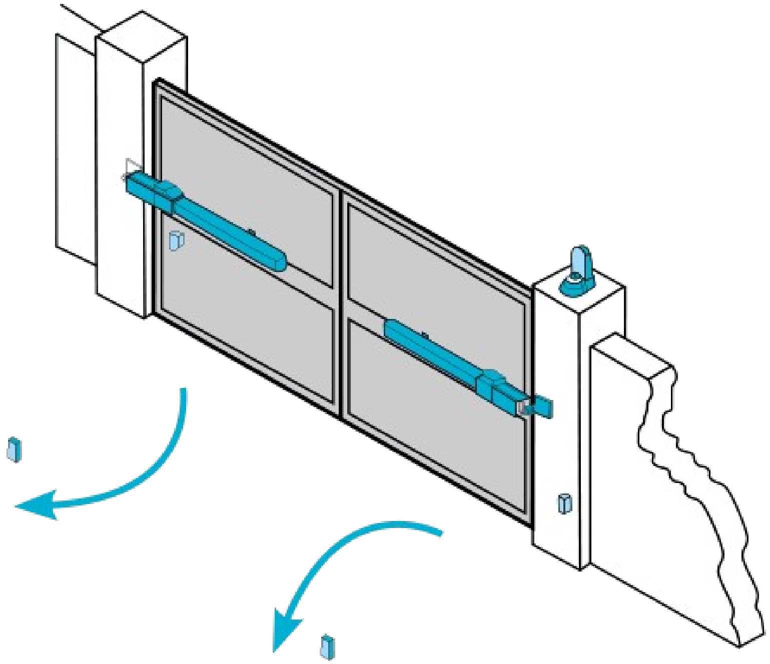
SCXML
State Chart XML

A superset of different dialects

Running Example

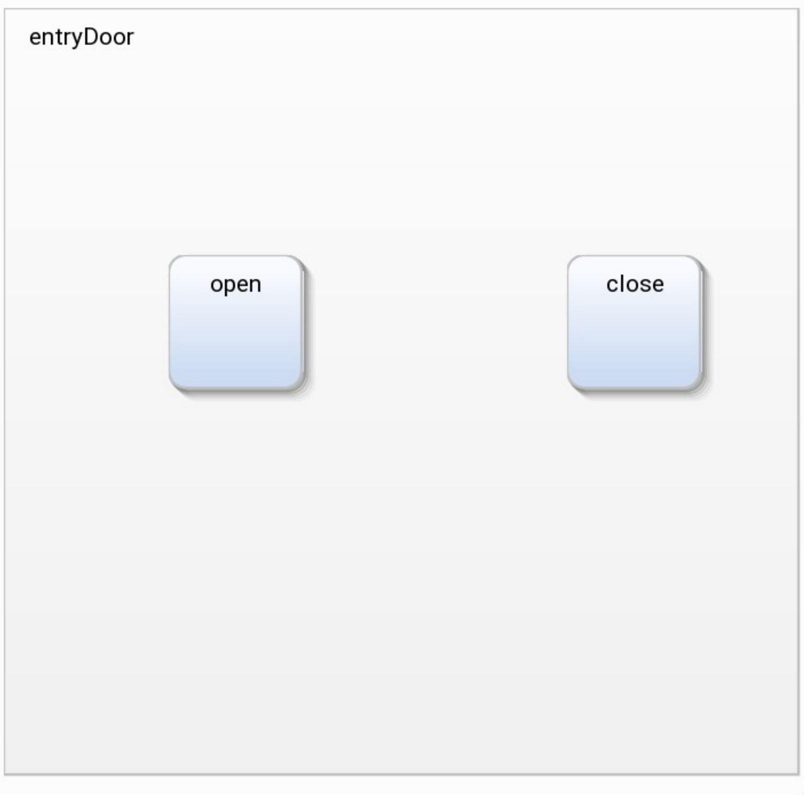
- We want to model the controller of an entry door by using a **FSM**.



Running Example

- We want to model the controller of an entry door by using a **FSM**.

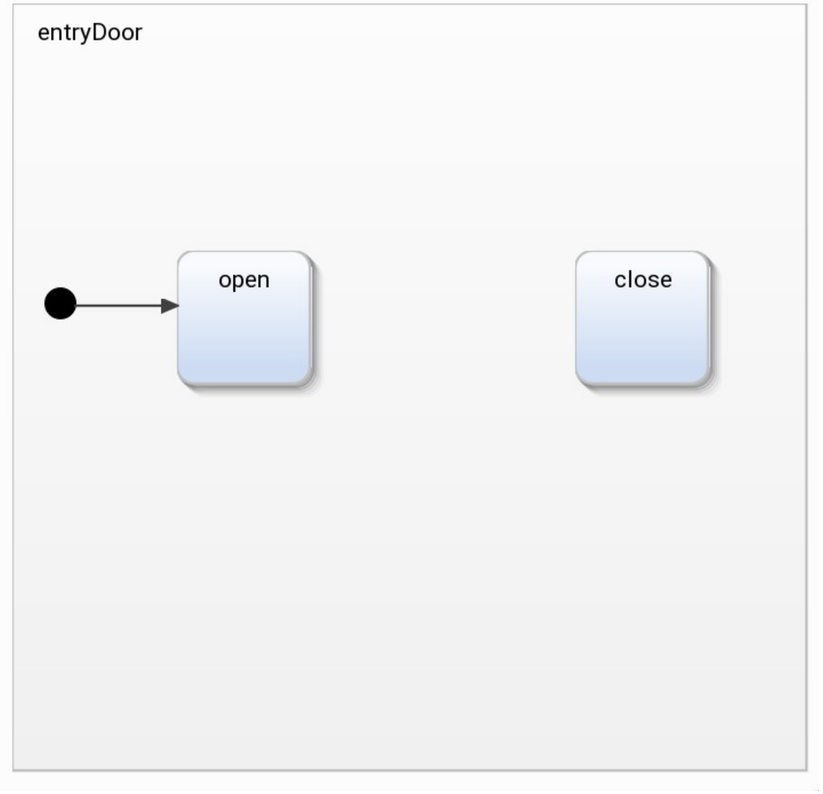
Q is a set of State



A **finite state transducer** is defined by < δ >

Running Example

- We want to model the controller of an entry door by using a FSM.



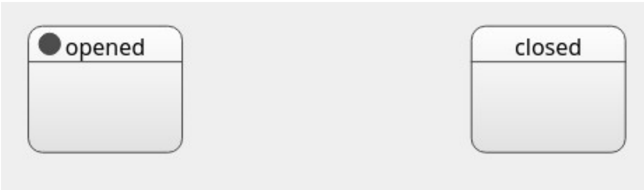
Q is a set of State

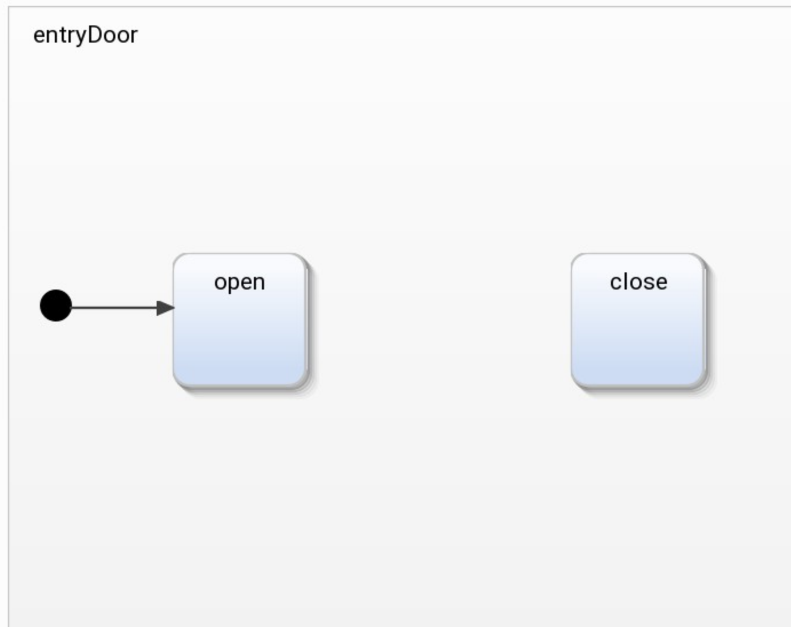
$q_0 \in Q$ is the initial state

A finite state transducer is defined by <Q , q_0 >

Running Example

- We want to model the controller of an entry door by using a FSM.





Q is a set of State

$q_0 \in Q$ is the initial state

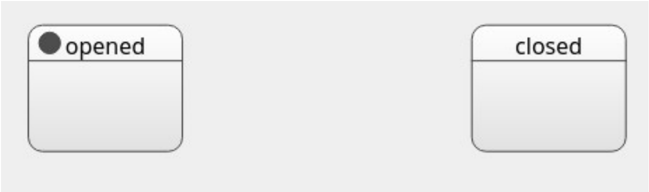
- The only difference between the <initial> element and the 'initial' attribute is that the <initial> element contains a <transition> element which may in turn contain executable content which will be executed before the default state is entered. If the 'initial' attribute is specified instead, the specified state will be entered, but no executable content will be executed.
- (If neither the <initial> child or the 'initial' element is specified, **the default initial state is the first child state in document order**

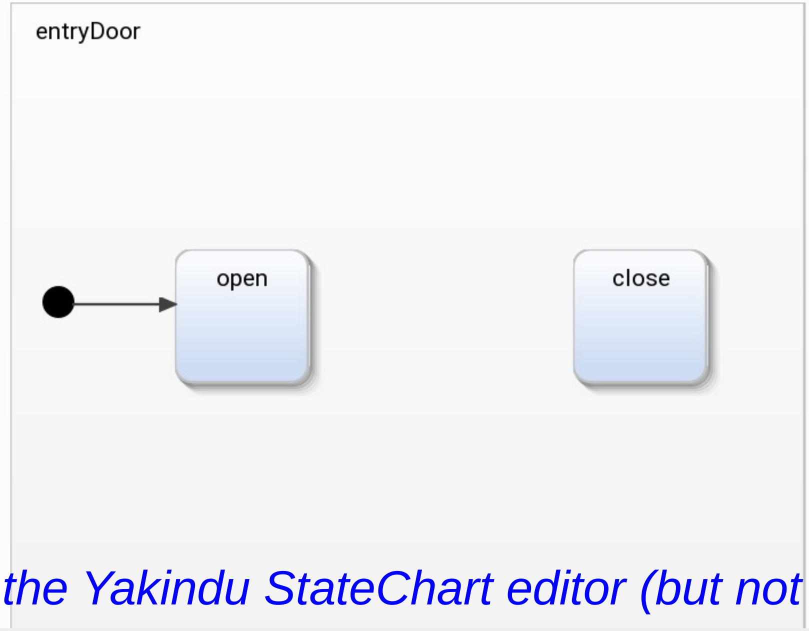
Taken from the official standard: https://www.w3.org/TR/scxml/

A finite state transducer is defined by <Q , q_0 , >

Running Example

- We want to model the controller of an entry door by using a FSM.





