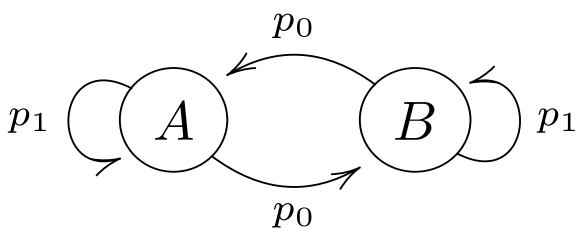
How do we represent *interaction* between machines?

This is via the *product* operation

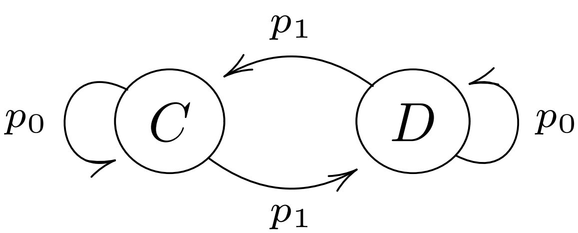
There are different kind of products

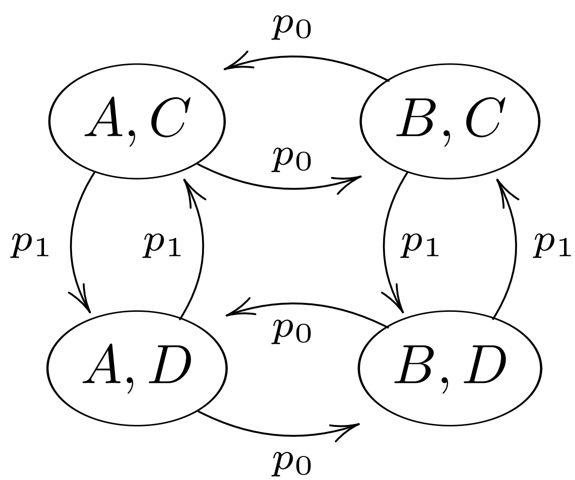
We may then have *combinatorial explosion*: the product of n automata with 2 states has 2^n states!

Product of automata (example)

The product of 

and



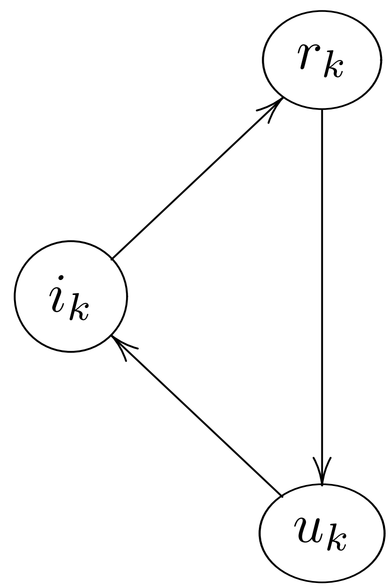
is 

If we start from A, C and after the word w we are in the state A,D we know that w contains an even number of p_0s and odd number of p_1s

Product of automata (example)

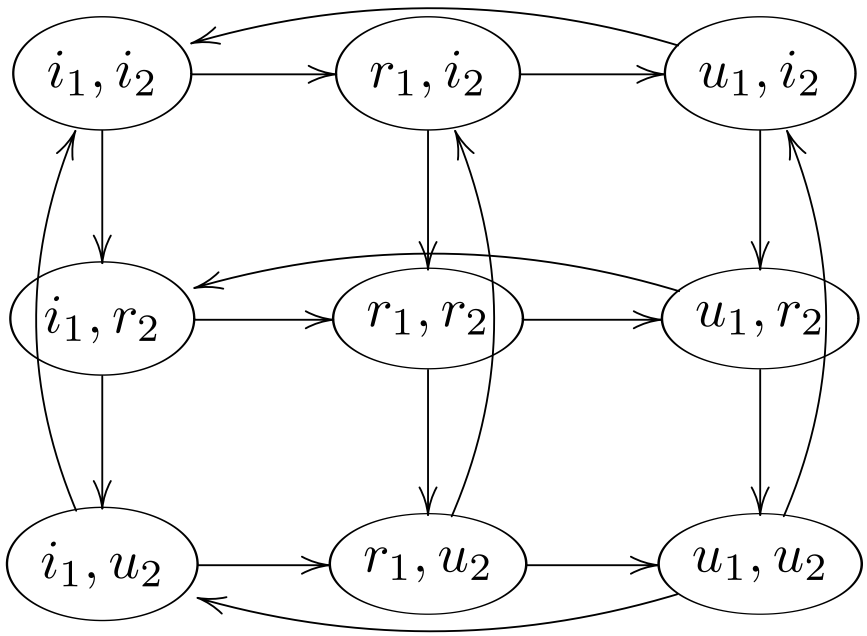
Model of a system of *users* that have three states I(dle), R(equesting) and U(sing). We have two users for $k = 1$ or $k = 2$

