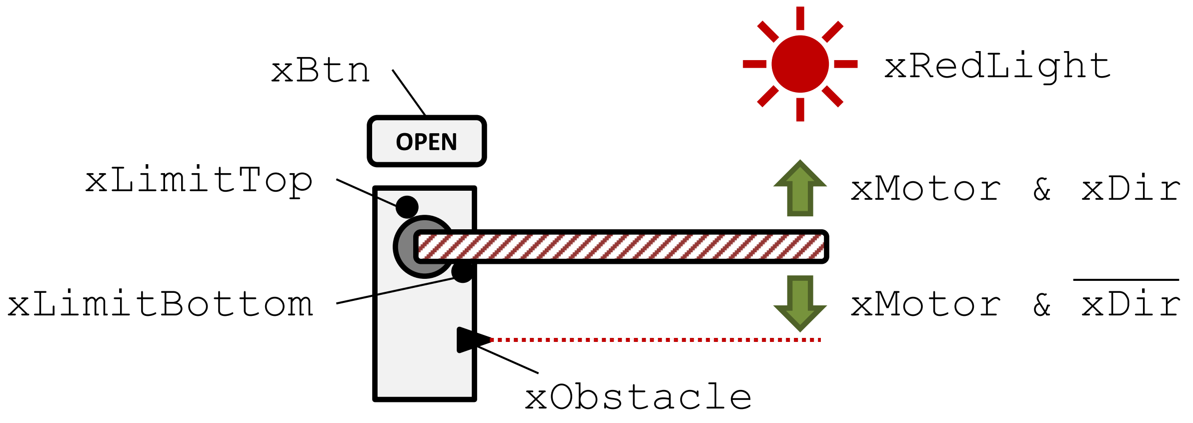
State machine-based control

Industrial Control

KOVÁCS Gábor

gkovacs@iit.bme.hu

Problem



PLC inputs

- `xLimitBottom, xLimitTop`: limit switches, reporting whether the barrier is in its top or bottom position
- `xObstacle`: proximity switch, which is active, if there is an obstacle below the barrier
- `xBtn`: pushbutton for opening the barrier

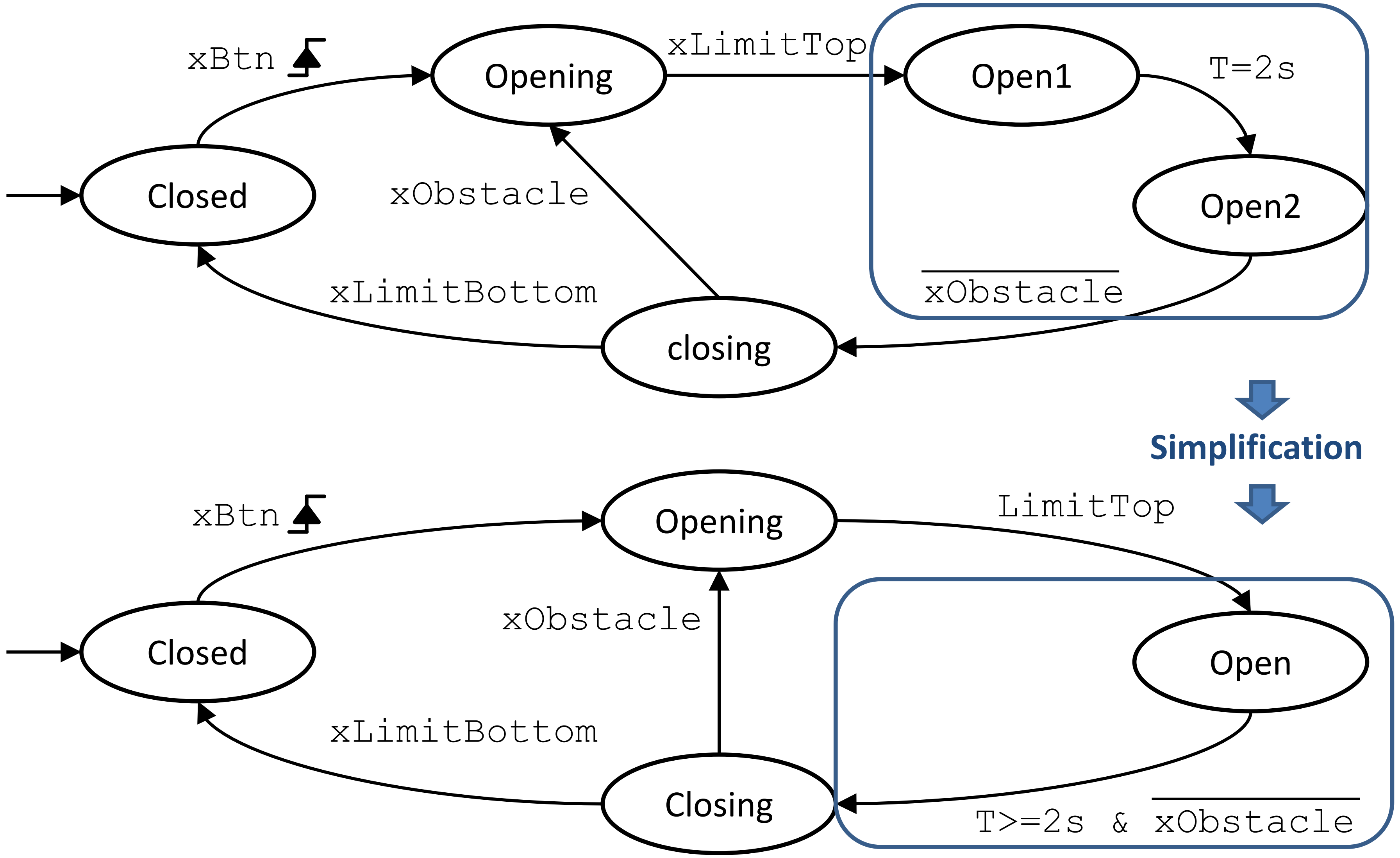
PLC outputs

- `xMotor`: movement of barrier
- `xDir`: direction of movement if motor is turned on (0: down, 1: up)
- `xRedLight`: red light

Specification

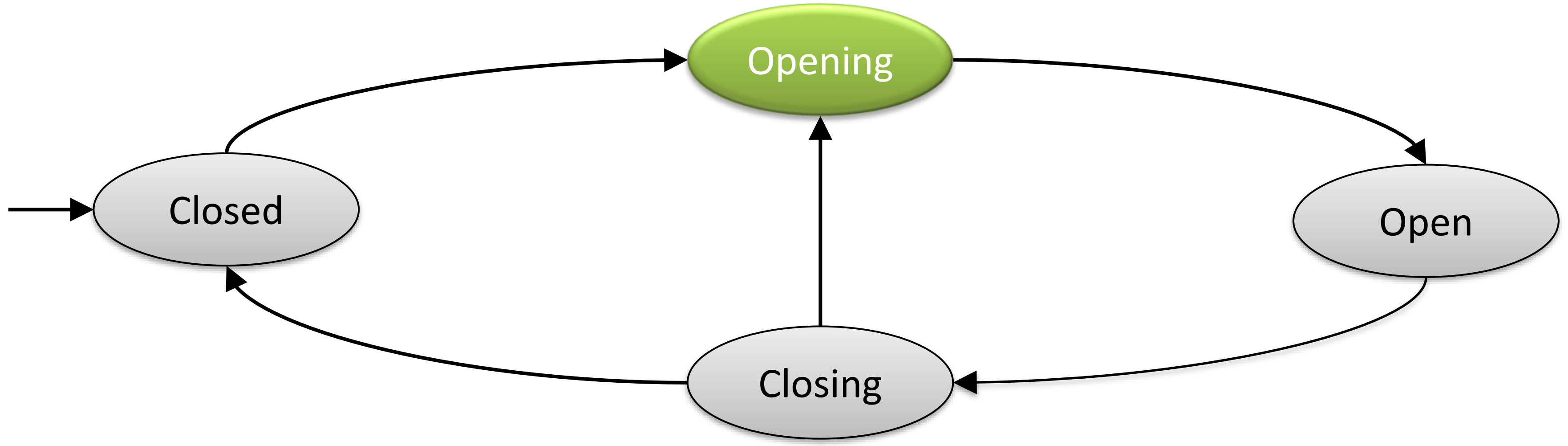
- The barrier should be lifted upon pressing the pushbutton (rising edge)
- The barrier should be lifted until the top limit switch is reached, and then should be left unactuated
- The barrier should stay at the top position for 2 seconds – after that, the barrier should be lowered if no obstacle is present
- If an obstacle is detected while the barrier is being lowered, it should be lifted again
- Upon returning to the botttom position, the barrier should be opened after the next rising edge of the button
- The red light should be on in any configuration except the fully open position of the barrier

State machine of the solution



States

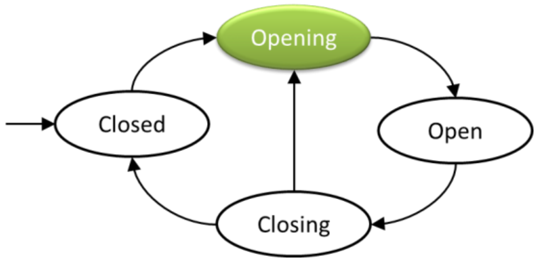
- A state represents a given step or phase of the operation
- Only one of the states should be active at a time (actual state)

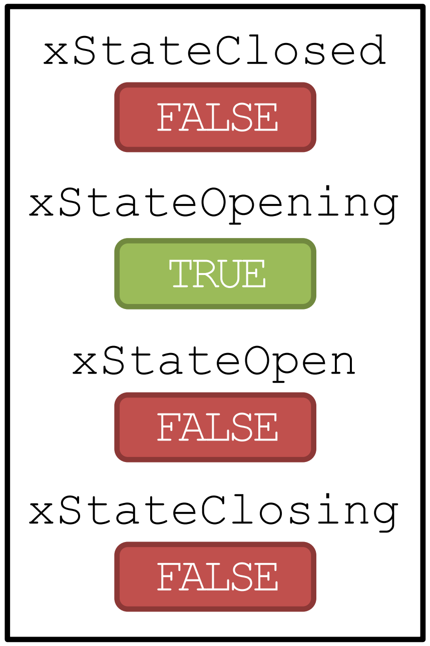


Representation of states

- **Boolean variables associated to each state**
  - active (TRUE) state of a Boolean variable represents that the corresponding state is the active one
  - only one of these variables can be active at a time
  - pro: simplicity
  - con: need to declare and use a large number of variables (number of states)

```
VAR
    xStateClosed   : BOOL;
    xStateOpening  : BOOL;
    xStateOpen     : BOOL;
    xStateClosing  : BOOL;
END_VAR
```