Q is a set of State

$q_0 \in Q$ is the initial state

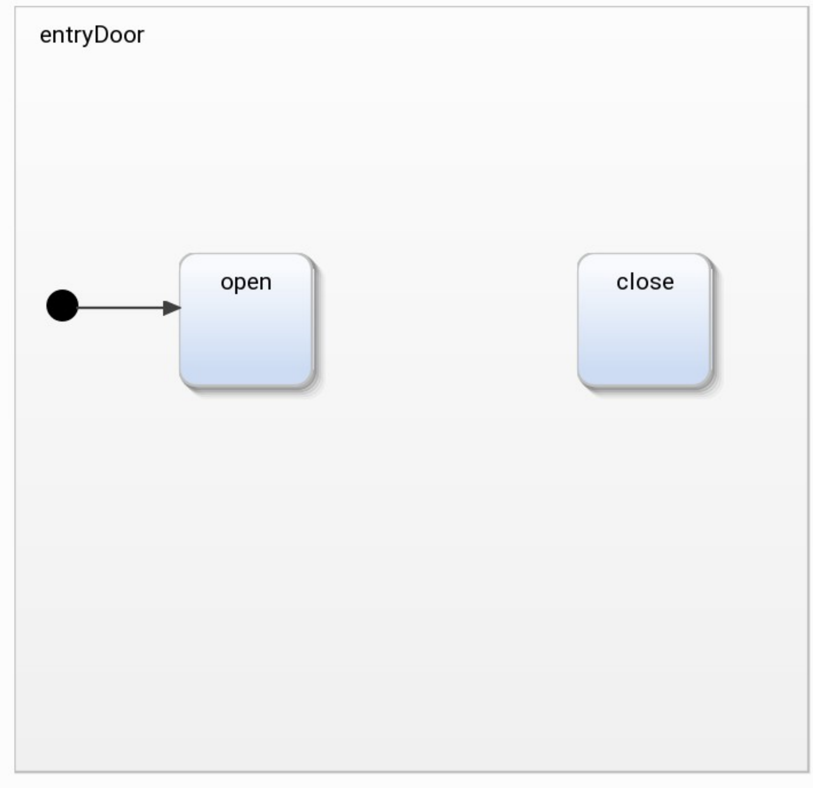
supported in the Yakindu StateChart editor (but not in many other tools)

- The only difference between the <initial> element and the 'initial' attribute is that the <initial> element contains a <transition> element which *may in turn contain executable content which will be executed before the default state is entered*. If the 'initial' attribute is specified instead, the specified state will be entered, but no executable content will be executed.
- (If neither the <initial> child or the 'initial' element is specified, **the default initial state is the first child state in document order**

A finite state transducer is defined by <Q , q_0 , >

Running Example

- We want to model the controller of an entry door by using a FSM.



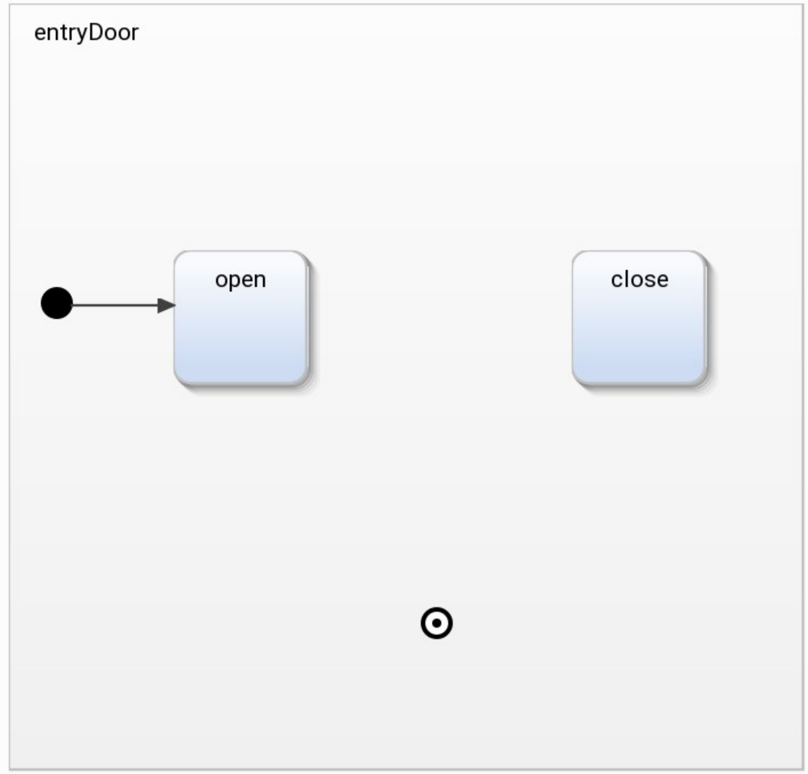
Q is a set of State

$q_0 \in Q$ is the initial state

A finite state transducer is defined by <Q , q_0 >

Running Example

- We want to model the controller of an entry door by using a FSM.



$\mathcal{Q}$ is a set of State

$q_0 \in \mathcal{Q}$ is the initial state

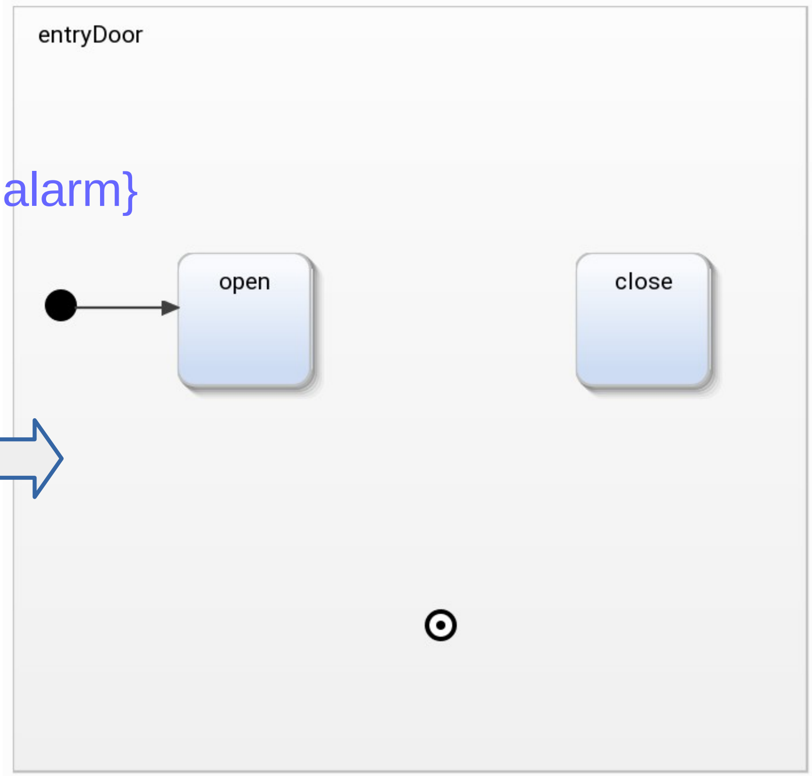
$\mathcal{F}$ is the set of final states

A finite state transducer is defined by $< \mathcal{Q}, q_0, \mathcal{F}, \quad >$

Running Example

- We want to model the controller of an entry door by using a FSM.

Σ_I = {leftB, rightB, alarm}



Q is a set of State

$q_0 \in Q$ is the initial state

$\mathcal{F}$ is the set of final states

Σ_I is the input alphabet

A finite state transducer is defined by $< Q, q_0, \mathcal{F}, \Sigma_I,$ $>$

Running Example

- We want to model the controller of an entry door by using a FSM.

Σ_I = {leftB, rightB, alarm}

Σ_O = {doOpen, doClose}

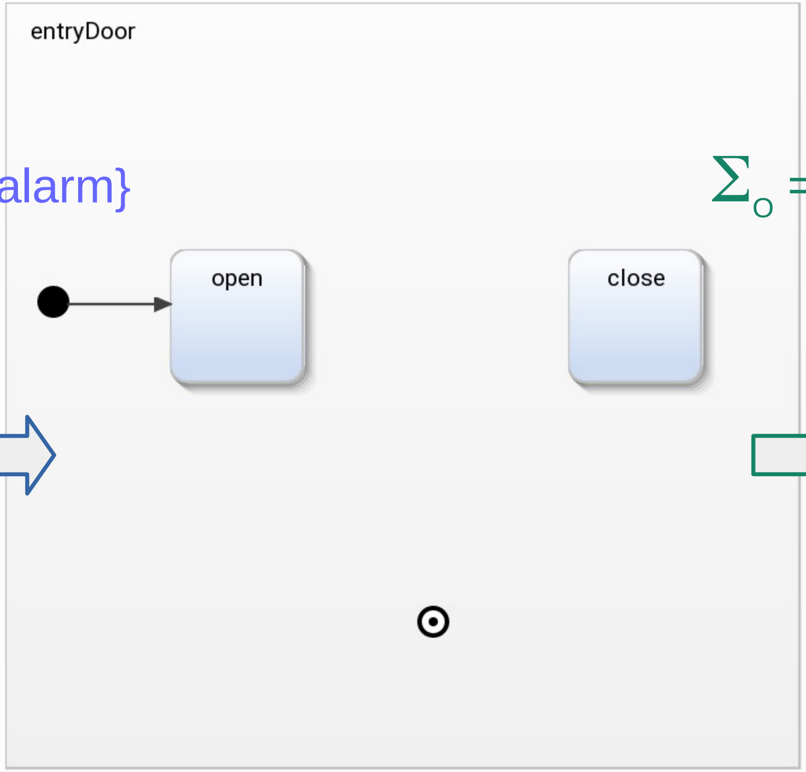
Q is a set of State

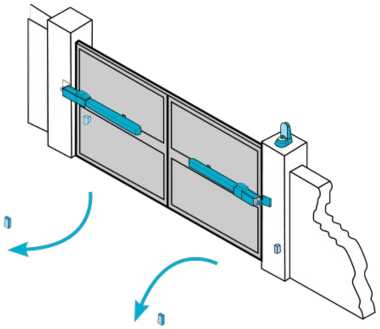
$q_0 \in Q$ is the initial state

$\mathcal{F}$ is the set of final states

Σ_I is the input alphabet

Σ_O is the input alphabet





A finite state transducer is defined by <Q , q_0, $\mathcal{F}$, Σ_I, Σ_O >

Running Example

- We want to model the controller of an entry door by using a FSM.