Each user is represented by a simple automaton



Product of automata (example)

The complete system is represented by the product of these two automata; it has $3 \times 3 = 9$ states



The Product Construction

Given $A_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ two DFAs *with the same alphabet* Σ we can define the product $A = A_1 \times A_2$

set of state $Q = Q_1 \times Q_2$

transition function $(r_1, r_2).a = (r_1.a, r_2.a)$

intial state $q_0 = (q_1, q_2)$

accepting states $F = F_1 \times F_2$

The Product Construction

Lemma: $(r_1, r_2).x = (r_1.x, r_2.x)$

We prove this by induction

BASE: the statement holds for $x = \epsilon$

STEP: if the statement holds for y it holds for $x = ya$

The Product Construction

Theorem: $L(A_1 \times A_2) = L(A_1) \cap L(A_2)$

Proof: We have $(q_1, q_2).x = (q_1.x, q_2.x)$ in F iff $q_1.x \in F_1$ and $q_2.x \in F_2$, that is $x \in L(A_1)$ and $x \in L(A_2)$

Example: let M_k be the "cyclic" automaton that recognizes multiple of k, such that $L(M_k) = \{a^n \mid k \textit{ divides } n\}$, then $M_6 \times M_9 \simeq M_{18}$

Notice that 6 divides k and 9 divides k iff 18 divides k

Product of automata

It can be quite difficult to build automata directly for the intersection of two regular languages

Example: build a DFA for the language that contains the subword ab twice and an even number of a's

Variation on the product

We define $A_1 \oplus A_2$ as $A_1 \times A_2$ but we change the notion of accepting state

(r_1, r_2) accepting iff $r_1 \in F_1$ *or* $r_2 \in F_2$

Theorem: *If A_1 and A_2 are DFAs, then* $L(A_1 \oplus A_2) = L(A_1) \cup L(A_2)$

Example: multiples of 3 or of 5 by taking $M_3 \oplus M_5$

Complement

If $A = (Q, \Sigma, \delta, q_0, F)$ we define the *complement* $\bar{A}$ of A as the automaton

$\bar{A} = (Q, \Sigma, \delta, q_0, Q - F)$

Theorem: *If A is a DFA, then* $L(\bar{A}) = \Sigma^* - L(A)$

Remark: We have $A \oplus A' = \overline{\overline{A} \times \overline{A'}}$

Languages

Given an alphabet Σ

A *language* is simply a *subset* of Σ^*

Common languages, programming languages, can be seen as sets of words

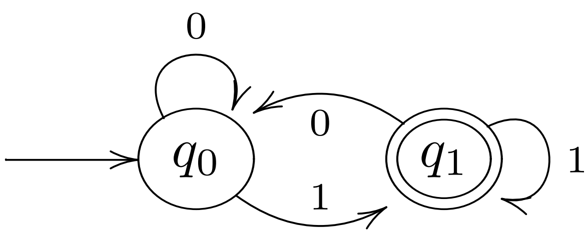
Definition: *A language $L \subseteq \Sigma^*$ is* regular *iff there exists a DFA A, on the same alphabet Σ such that $L = L(A)$*

Theorem: *If L_1, L_2 are regular then so are* $L_1 \cap L_2,\ L_1 \cup L_2,\ \Sigma^* - L_1$