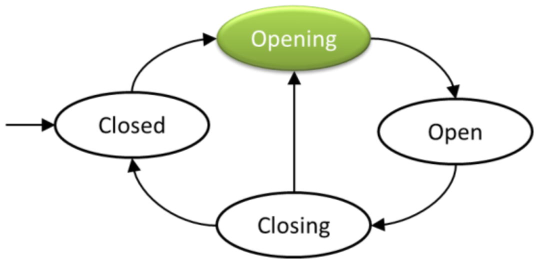


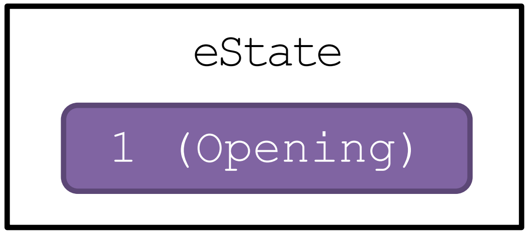
xStateClosed	FALSE
xStateOpening	TRUE
xStateOpen	FALSE
xStateClosing	FALSE

Representation of states

- One single state variable
  - variable contains the number (ID) of the active state
  - an enumerated data type allows the use of textual labels
  - pro: one single variable
  - con: requires non-Boolean comparison and assignment operations (cumbersome to realize in LD)

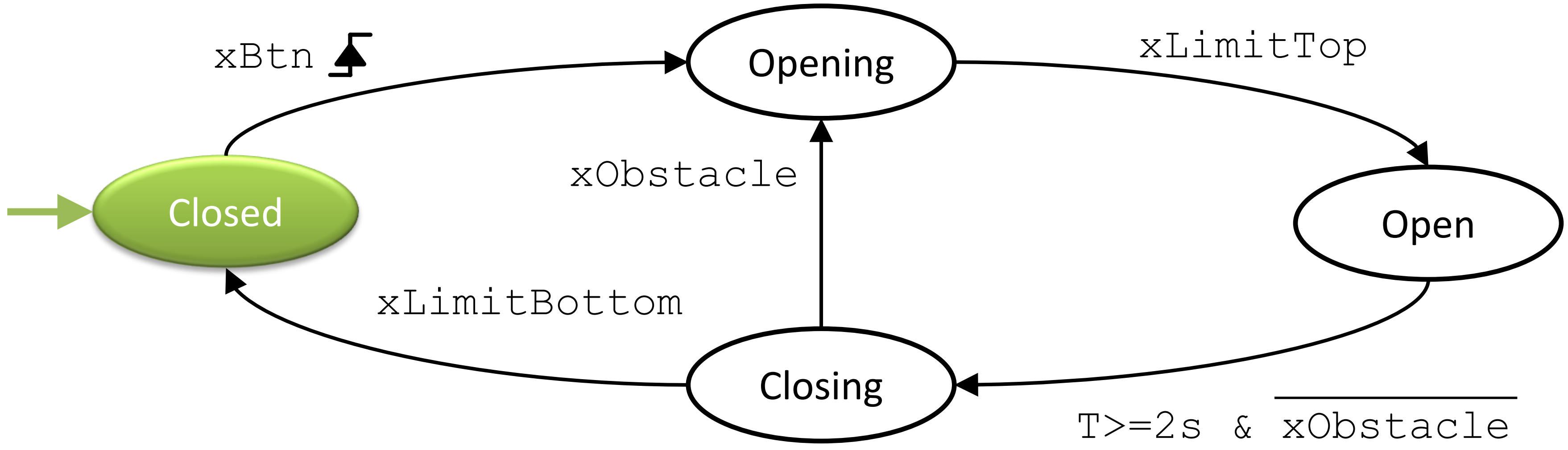
```
VAR
 eState : (Closed, Opening,
           Open, Closed);
END_VAR
```





Initial state

Which is the active state when the state machine is initialized?



Initialization of the state machine by specific initial values of state variable(s)

- In case of Boolean state variables
  - initial value of Boolean (`BOOL`) variables is `FALSE`
  - an instance-specific initial value of `TRUE` should be assigned to the variable corresponding to the initial state in the declaration part:

```
xStateClosed: BOOL := TRUE;
```

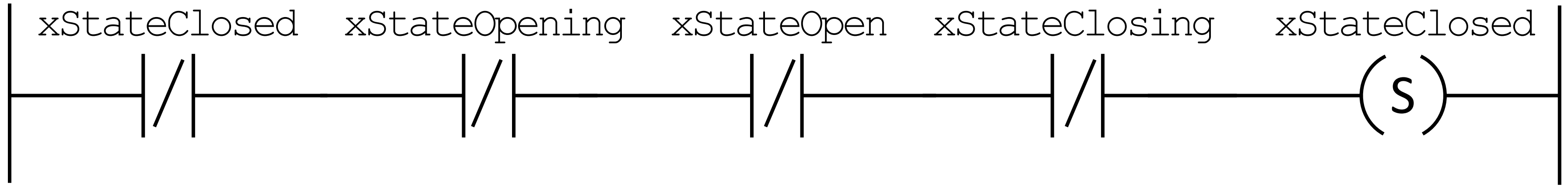
- In case of a single state variable
  - initial value of a variable of enumerated data type is the first label in the list (a different initial value might also be assigned):

```
xState: (Closed,Opening,Open) := Closed
```

Initializing the state machine in the code

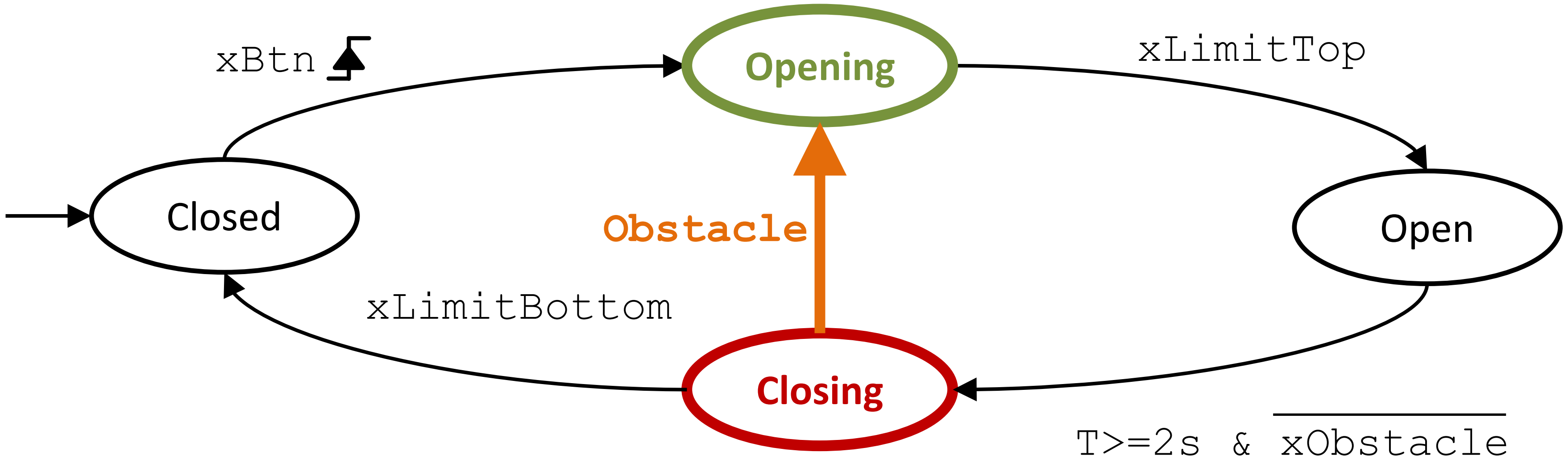
- Principle: if there is no active state, the register of the initial state shall be set
- Generic solution for Boolean state variables
- Might be used even for most simple, legacy PLCs

Not required if initial value of state variables can be set explicitly (e.g. in CODESYS)



State transitions

„I am in the state `Closing`. Where should I move to if the signal `Obstacle` is active?”