Σ_I = {leftB, rightB, alarm}

Σ_O = {doOpen, doClose}

$\mathcal{Q}$ is a set of State

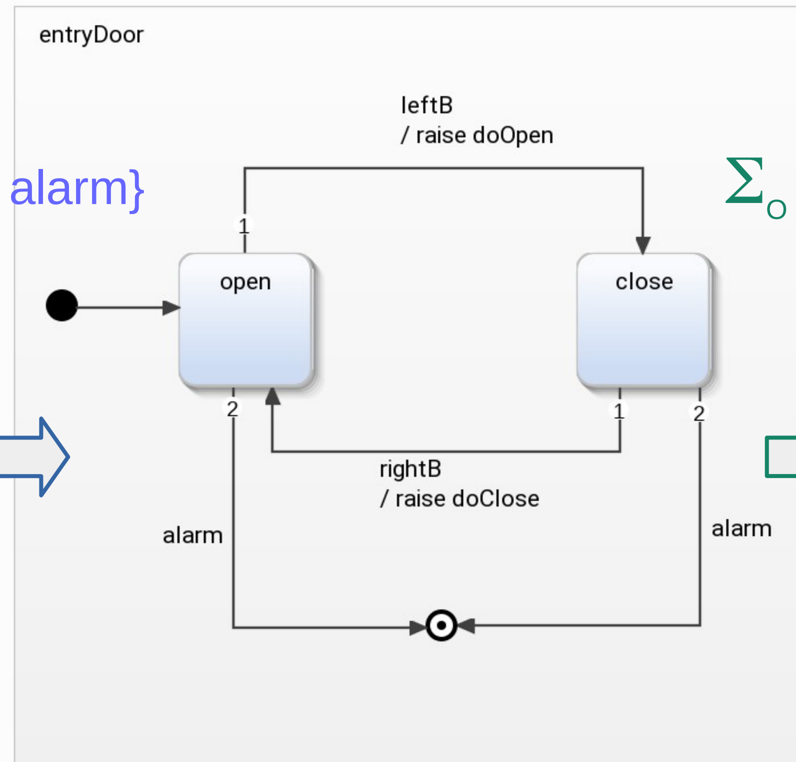
$q_0 \in \mathcal{Q}$ is the initial state

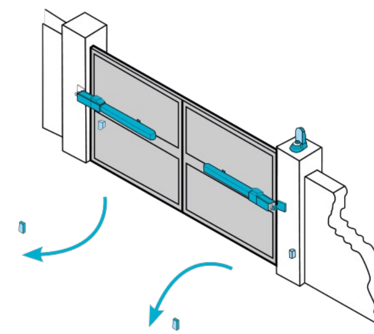
$\mathcal{F}$ is the set of final states

Σ_I is the input alphabet

Σ_O is the input alphabet

$\delta \subseteq \mathcal{Q} \times \Sigma_I \times \Sigma_O \times \mathcal{Q}$





$\delta = \{$<opened, leftB, doClose, closed>,
<closed, rightB, doOpen, opened>,
<opened, alarm, ?, final>,
<closed, alarm, ?, final>$\}$

A finite state transducer is defined by $< \mathcal{Q}, q_0, \mathcal{F}, \Sigma_I, \Sigma_O, \delta >$

Running Example

- We want to model the controller of an entry door by using a FSM.

Σ_I = {leftB, rightB, alarm}

Σ_O = {doOpen, doClose}

$\mathcal{Q}$ is a set of State

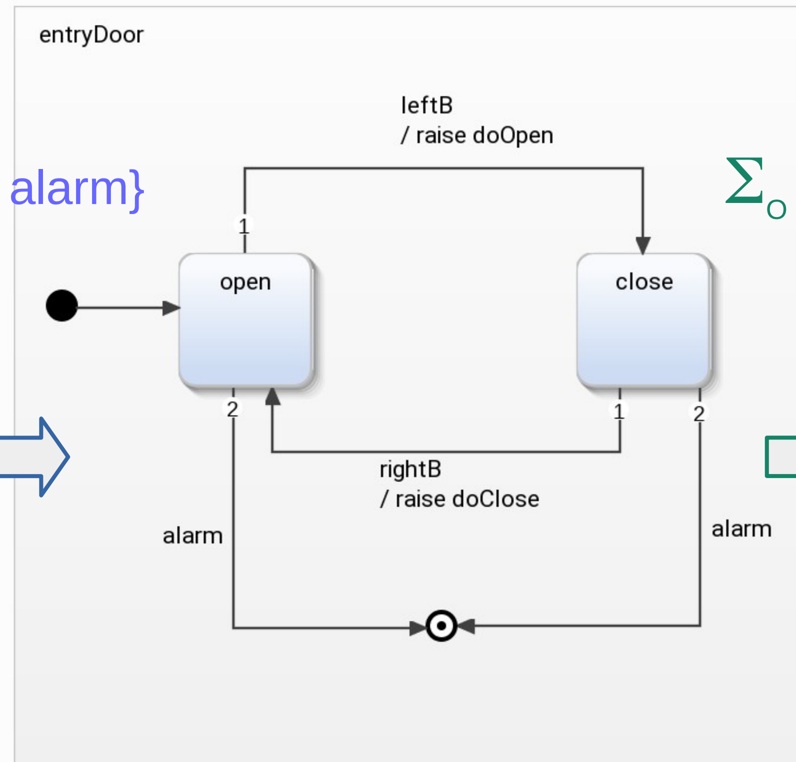
$q_0 \in \mathcal{Q}$ is the initial state

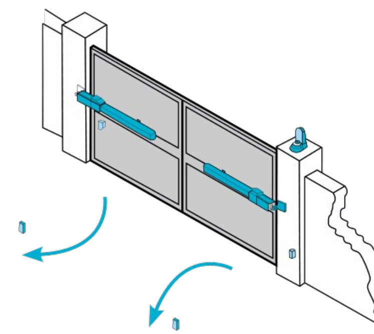
$\mathcal{F}$ is the set of final states

Σ_I is the input alphabet

Σ_O is the input alphabet

$\delta \subseteq \mathcal{Q} \times \Sigma_I \times \Sigma_O \times \mathcal{Q}$





$\delta = \{$<opened, leftB, doClose, closed>,
<closed, rightB, doOpen, opened>,
<opened, alarm, ?, final>,
<closed, alarm, ?, final>$\}$

A finite state transducer is defined by <$\mathcal{Q}$, q_0, $\mathcal{F}$, Σ_I, Σ_O, δ >

Running Example

- We want to model the controller of an entry door by using a FSM.