Remark: Accessible Part of a DFA

Consider the following DFA

it is clear that it accepts the same language as the DFA



which is the *accessible part* of the DFA

The remaining states are not accessible from the start state and can be removed

Remark: Accessible Part of a DFA

The set

$Acc = \{q_0.x \mid x \in \Sigma^*\}$

is the set of *accessible* states of the DFA (states that are accessible from the state q_0)

Remark: Accessible Part of a DFA

Proposition: *If* $A = (Q, \Sigma, \delta, q_0, F)$ *is a DFA then and* $A' = (Q \cap Acc, \Sigma, \delta, q_0, F \cap Acc)$ *is a DFA such that* $L(A) = L(A')$.

Proof: It is clear that A' is well defined and that $L(A') \subseteq L(A)$.

If $x \in L(A)$ then we have $q_0.x \in F$ and also $q_0.x \in Acc$. Hence $q_0.x \in F \cap Acc$ and $x \in L(A')$.

Automatic Theorem Proving

Take $\Sigma = \{a, b\}$.

Define L set of $x \in \Sigma^*$ such that any a in x is followed by a b

Define L' set of $x \in \Sigma^*$ such that any b in x is followed by a a

Then $L \cap L' = \{\epsilon\}$

Intuitively if $x \neq \epsilon$ in L we have

$\dots a \dots \rightarrow \dots a \dots b \dots$

if x in L' we have

$\dots b \dots \rightarrow \dots b \dots a \dots$

Automatic Theorem Proving

We should have $L \cap L' = \{\epsilon\}$ since a nonempty word in $L \cap L'$ should be infinite

We can prove this automatically with automata!

L is regular: write a DFA A for L

L' is regular: write a DFA A' for L

We can then compute $A \times A'$ and check that

$$L \cap L' = L(A \times A') = \{\epsilon\}$$

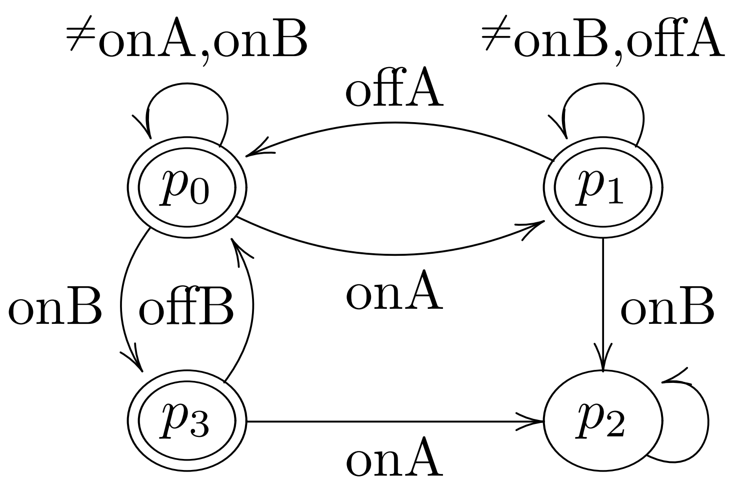
Application: control system

We have several machines working concurrently

We need to forbid some sequence of actions. For instance, if we have two machines MA and MB, we may want to say that MB cannot be on when MA is on. The alphabets will contain: onA, offA, onB, offB

Between onA, on2 there should be at least one offA

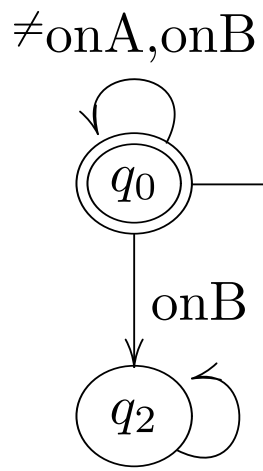
The automaton expressing this condition is



Application: control system

What is interesting is that we can use the product construction to combine several conditions

For instance, another condition maybe that onA should appear before onB appear. One automaton representing this condition is



We can take the product of the two automata to express the two conditions as one automaton, which may represent the control system