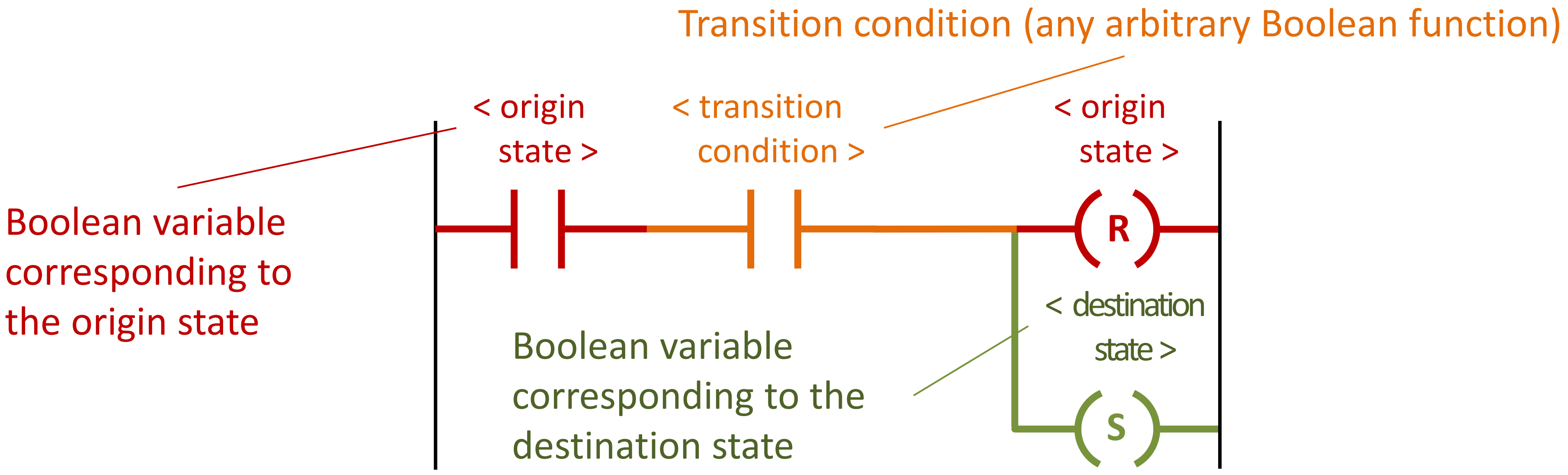
State transitions

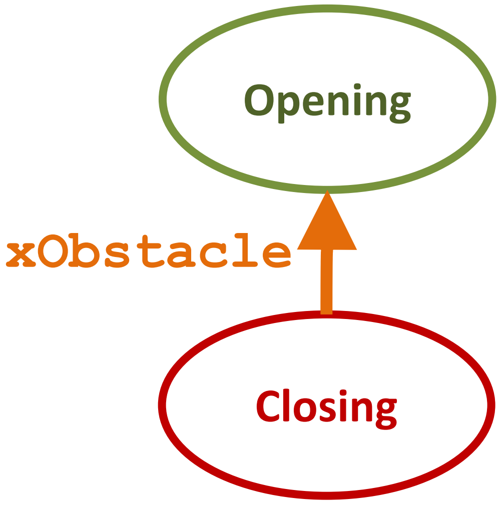
- A state transition should be executed if and only if
  - its origin state is active

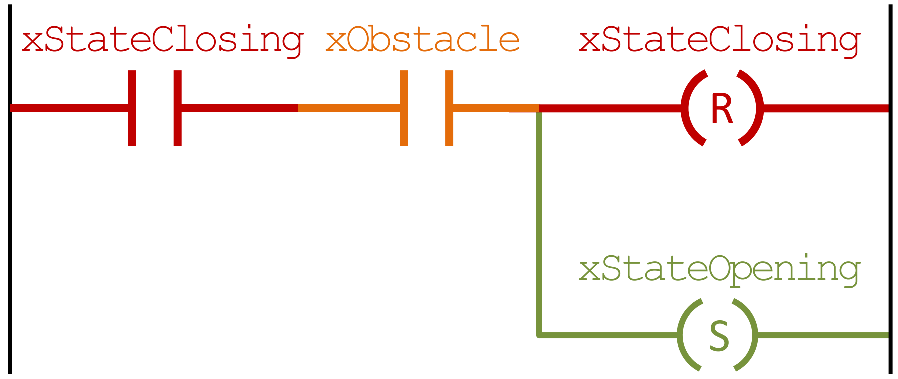
    AND
  - its condition evaluates to TRUE

- During a state transition, the actual state needs to be updated:
  - reset the state variable of the origin state
  - set the state variable of the destination state

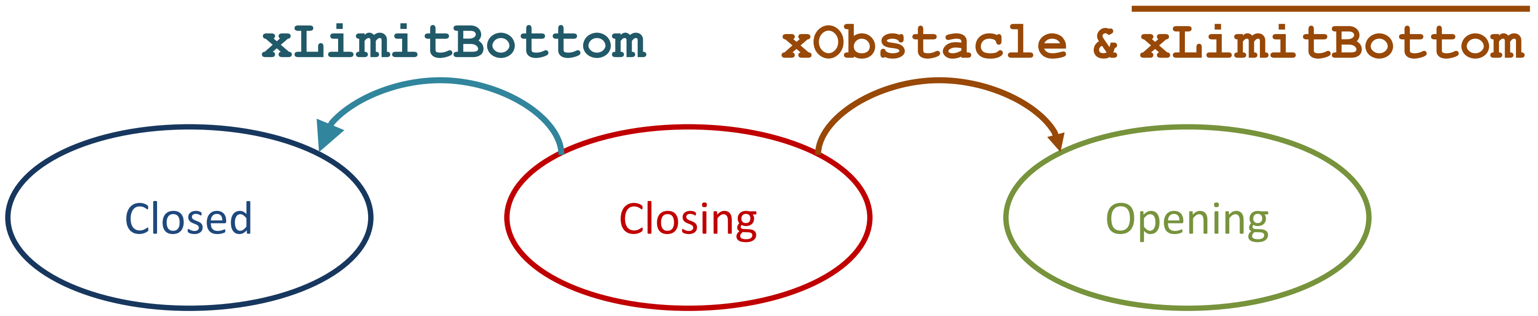
Implementation of state transitions


Transition condition (any arbitrary Boolean function)
< origin state >
< transition condition >
< origin state >
R
< destination state >
S
Boolean variable corresponding to the origin state
Boolean variable corresponding to the destination state

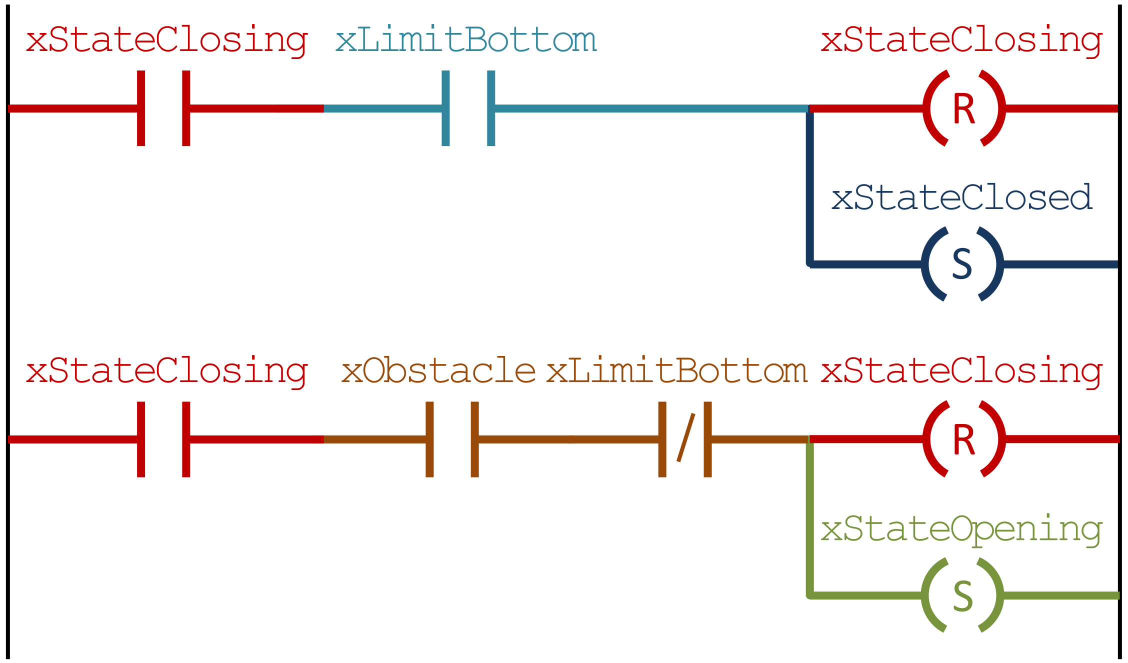




State transitions – mutual exclusion

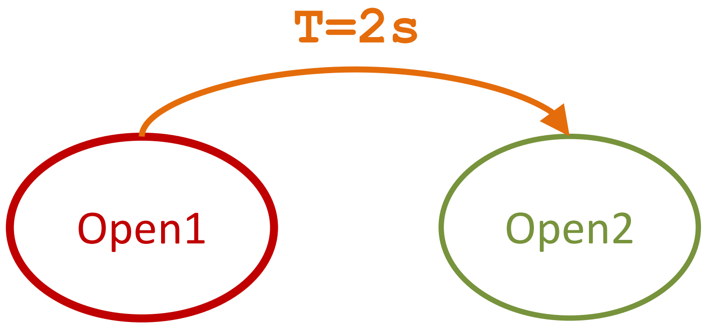


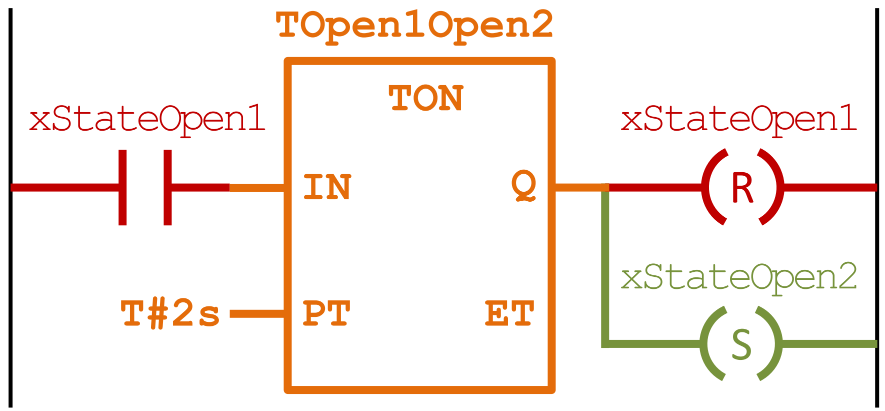
- If more than one transition originates from a state, their conditions should be transformed to be mutually exclusives
- Mutual exclusion defines also a priority of transitions if more conditions are satisfied at the same time



Transition timing

- It is common in practice that a transition should be executed after spending a given time in its origin state
- Such a condition can be represented by a TON timer
- **Best practice: use a unique timer for each timed transition**