Σ_I = {leftB, rightB, alarm}

Σ_O = {doOpen, doClose}

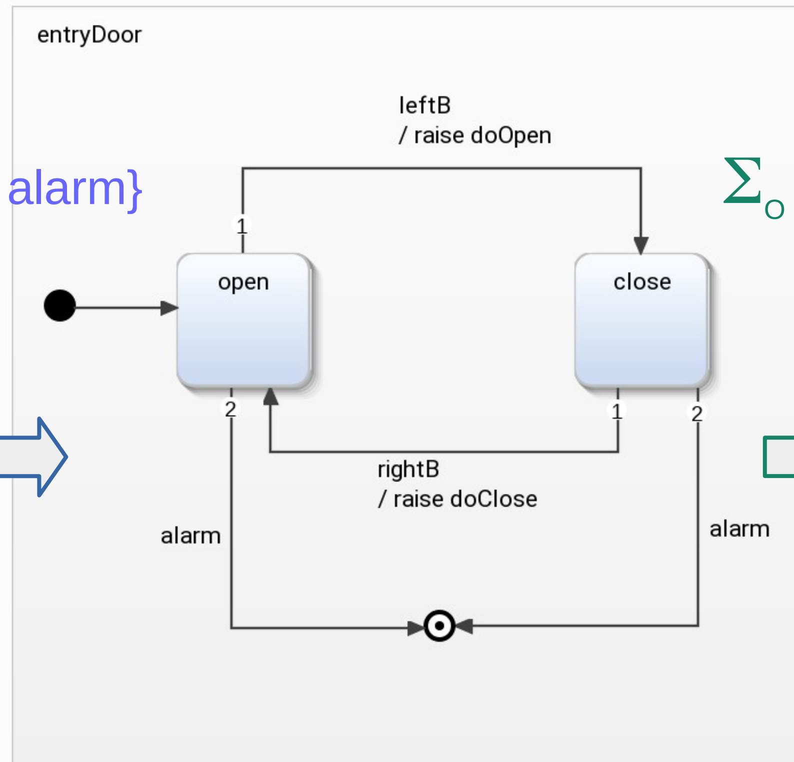
Q is a set of State

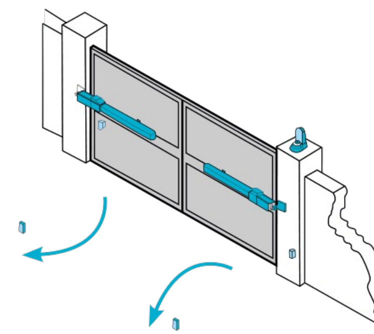
$q_0 \in Q$ is the initial state

$\mathcal{F}$ is the set of final states

Σ_I is the input alphabet

Σ_O is the input alphabet





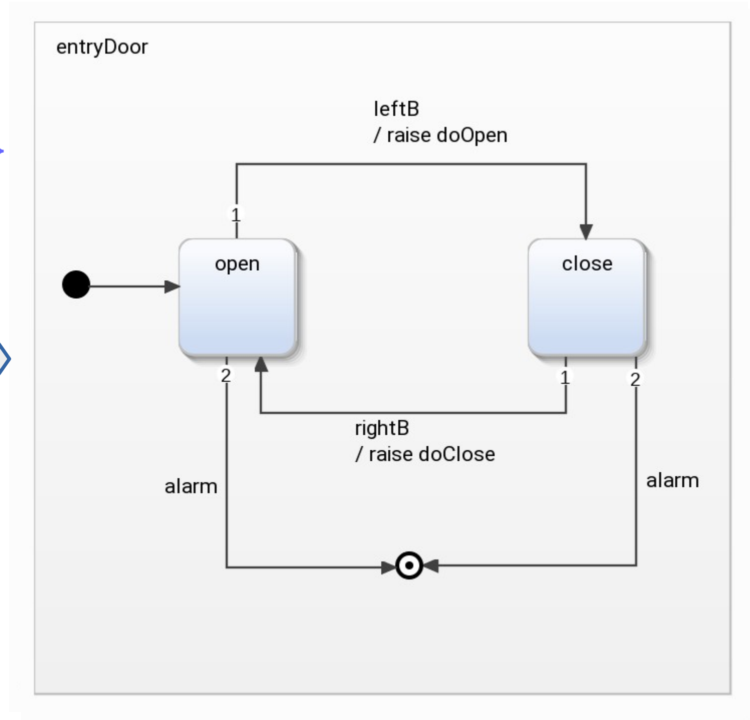
$$\delta \subseteq Q \times (\Sigma_I \cup \{\varepsilon\}) \times (\Sigma_O \cup \{\varepsilon\}) \times Q$$

$\delta = \{$<opened, leftB, doClose, closed>,
<closed, rightB, doOpen, opened>,
<opened, alarm, ε, final>,
<closed, alarm, ε, final>$\}$

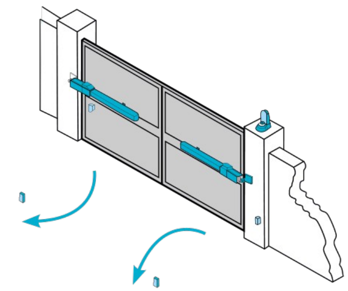
A finite state transducer is defined by $< Q, q_0, \mathcal{F}, \Sigma_I, \Sigma_O, \delta >$

Running Example

Σ_I = {leftB, rightB, alarm}



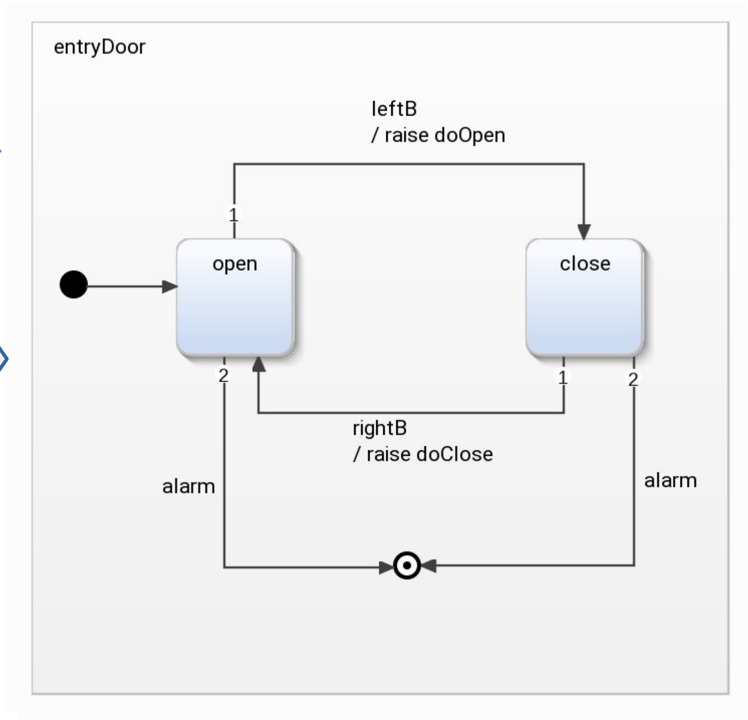
Σ_O = {doOpen, doClose}



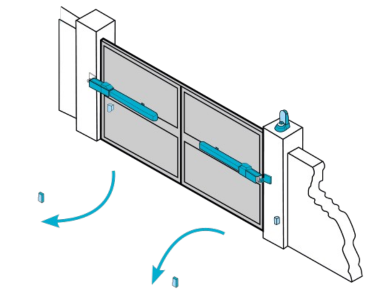
A finite state transducer is defined by $< Q, q_0, \mathcal{F}, \Sigma_I, \Sigma_O, \delta >$

Running Example

Σ_I = {leftB, rightB, alarm}



Σ_O = {doOpen, doClose}



Similarities with the automata studied in *LFA*:

- It is a mean to represents the, possibly infinite, set of “meaningful” words in input of the system
- It is possible to compose automaton together (we’ll see it later)

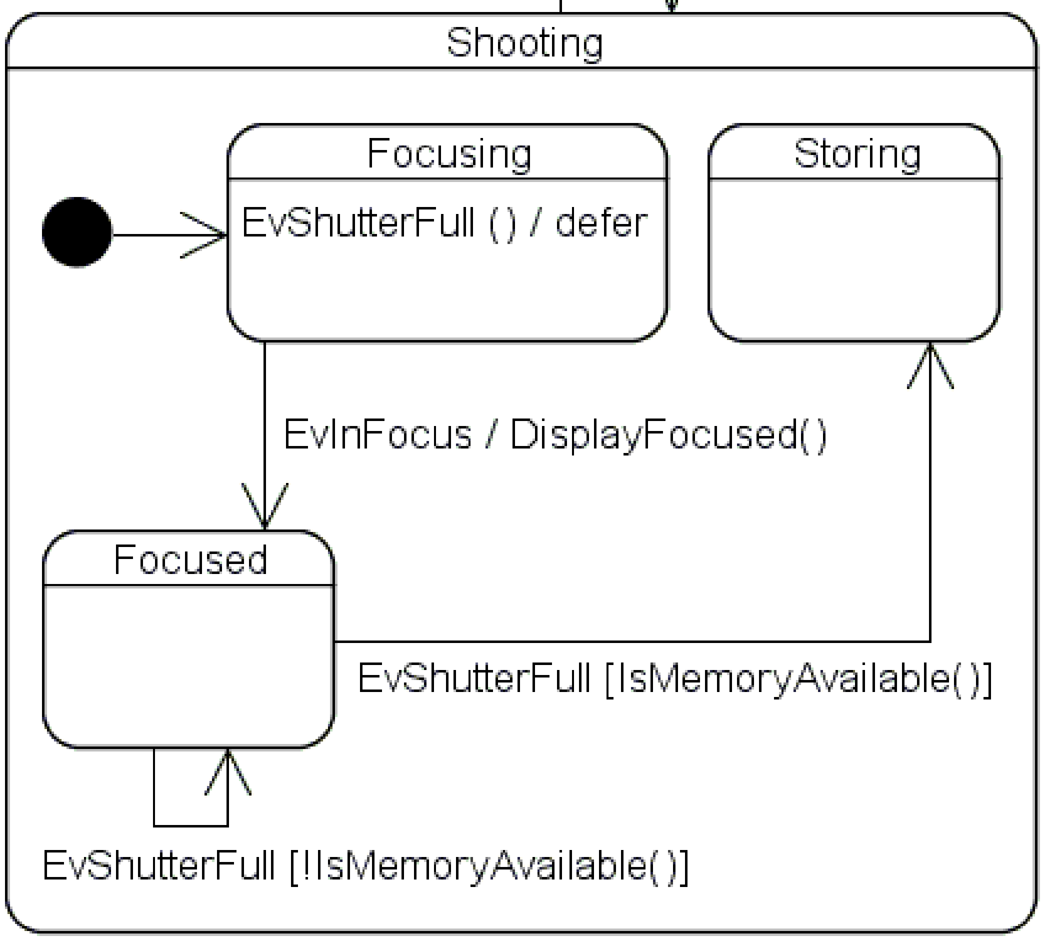
Differences with the automata studied in *LFA*:

- We distinguish the input and the output alphabets
- It is seldom used to reason on languages but rather to structure and reason on control code.