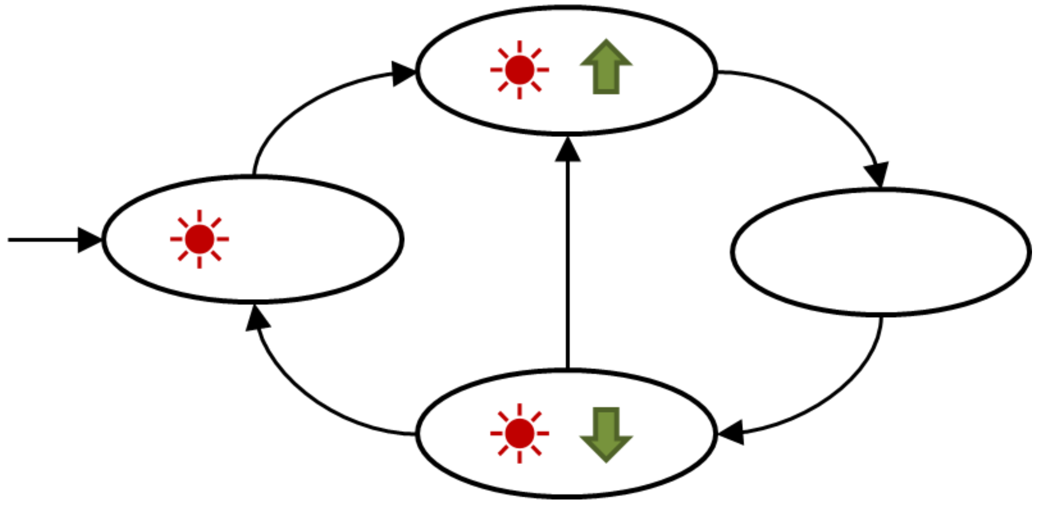




The state machine of the barrier has been simplified by joining states *Open1* and *Open2*, hence this part of code will not be used directly.

Output mapping

- Mealy machine: value of outputs depend both on the actual state and on the value of inputs
- **Moore machine: value of outputs depend on the actual state only**



Output mapping

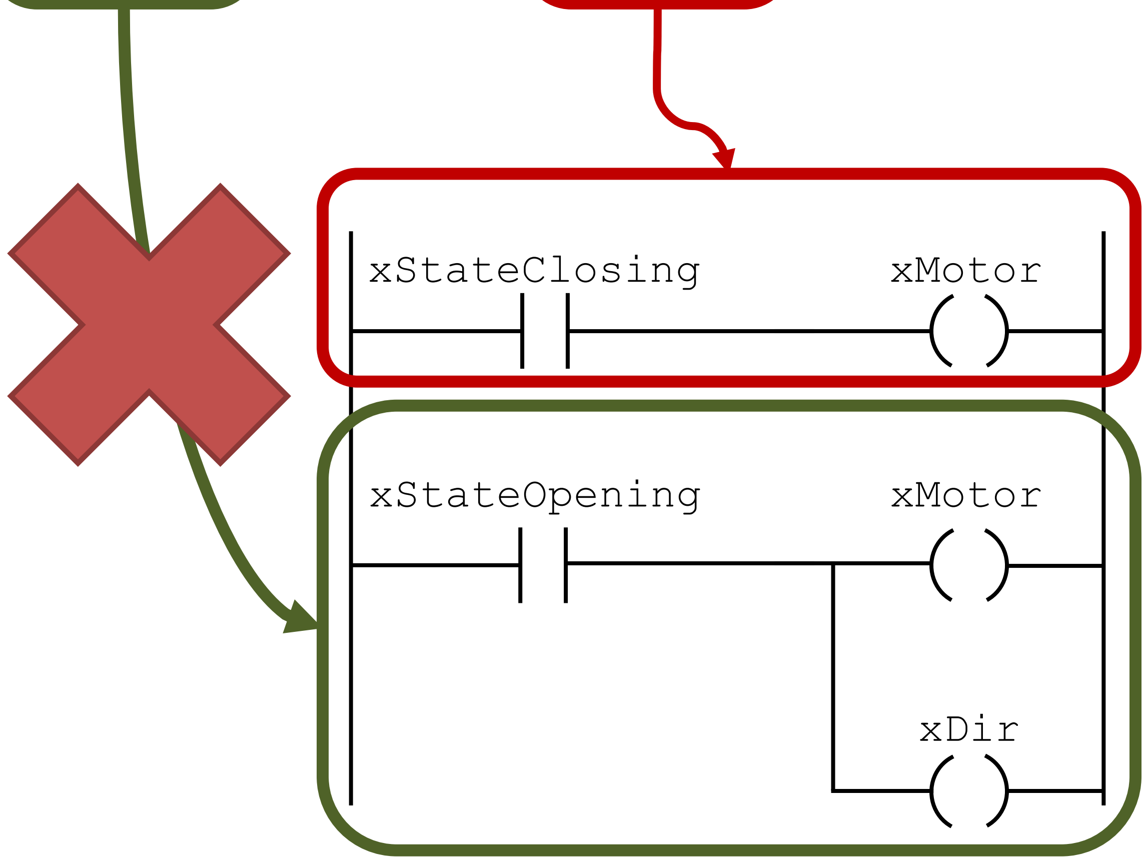
- At each state a given combination of outputs should be activated
- Output mapping can be defined as a table

State	Closed	Opening	Open	Closing
`xMotor`	0	1	0	1
`xDir`	X	1	X	0
`xRedLight`	1	1	0	1

X : don’t care

Implementation of output mapping

State	Closed	Opening	Open	Closing
xMotor	0	1	0	1
xDir	X	1	X	0

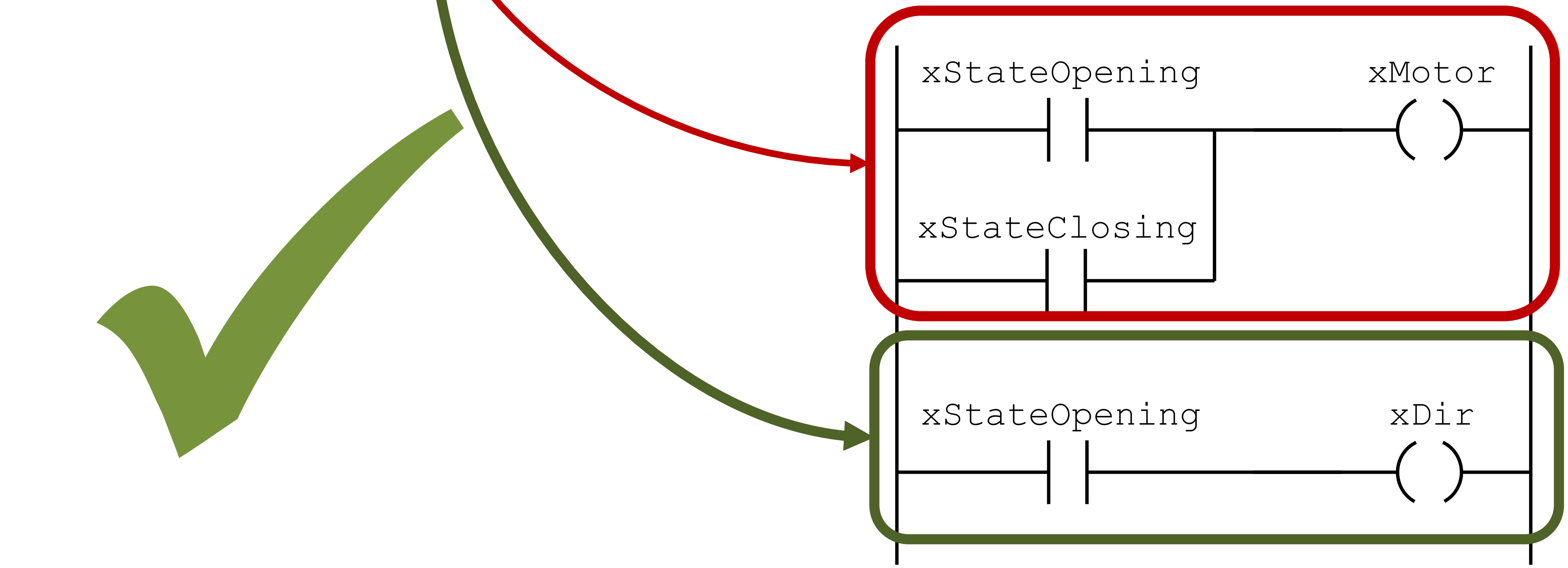


Using this implementation, the motor will not operate in the `Closing` state

Implementation of output mapping

Állapot	Closed	Opening	Open	Closing
xMotor	0	1	0	1
xDir	X	1	X	0

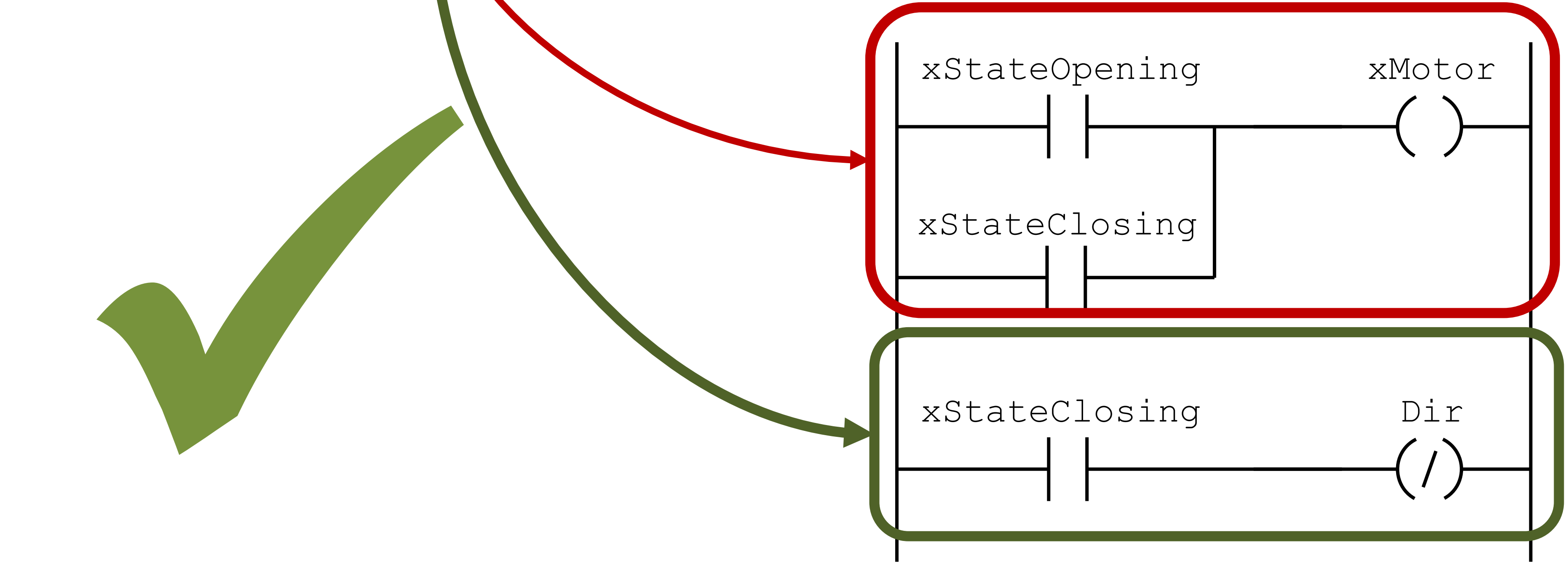
Write each output in one single rung only by an NO coil. State variables corresponding to the state in which the given output is active should be associated to NO contacts in parallel connection.