A finite state transducer is defined by $< Q, q_0, \mathcal{F}, \Sigma_I, \Sigma_O, \delta >$

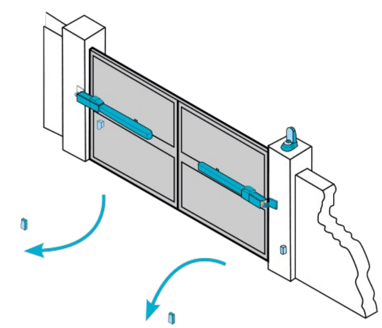
Running

Σ_I = {leftB, rightB, al



[

{doOpen, doClose}

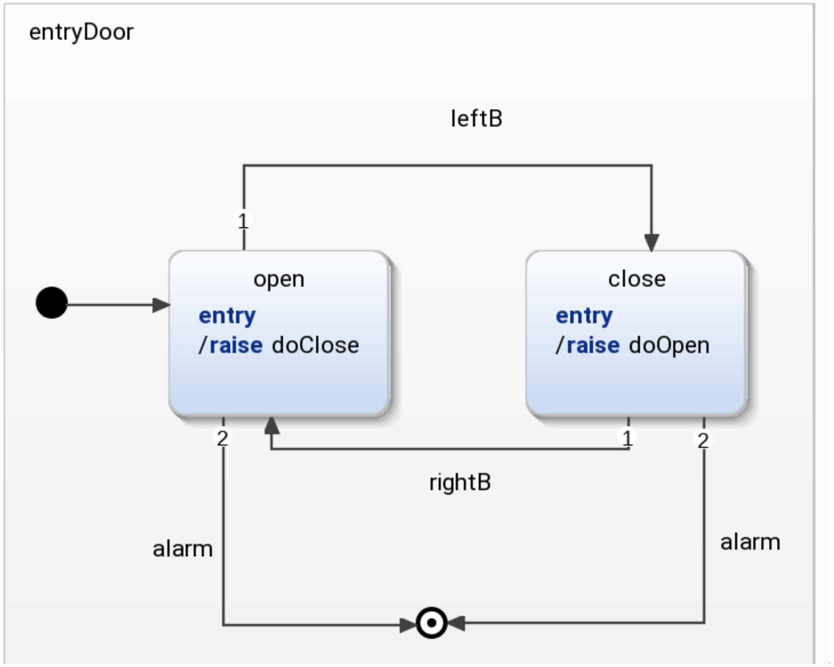


A finite state transducer is defined by < $\mathcal{Q}$, q_0, $\mathcal{F}$, Σ_I, Σ_O, δ >

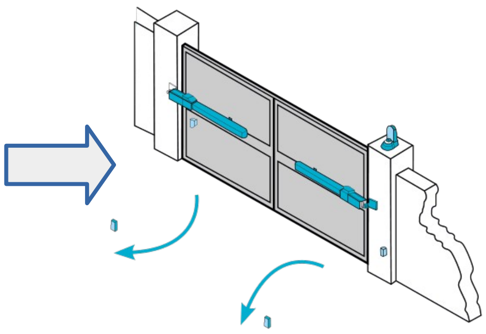
→ note 1: pragmatically in executable FSMs, Σ_I is often a set of **events** and Σ_O is a set of ***Actions*** (for instance the sending of an event, the call to a method, etc).

Running Example

Σ_I = {open, close, stop}



Σ_O = {doOpen, doClose}



A finite state transducer is defined by < Q , q_0, $\mathcal{F}$, Σ_I, Σ_O, δ >

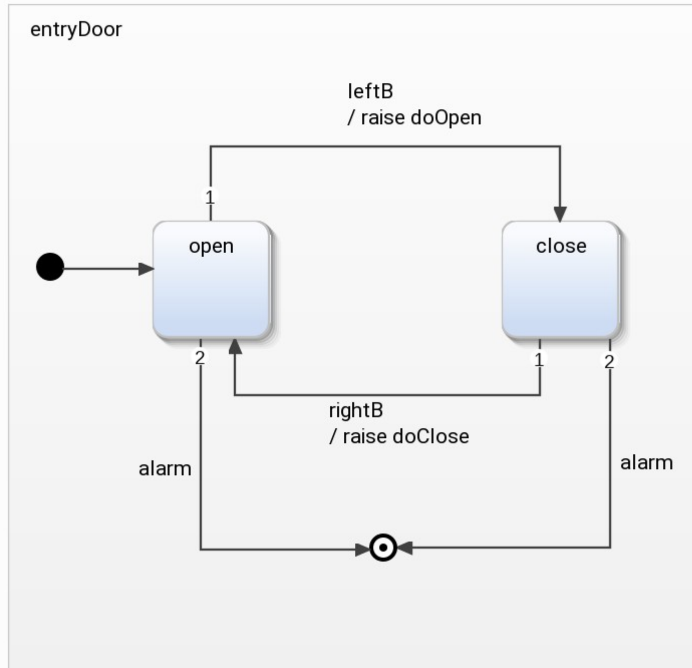
→ note 1: pragmatically in executable FSMs, Σ_I is often a set of events and Σ_O is a set of *Actions* (for instance the sending of an event, the call to a method, etc).

→ note 2: the same behavior can be encoded by a Moore machine, the difference being in the transition function (δ) and a new output function (f_o)

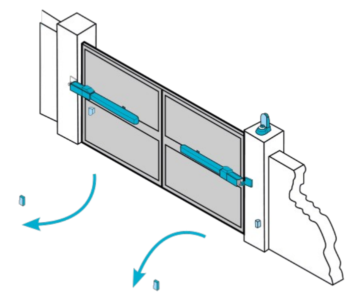
$$\delta \subseteq Q \times \Sigma_I \times Q \qquad f_o : Q \rightarrow \Sigma_O$$

Running Example

Σ_I = {leftB, rightB, alarm}



Σ_O = {doOpen, doClose}



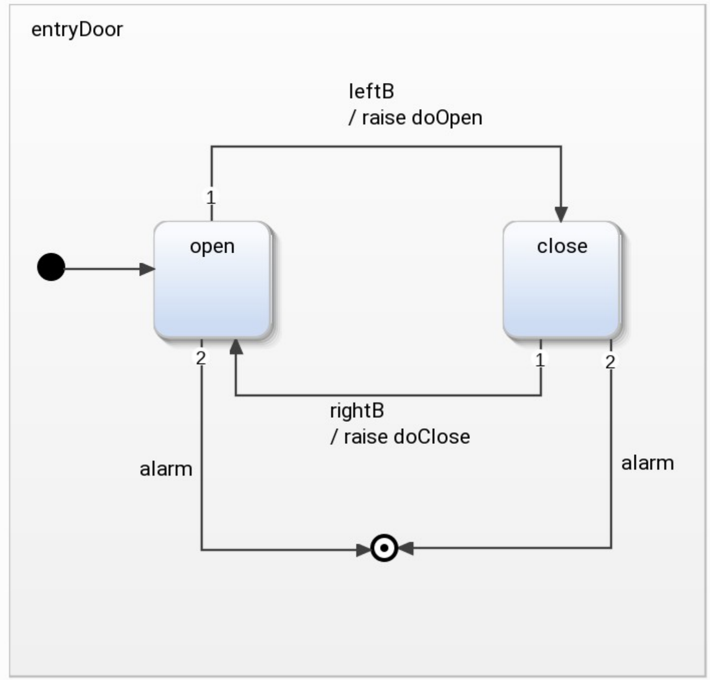
A finite state transducer is defined by $< \mathcal{Q}\, , q_0 , \mathcal{F} , \Sigma_I , \Sigma_O , \delta >$

→ note 1: pragmatically in executable FSM, Σ_I is often a set of events and Σ_O is a set of *Action* (for example the sending of an event, the call to a method, etc).

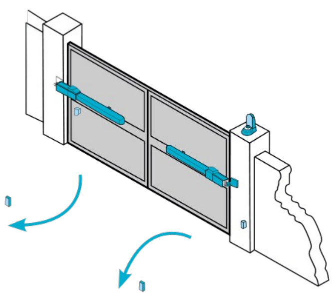
- Events are one of the basic concepts in SCXML since they drive most transitions.

Running Example

Σ_I = {leftB, rightB, alarm}



Σ_O = {doOpen, doClose}



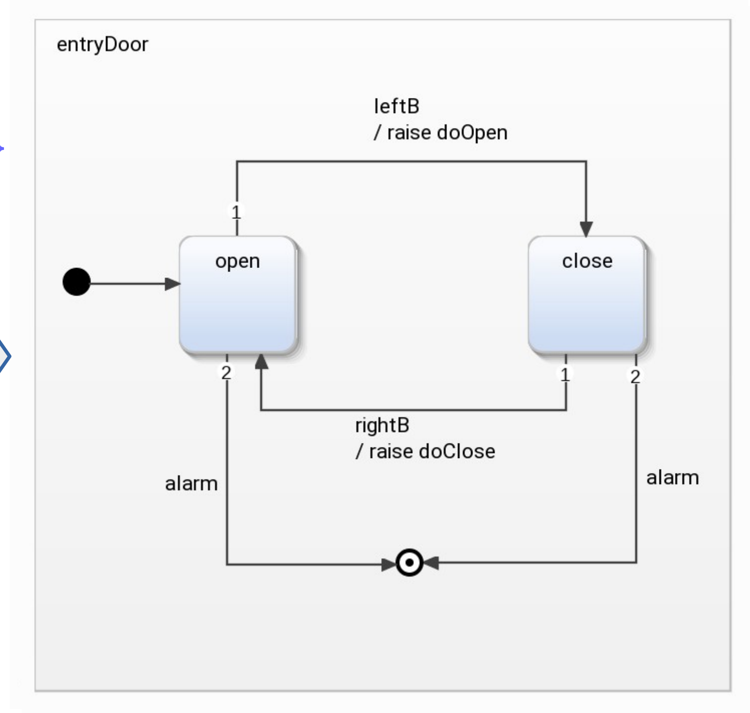
A finite state transducer is defined by $< \mathcal{Q}, q_0, \mathcal{F}, \Sigma_I, \Sigma_O, \delta >$

→ note 1: pragmatically in executable FSM, Σ_I is often a set of events and Σ_O is a set of *Action* (for example the sending of an event, the call to a method, etc).