Implementation of output mapping

Állapot	Closed	Opening	Open	Closing
xMotor	0	1	0	1
xDir	X	1	X	0

Output is written in one single rung only by an NC coil. State variables corresponding to the state in which the given output is inactive should be associated to NO contacts in parallel connection.



Formally

A Moore machine is a 6-tuple: $C = (Q, \Lambda, \Omega, \rho, \varphi, q_0)$

- Q*: set of states*
- Λ*: set of input symbols (conditions and events)*
- Ω*: set of output symbols*
- $\rho: Q \times \Sigma \rightarrow Q$*: state transition function*
- $\varphi: Q \rightarrow \Omega$*: output mapping*
- q_0*: initial state*

Informally: state machine =

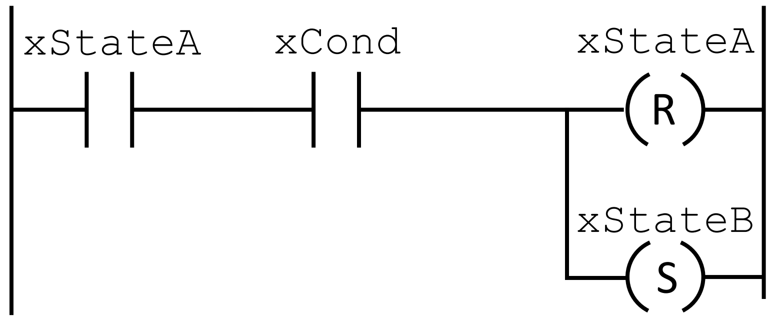
states

```
VAR
   xStateB, xStateC,… : BOOL;
```

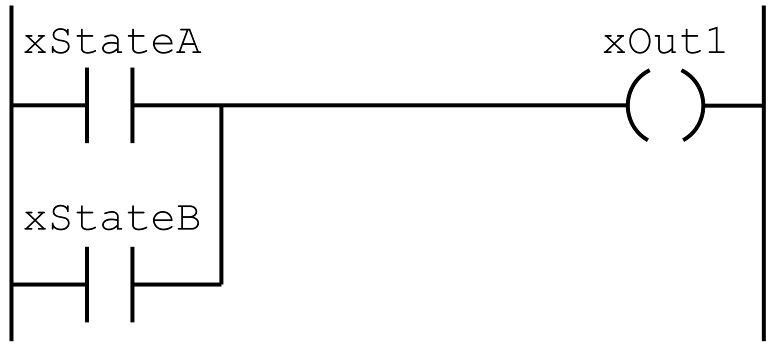
+ initial state

```
   xStateA: BOOL:=TRUE;
END_VAR
```

+ state transitions



+ output mapping



Program for controlling the barrier

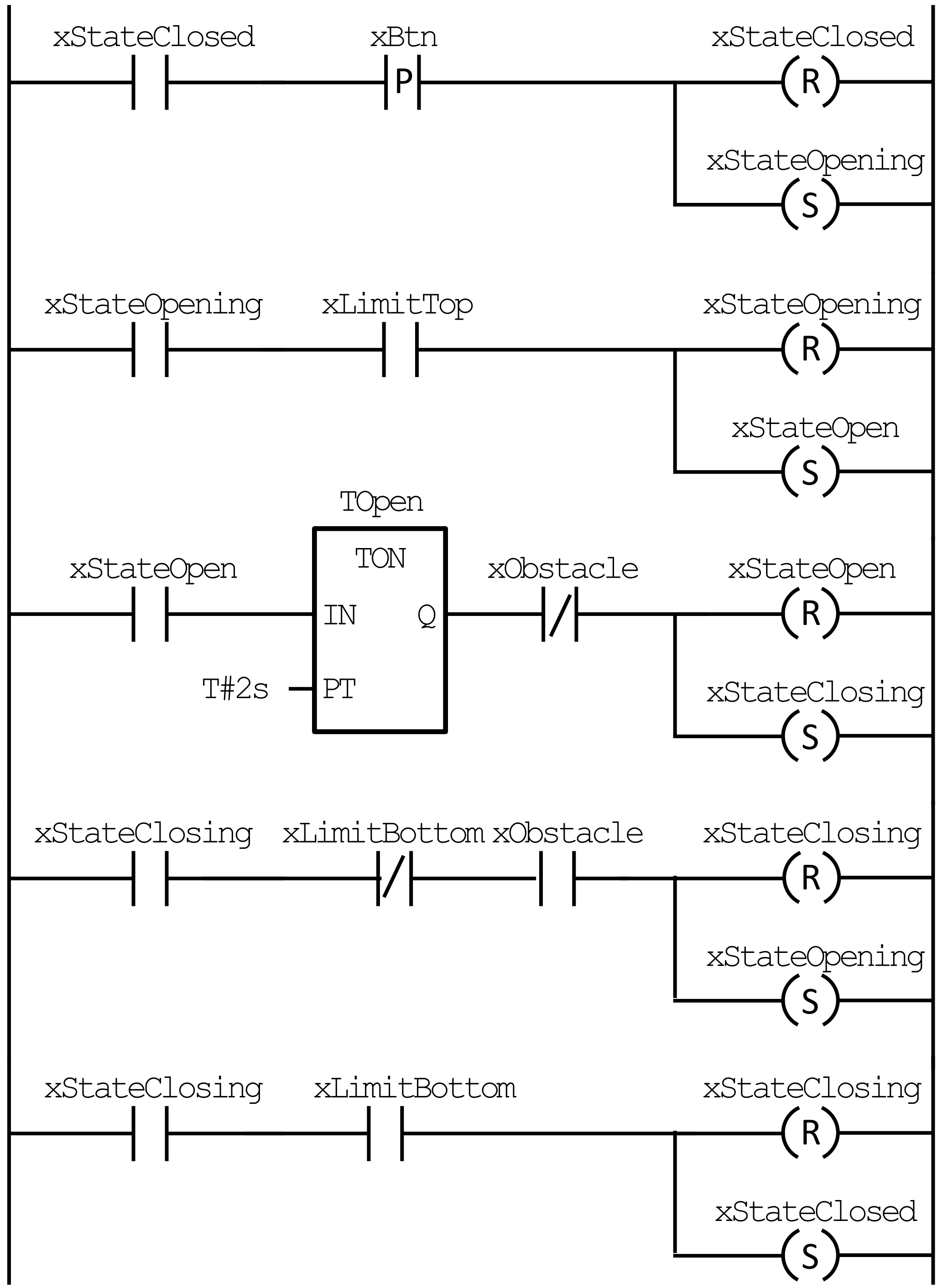
```
PROGRAM BarrierControl
VAR_INPUT
   xLimitBottom AT %IX0.0: BOOL;
   xLimitTop    AT %IX0.1: BOOL;
   xObstacle    AT %IX0.2: BOOL;
   xBtn         AT %IX0.3: BOOL;
END_VAR
VAR_OUTPUT
   xMotor       AT %QX0.0: BOOL;
   xDir         AT %QX0.1: BOOL;
   xRedLight    AT %QX0.2: BOOL;
END_VAR
VAR
   xStateClosed: BOOL := TRUE;
   xStateOpening, xStateOpen, xStateClosing: BOOL;
   TOpen: TON;
END_VAR
```

Initial value of the state variable corresponding to the initial state is `TRUE`

Since no initial value is specified explicitly, other state variables are initialized to the default value of their data type, i.e. `FALSE`

Instance of the timer FB which measures the time spent at state *Open*

State transitions

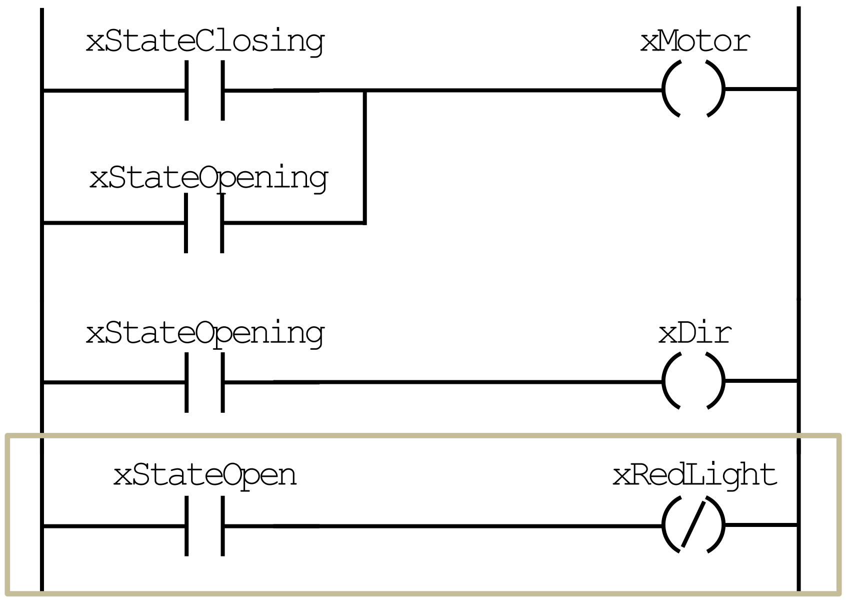


The barrier is only opened upon a rising edge of the pushbutton (protection against weighting down)