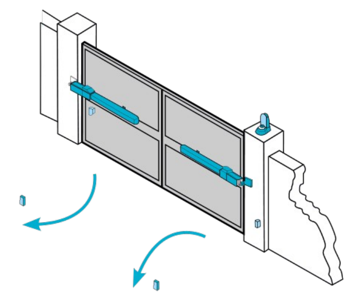
- Events are one of the basic concepts in SCXML since they drive most transitions.
- For example, a transition with an 'event' attribute of "error foo" will match event names "error", "error.send", "error.send.failed", etc. (or "foo", "foo.bar" etc.) but would not match events named "errors.my.custom", "errorhandler.mistake", "errorsend" or "foobar".
- [...] an event descriptor MAY also end with the wildcard '.*', which matches zero or more tokens at the end of the processed event's name. Note that a transition with 'event' of "error", one with "error.", and one with "error.*" are functionally equivalent since they are token prefixes of exactly the same set of event names.
- An event designator consisting solely of "*" can be used as a wildcard matching any sequence of tokens, and thus any event

Running Example

Σ_I = {leftB, rightB, alarm}



Σ_O = {doOpen, doClose}



A finite state transducer is defined by $< \mathcal{Q}, q_0, \mathcal{F}, \Sigma_I, \Sigma_O, \delta >$

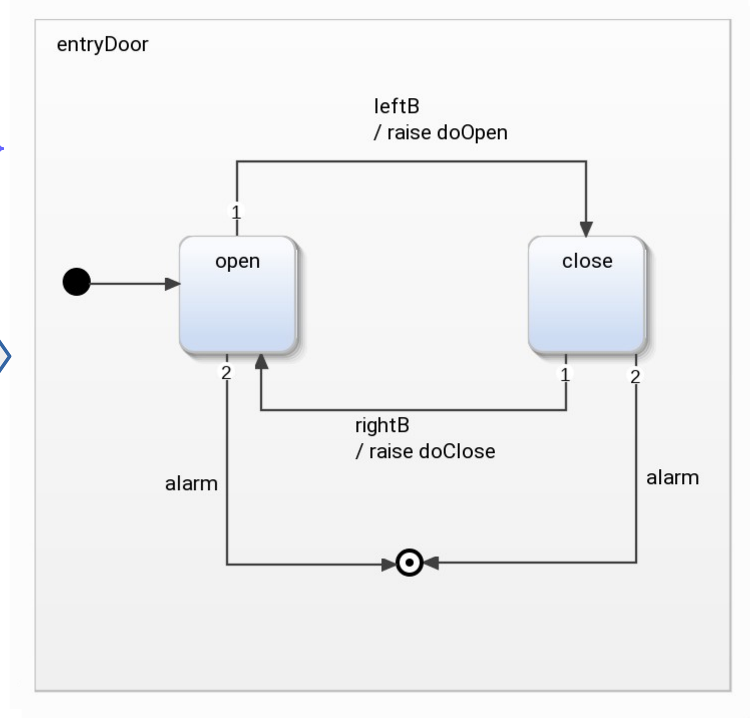
→ it can be seen as a directed graph where $\mathcal{Q}$ is the set of vertices and δ the set of “labeled” edges.

We can “ask questions” to the graph:

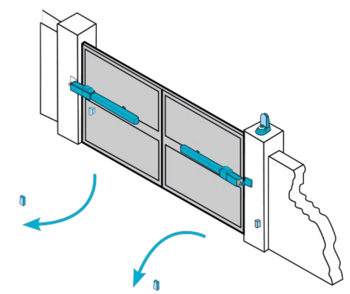
- *Classical ones*: Is there any cycle ? Is there a path from state X to state Y ? What is the shortest path from X to Y ? etc.
- *Temporal logic*: whenever `close` is requested, is the door eventually *`closed`*

Running Example

Σ_I = {leftB, rightB, alarm}



Σ_O = {doOpen, doClose}



A finite state transducer is defined by $< Q\,,\, q_0,\, \mathcal{F}\,,\, \Sigma_I,\, \Sigma_O,\, \delta >$

→ it can be seen as a directed graph where Q is the set of vertices and δ the set of “labeled” edges.

We can “ask questions” to the graph:

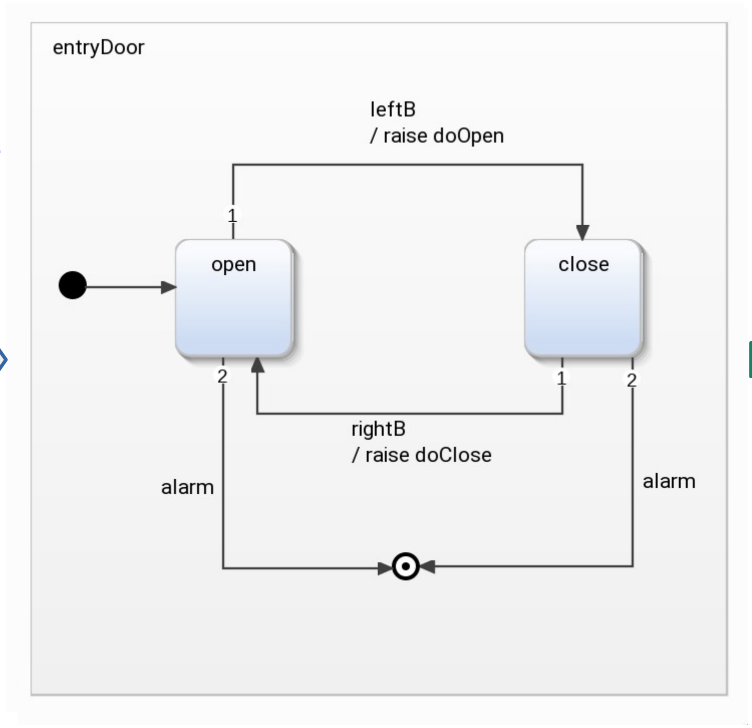
- *Classical ones*: Is there any cycle ? Is there a path from state X to state Y ? What is the shortest path from X to Y ? etc.
- *Temporal logic*: whenever `close` is requested, is the door eventually *`closed`*

→ **note:** if Boolean conditions are used to guard the transition, it is more difficult to “ask question” to the graph since both the conditions and the underlying action language need to be analyzed first and usually depends on arbitrary data from the environment.

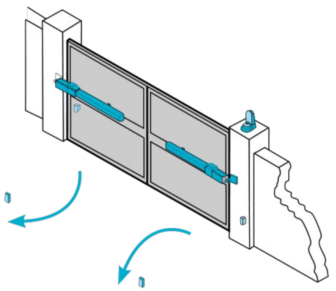
Running Example

- We want to model the controller of an entry door by using a FSM.

Σ_I = {leftB, rightB, alarm}



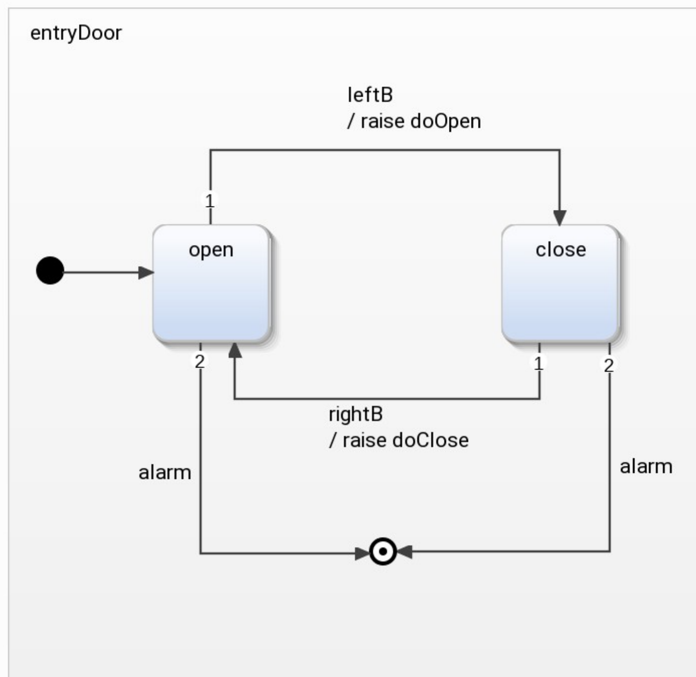
Σ_O = {doOpen, doClose}



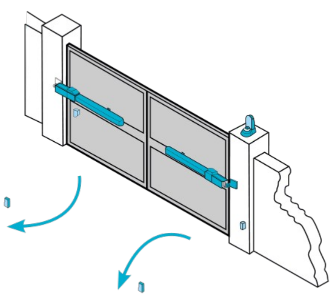
Running Example

- We want to model the controller of an entry door by using a FSM.