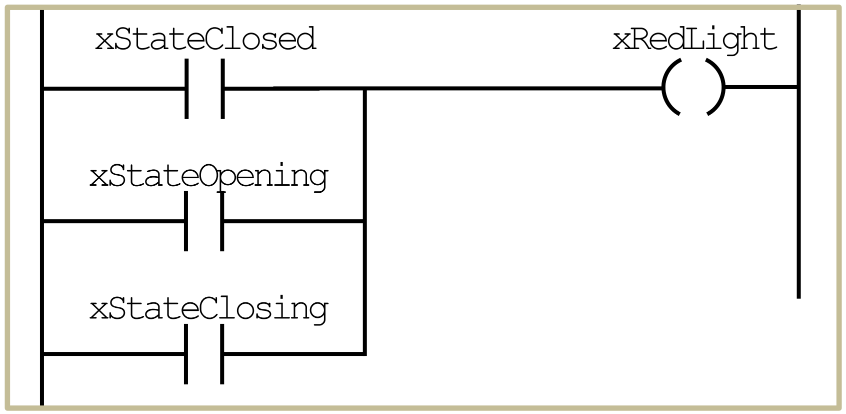
The barrier can be closed if it is open for at least 2 seconds and there is no obstacle present

Mutually exclusive conditions: if an obstacle appears when the barrier reaches its bottom position, it is not reopened (only one of the two transitions can fire)

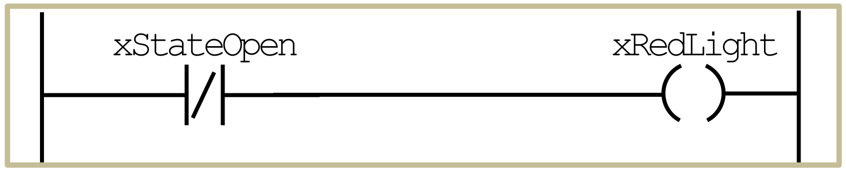
Output mapping



The `RedLight` output (light) is turned off in the *Open* state, i.e. it will be on in every other state



Alternative solution: the `RedLight` output is activated in the *Closed*, *Closing* and *Opening* states (therefore it will be inactive in the *Open* state)

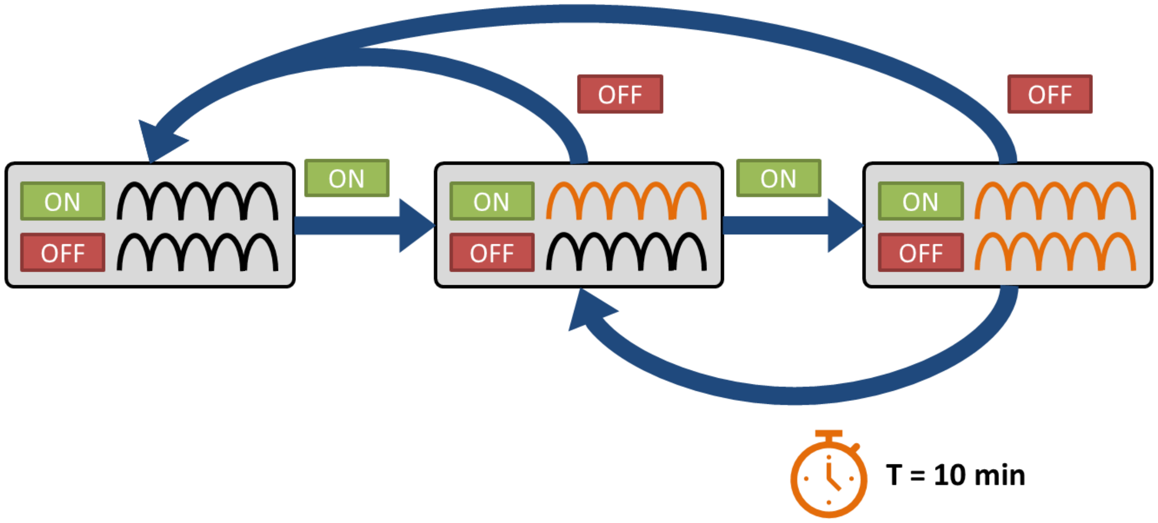


Alternative solution: the `RedLight` output is turned on in every state but the *Open* state (therefore it will be active in the *Open* state)

Implementation in the template project

- Presence of a car activating the obstacle sensor can be turned on and off by the Enter Barrier / Leave Barrier button of the visualization screen
- Behavior of the opening pushbutton is a toggle switch in the visualization (it changes its state when clicked), so protection against weighting down can be tested
- Unused input and output ports of a call can be deleted by right-clicking the block and selecting the *Remove Unused FB Call Parameters* option
- Don’t forget to add the task executing the program!
  - tasks can be added to the *Task Configuration* element of the project tree
  - the program shall be executed by a *Freewheeling* task
  - priority level of the task (*Priority* parameter) shall be set greater than 0

Problem: Heater



<u>PLC inputs</u>

- `xBtnOn` – ON pushbutton
- `xBtnOff` – OFF pushbutton

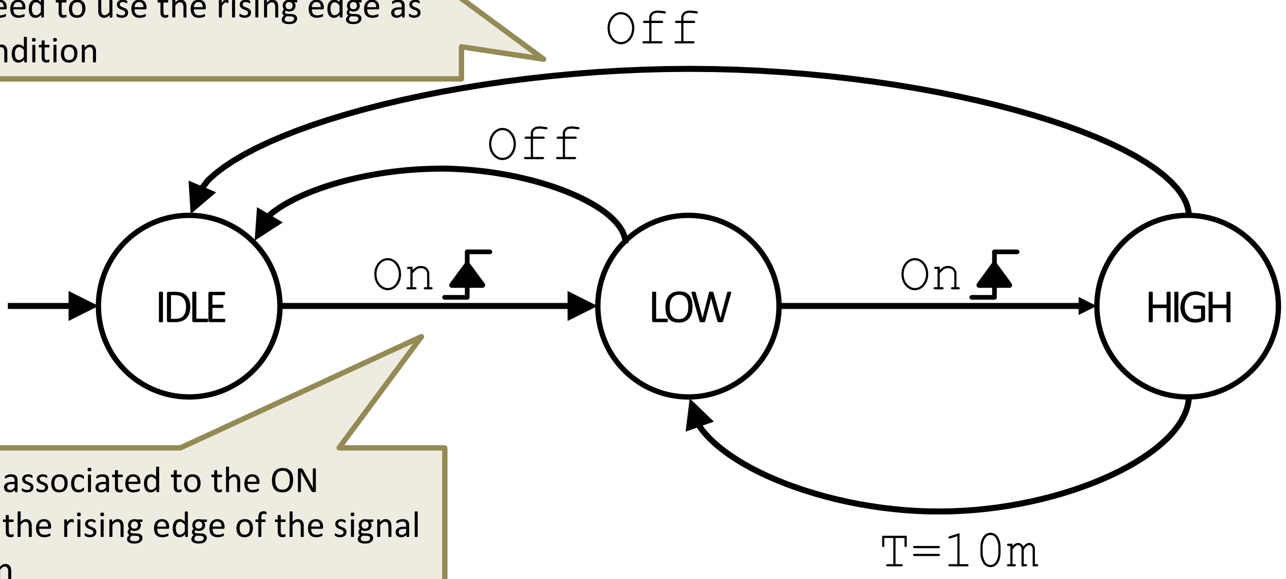
<u>PLC outputs</u>

- `xHeater1` – Heater 1
- `xHeater2` – Heater 2

- When the ON button is pressed, the first heater shall be turned on
- Upon a second press on the ON button, the second heater shall also be turned on. In order to prevent overheating, the second heater shall be turned off after 10 minutes (if the ON button is pressed again, the second heater shall be turned on for another 10 minutes)
- Pressing the OFF pushbutton shall turn both heaters off