Σ_I = {leftB, rightB, alarm}

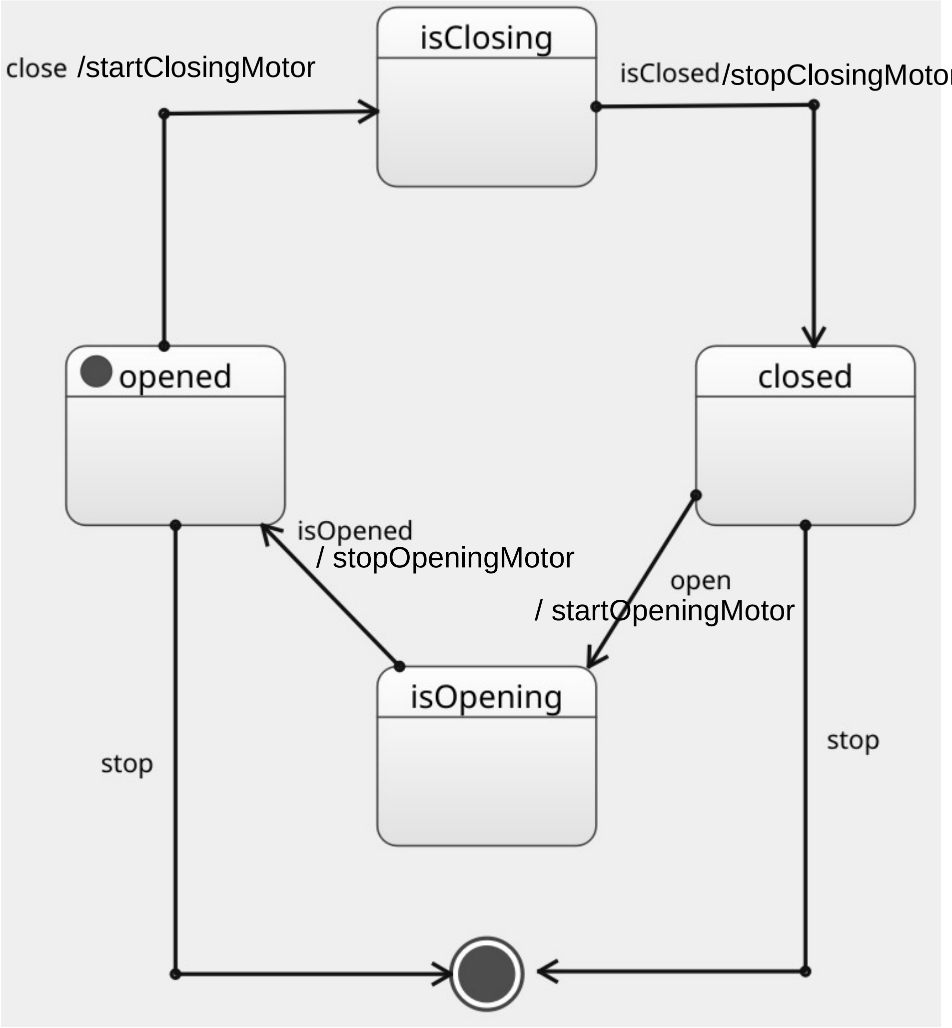


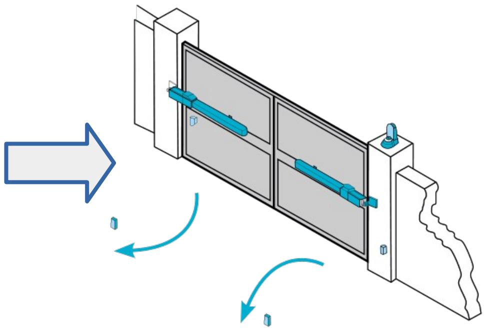
Σ_O = {doOpen, doClose}



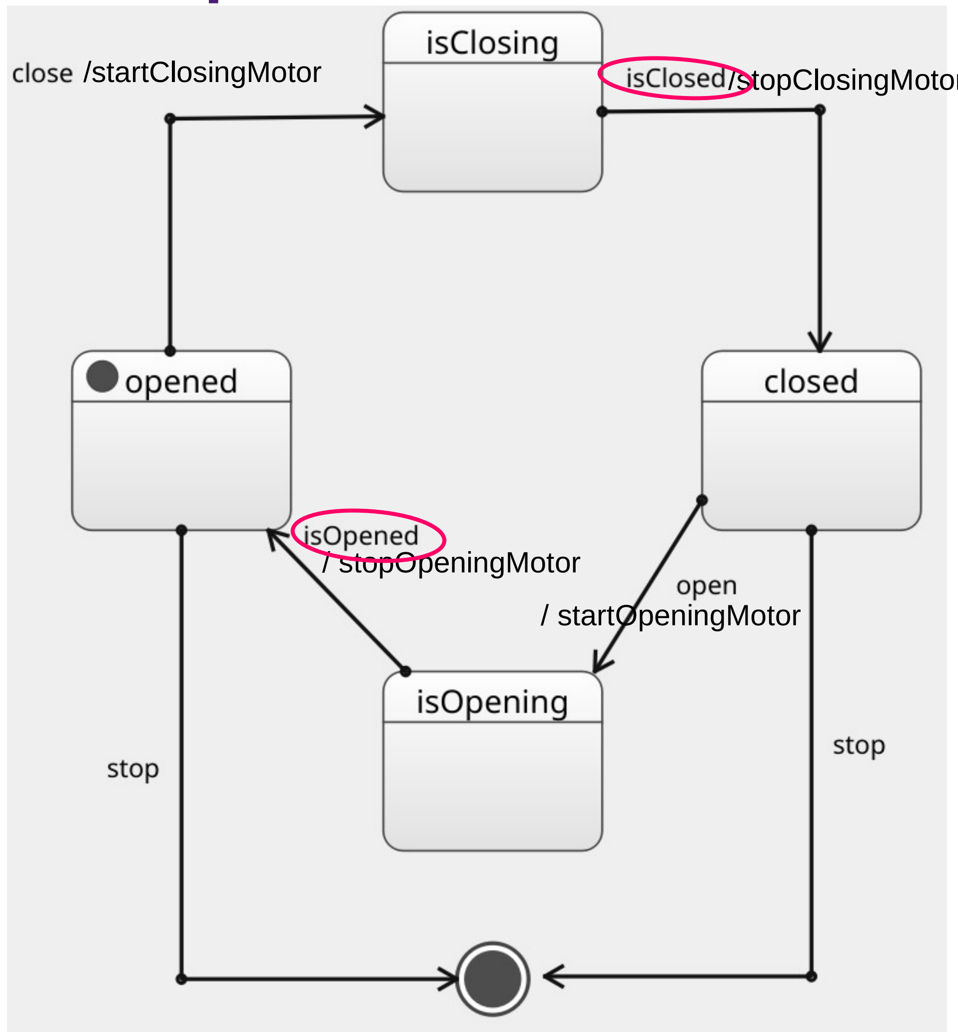
Strong abstraction...

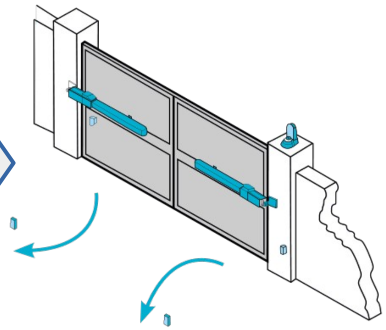
Running Example





Running Example



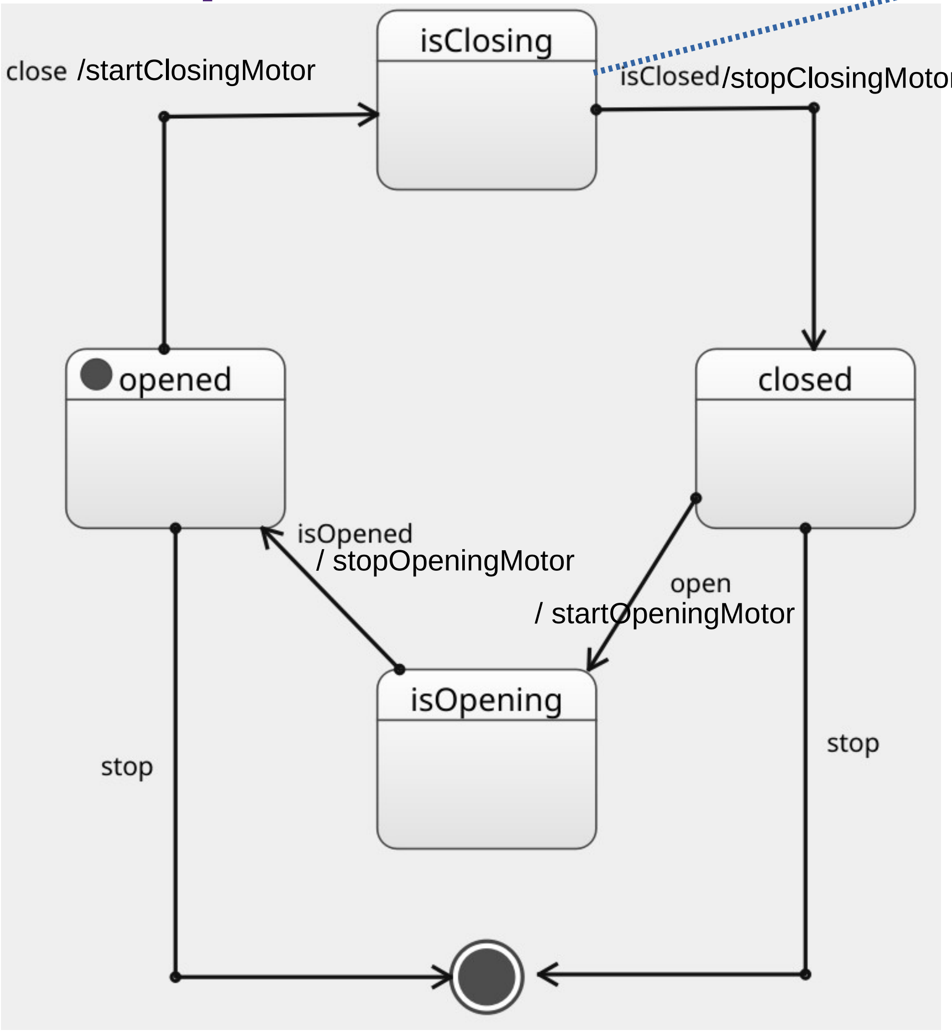


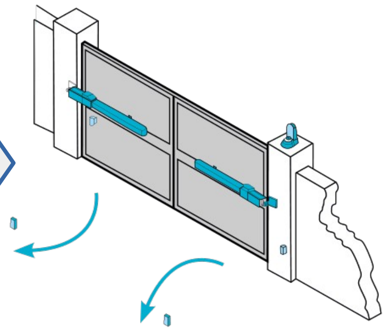
We do not know where the events isClosed and isOpened are coming from (e.g., new stop sensors, from “the environment”).

Running Example

```
onEntry:
send isClosed
after 25s
```

Or a self loop !

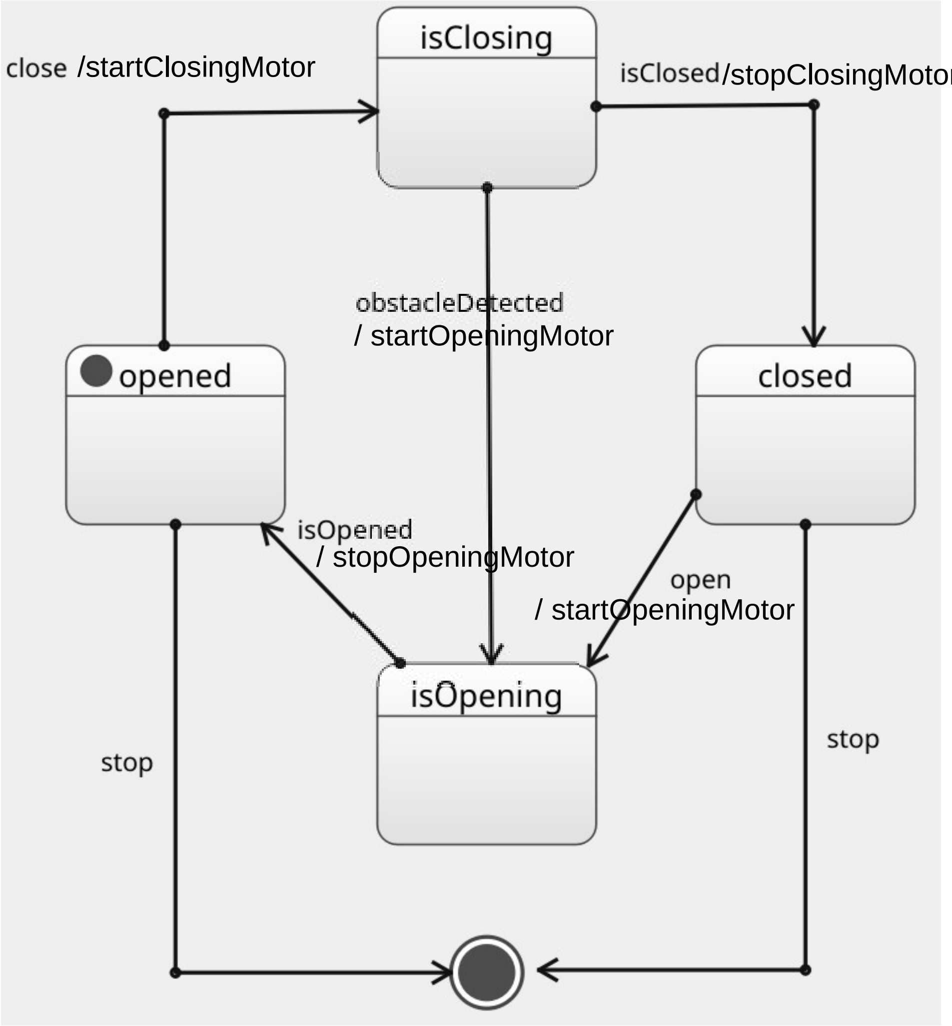


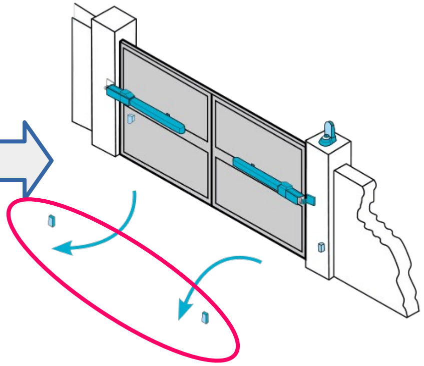


We do not know where the events isClosed and isOpened are coming from (e.g., new stop sensors, from “the environment”).

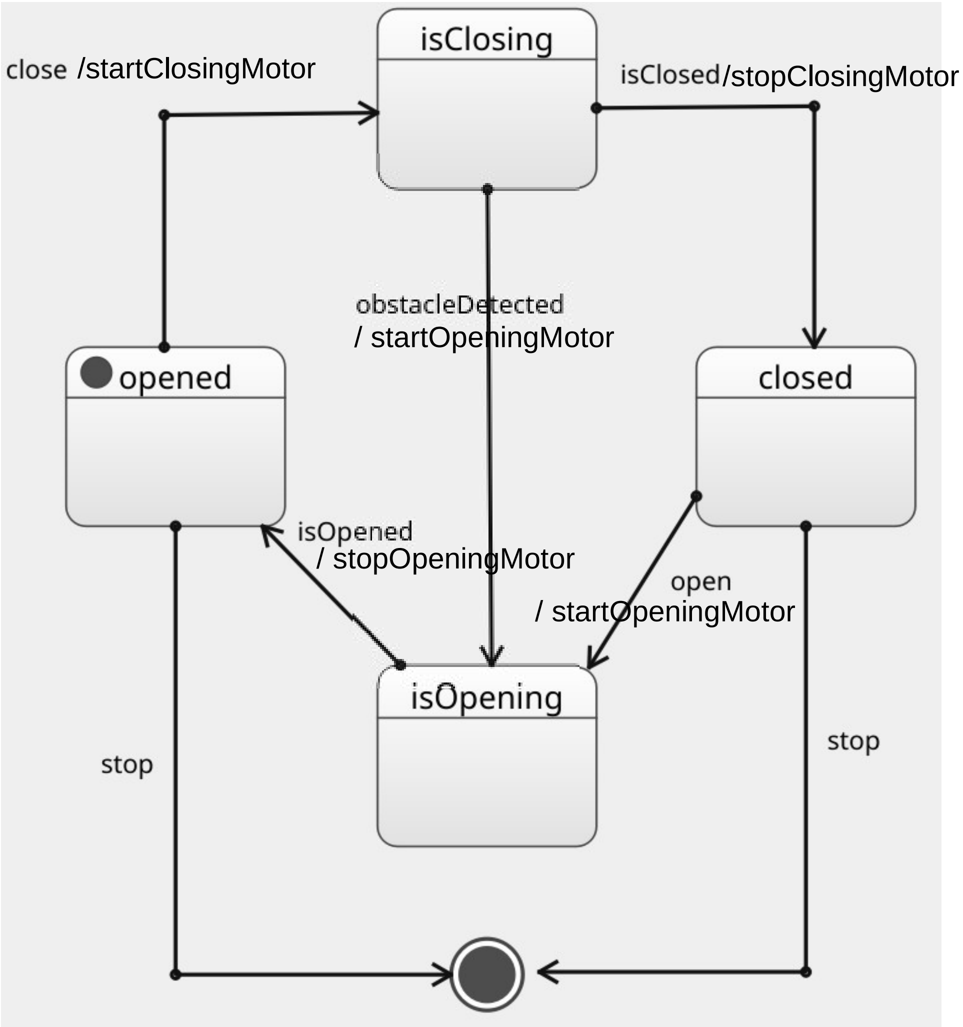
if we want them to occur after some **time** following the entry in the isClosing state, it is not a traditional finite state transducer anymore but a timed automata

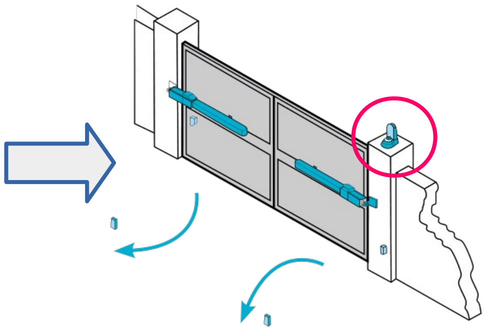
Running Example



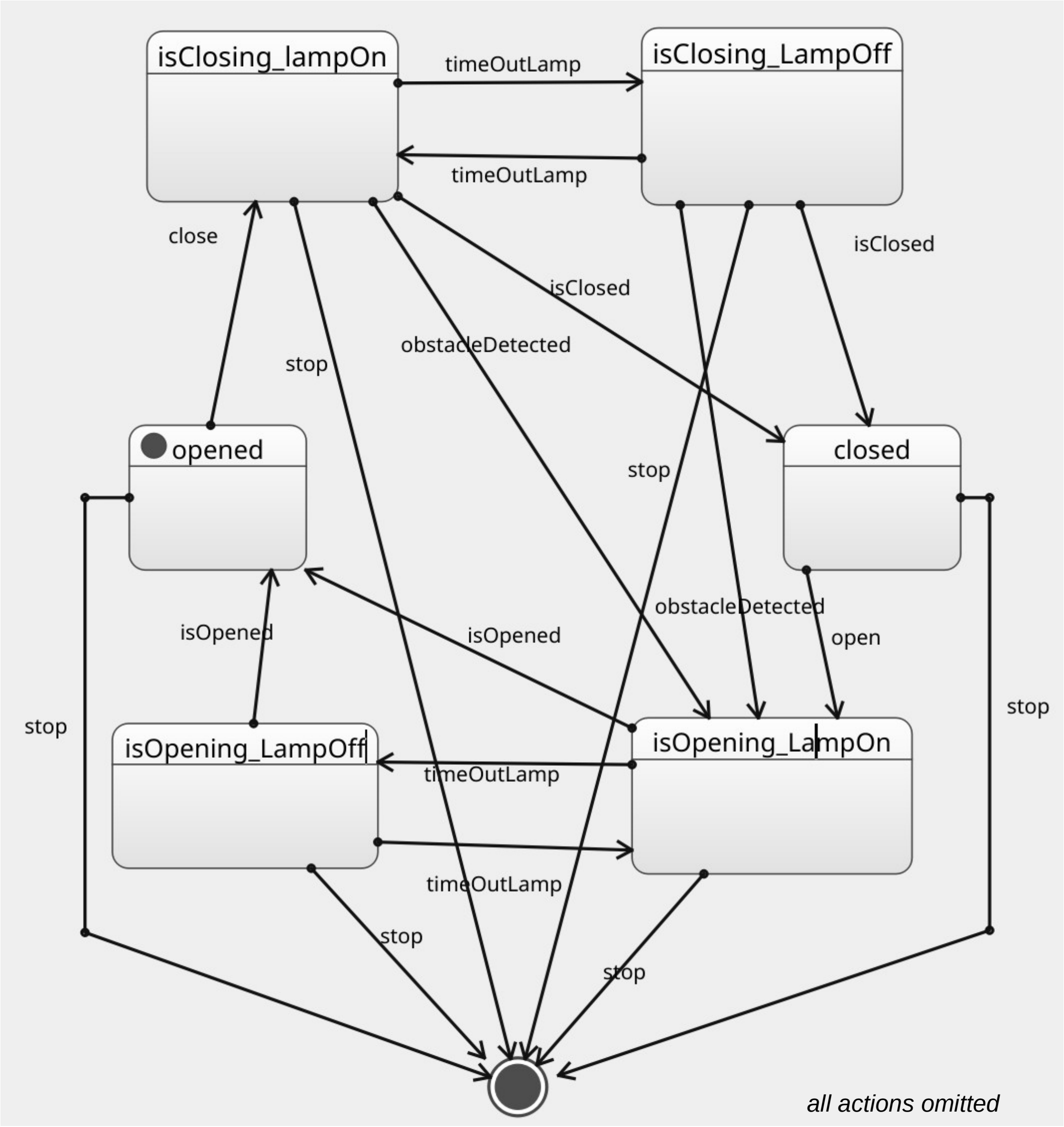


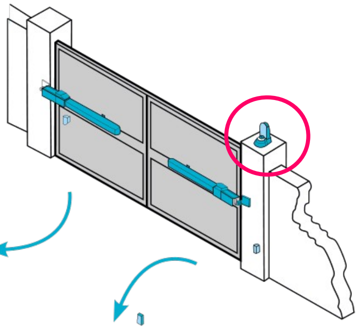
Running Example





Running Example





wasn't it supposed to help ?

State Charts

statecharts = state-diagrams + depth

+ orthogonality + broadcast-communication.

David Harel
Statecharts: A visual formalism for complex systems
Science of computer programming 8 (3), 231-274
1987

State Charts

statecharts = state-diagrams + depth

+ orthogonality + broadcast-communication.