State machine of the heater

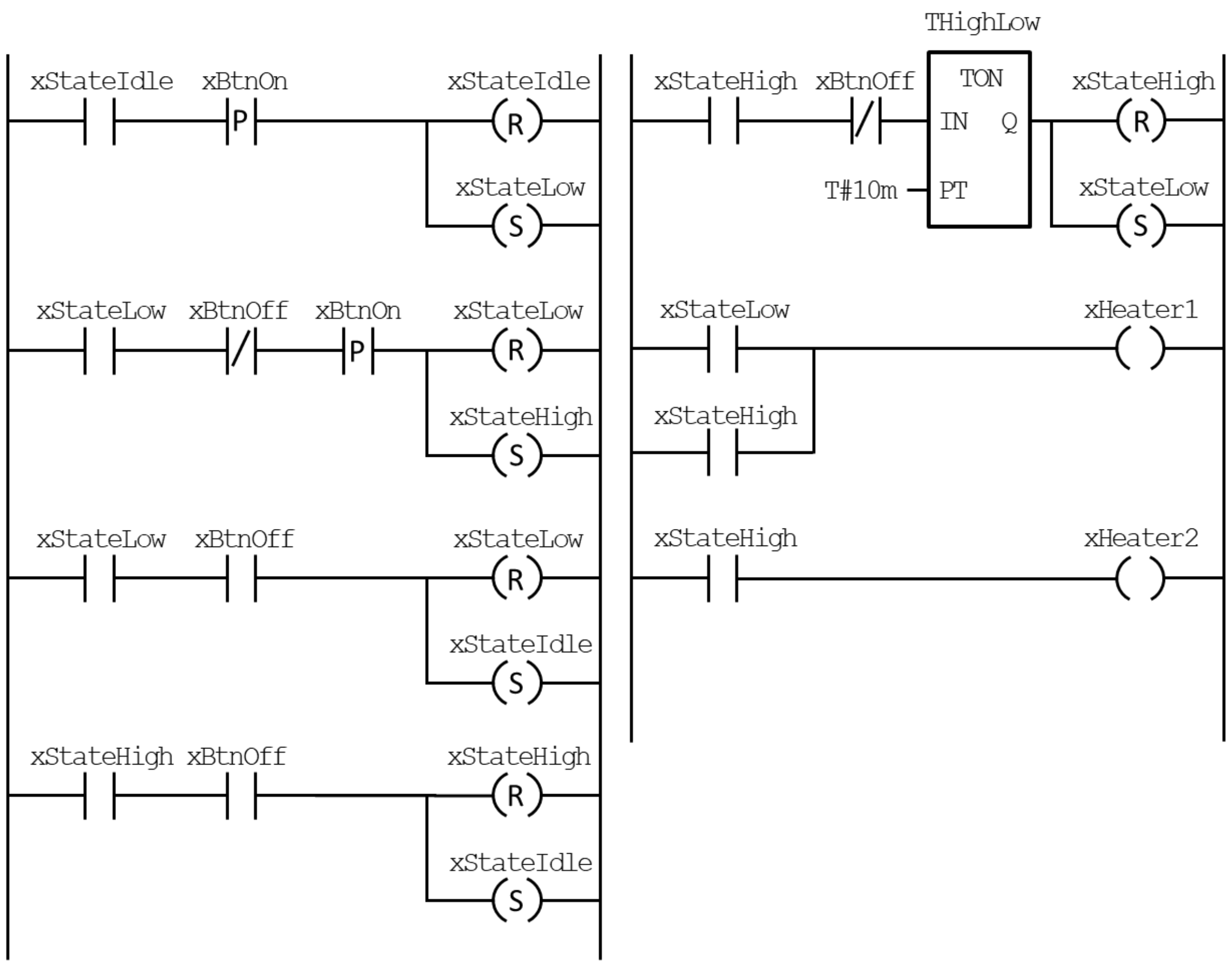
Since there is no transition associated to the OFF button leaving the *Idle* state, there is no need to use the rising edge as transition condition



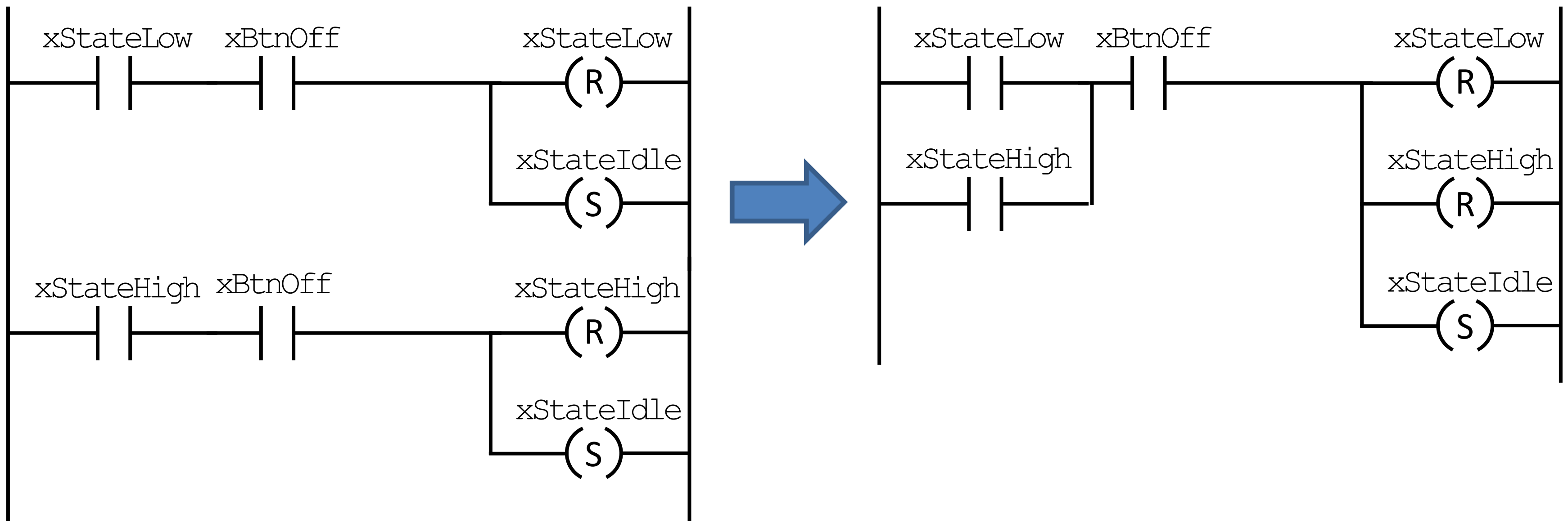
Transitions associated to the ON button use the rising edge of the signal as condition

Program for heater control

```
PROGRAM Heater
VAR
   xStateIdle : BOOL:=TRUE;
   xStateLow  : BOOL;
   xStateHigh : BOOL;
   THighLow   : TON;
END_VAR
VAR_INPUT
   xBtnOn  AT %IX0.0 : BOOL;
   xBtnOff AT %IX0.1 : BOOL;
END_VAR
VAR_OUTPUT
   xHeater1 AT %QX0.0 : BOOL;
   xHeater2 AT %QX0.1 : BOOL;
END_VAR
```



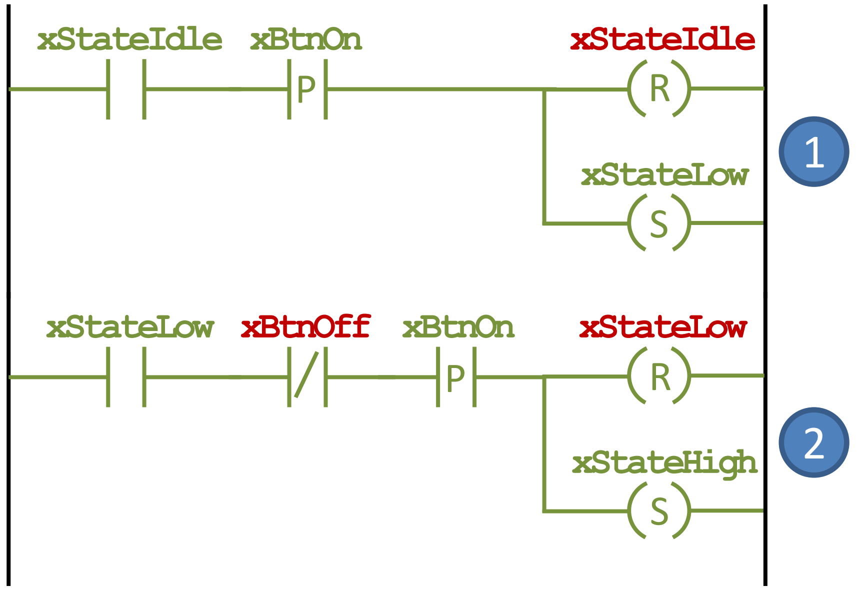
Joining transitions

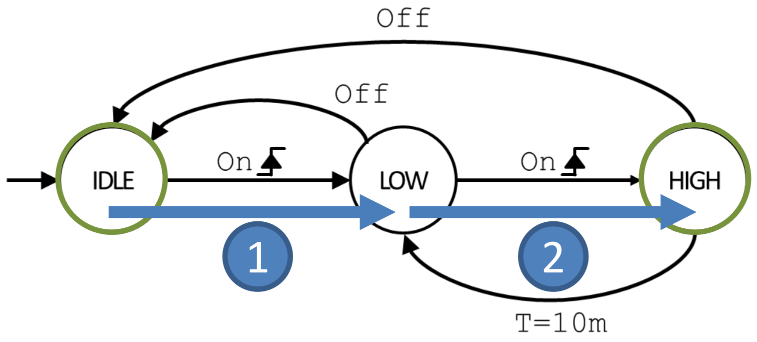


- As both **conditions and destination states** of the two transitions are the same, they can be joined into one single network
- Pro: less networks in the diagram, shorter code
- Con: readability and maintainability of the code is slightly decreased
- Joining transitions is recommended only if there are multiple transition leading to a state with the same condition (common case: error handling)

Rising edge detection

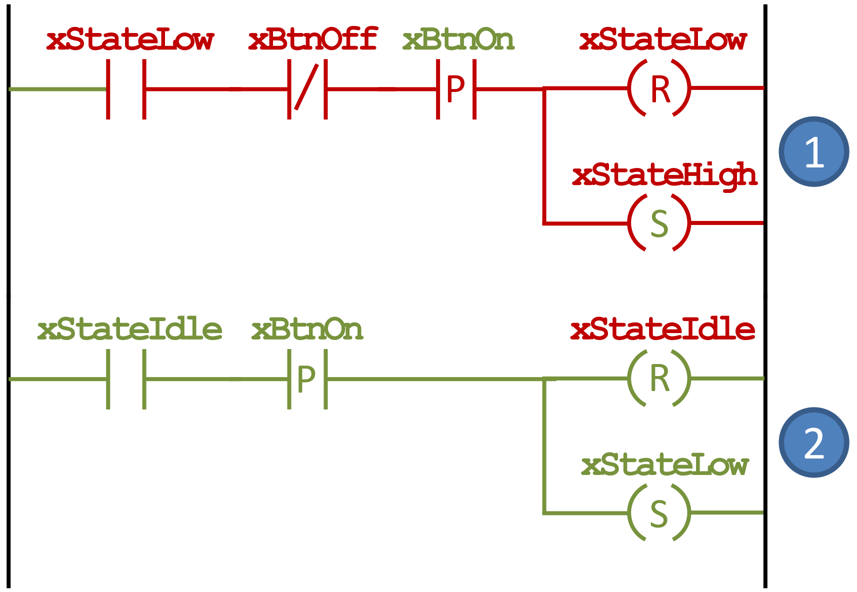
- We use the rising edge of the input `xBtnOn` (associated to edge-sensing contacts) in the conditions of two transitions
- Depending on the development environment, all edge-sensing contacts associated to the same variable might conduct during an evaluation cycle (e.g. CODESYS)
- Both state transitions are executed during a single evaluation: when pressing the button, we reach the state *High* from the state *Idle*

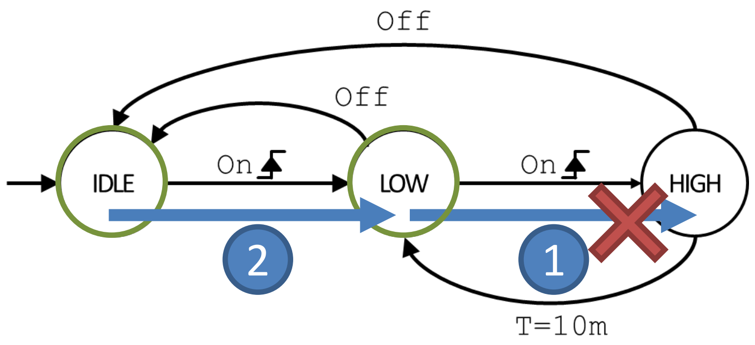




Possible solution

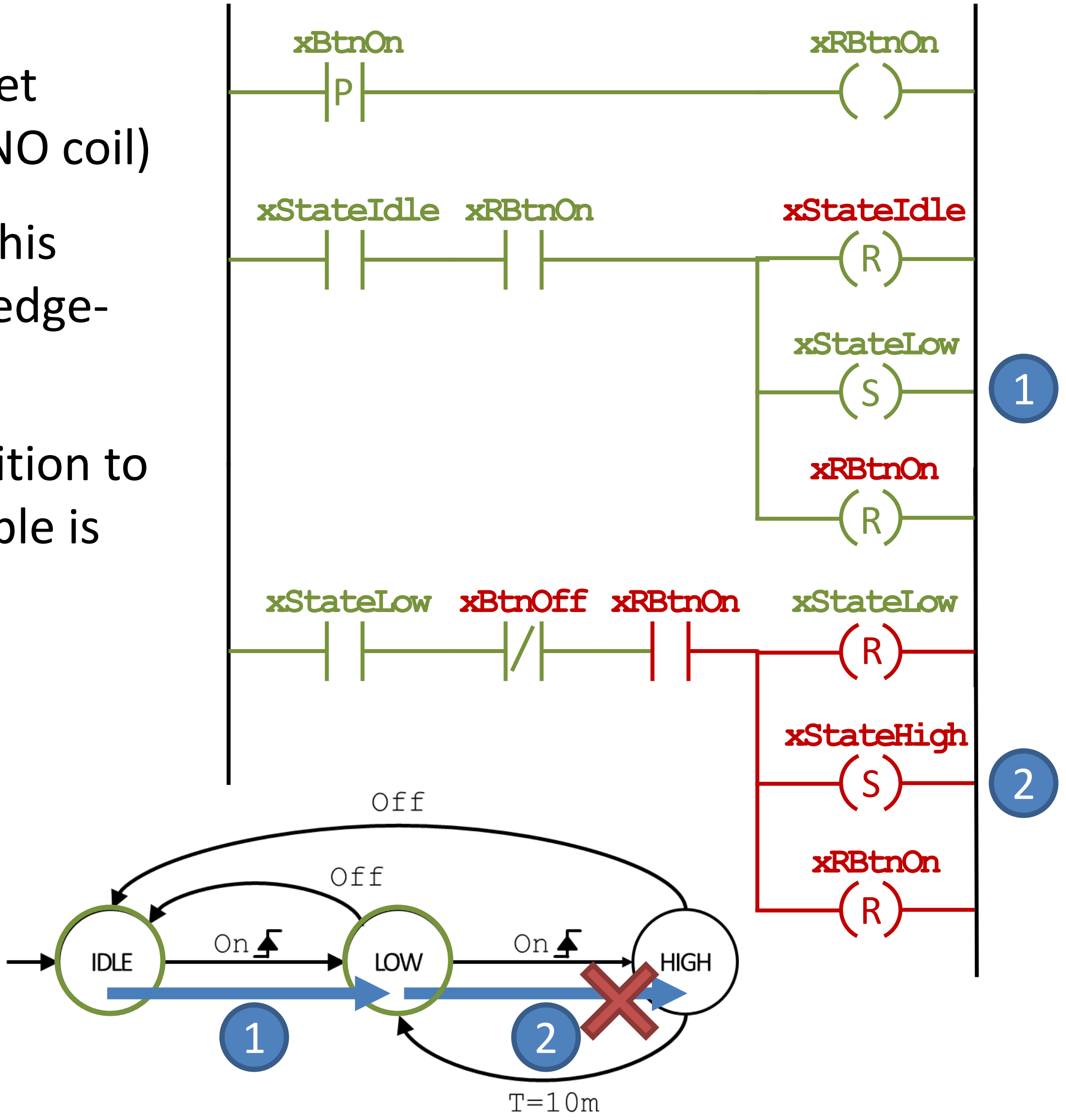
- By changing the order of networks corresponding to the transitions, the *Low* → *High* transition is not executed because its origin state is not active
- During the same evaluation, the *Idle* → *Low* transition is executed because the *Idle* state is active and a rising edge is being detected
- Edge-sensing contacts do not conduct during the next evaluation (cycle), hence the *Low* → *High* transition is executed only upon the next rising edge





Possible solution

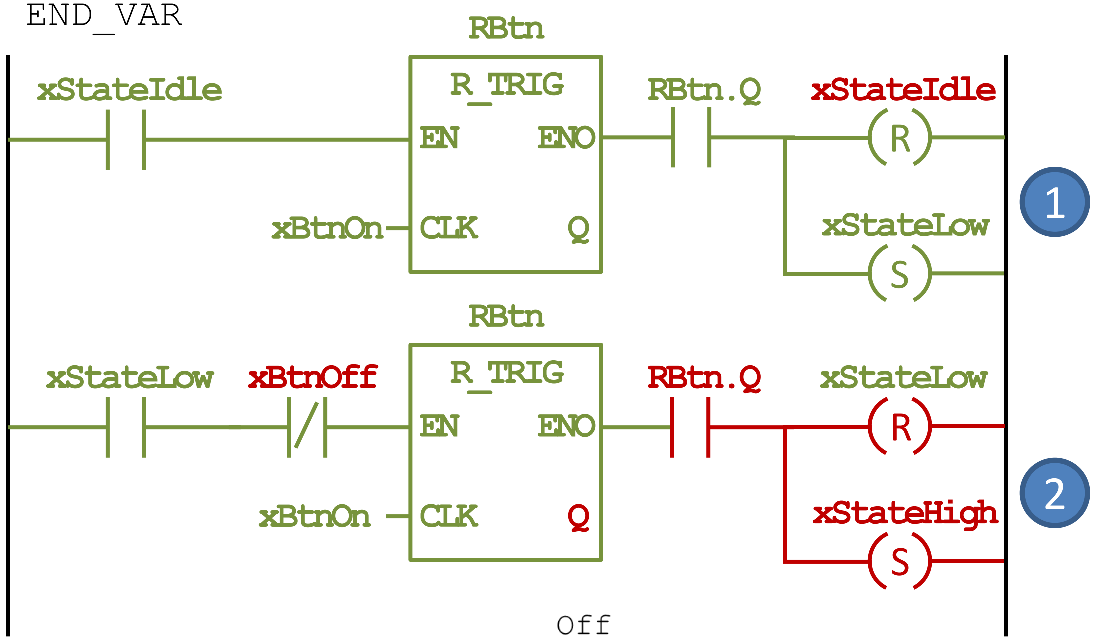
- A Boolean auxiliary variable is set upon the rising edge (using an NO coil)
- Transition conditions shall use this auxiliary variable instead of an edge-sensing contact
- If the rising edge causes a transition to be executed, the auxiliary variable is reset hence conditions of other transitions are not met

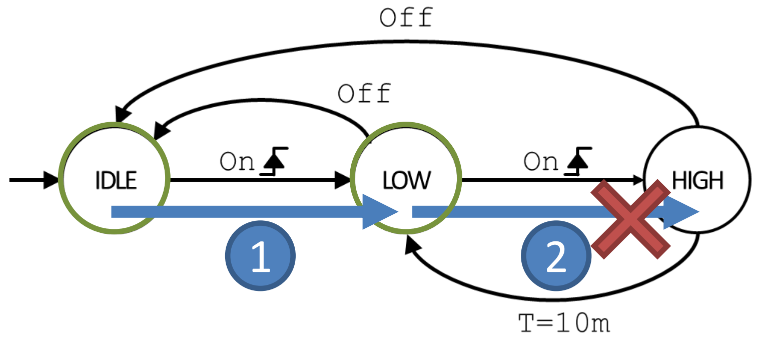


Possible solution

- Instead of an edge-sensing contact, the <u>same</u> edge-sensing function block <u>instance</u> is used in transition conditions
- The FB instance stores the previous value of its `CLK` input in a local variable
- The FB instance sets this local variable upon its first call, so during the further calls the previous value will be the same as the `CLK` input hence the output `Q` will not be set
- FB instances are connected by their input `EN`, so they are being called only if the origin state of the transition is active
- As the output `ENO` is connected to the rung, output `Q` of the FB instances can be evaluated using a further contact

```
VAR
   RBtn: R_TRIG;
END_VAR
```





Implementation in the template project

- Behavior of the pushbuttons is toggle switch in the visualization, so simultaneous press of buttons can also be tested
- It is recommended to use a timing value less than 10 minutes (e.g. 5 seconds)
- Unused input and output ports of a call can be deleted by right-clicking the block and selecting the *Remove Unused FB Call Parameters* option
- Don’t forget to add the task executing the program!
  - tasks can be added to the *Task Configuration* element of the project tree
  - the program shall be executed by a *Freewheeling* task
  - priority level of the task (*Priority* parameter) shall be set greater than 0

CODESYS