Robots and People

Vlad Estivill-Castro

Correctness by Construction with Logic-Labeled Finite-State Machines – Comparison with Event-B

Robots and people

Thanks for your interest

Vlad Estivill-Castro

School of Information and Communication Technology
Institute for Intelligent and Integrated Systems
Griffith University
Australia

Rene Hexel

School of Information and Communication Technology
Institute for Intelligent and Integrated Systems
Griffith University
Australia

Outline

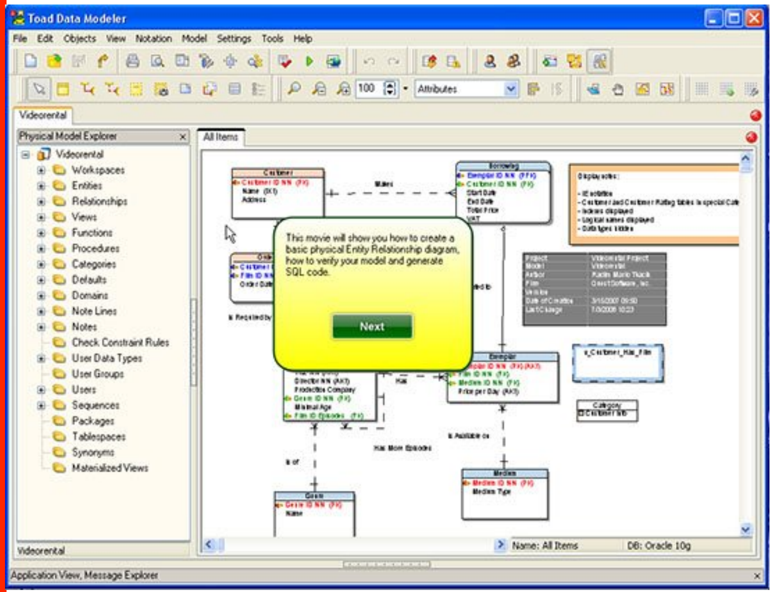
- Motivation
  - Model-Driven Development (MDD) vs formal methods
    - Event-B
    - Logic-labelled Finite State Machines
- Case Studies
  - Bridge – controller
  - Car-window Controller
- Conclusions

Model-Driven Development (MDD)

- Widely successful approach to developing software
- Ensures traceability, validation against requirements, and platform independence
- Tools and techniques are resulting in faster and simpler (easier to maintain) products and applications than traditional language parser/compiler or interpreter approaches

MDD illustration (static modelling)

- Data/Class Modelling tools that generate code
  - (ER diagrams to SQL CREATE)
  - (Class Diagrams to Java class templates)



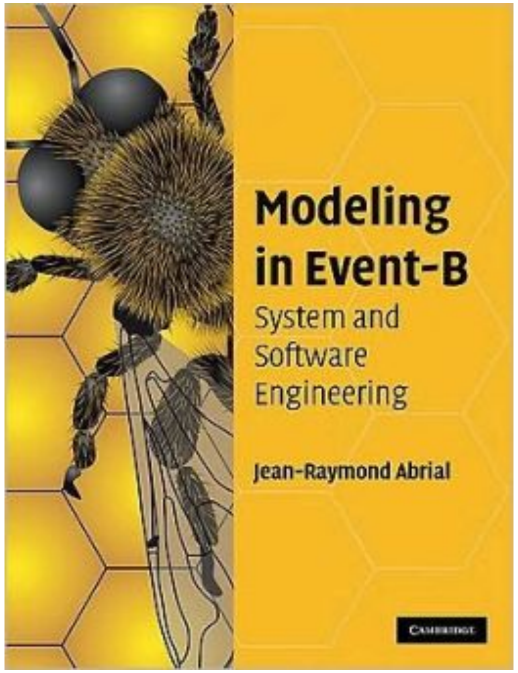
This movie will show you how to create a basic physical Entity Relationship diagram, how to verify your model and generate SQL code.

[4][5]:The two most used (100% of the time) UML constructs are

- class diagrams
- state-charts

What is Event-B?

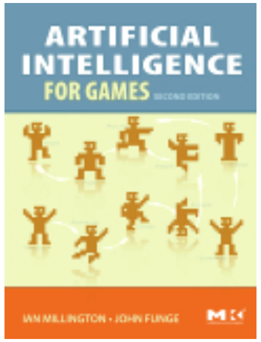
- The B method is method of software development based on B,
    - a tool-supported formal method based around an abstract machine notation, used in the development of computer software.

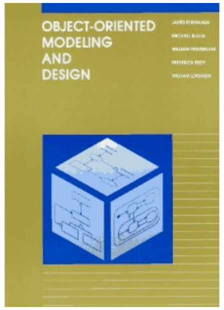

Modeling in Event-B
System and Software Engineering
Jean-Raymond Abrial

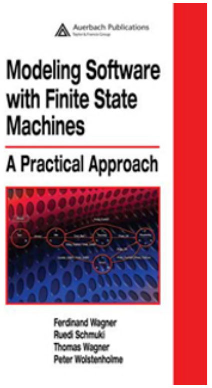
Robots and people

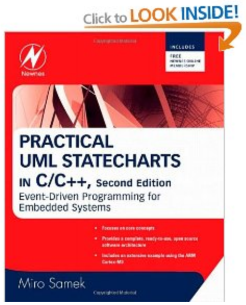
Finite-State Machines (FSM)

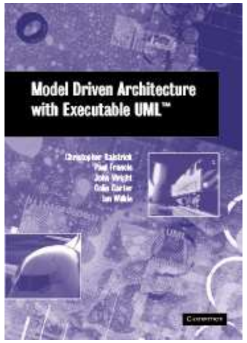
- Widely used model of behavior in embedded systems
  - *QP* (Samek, 2008), *Bot- Studio* (Michel, 2004) *StateWORKS* (Wagner et al., 2006) and *MathWorks*® *StateFlow*. The UML form of FSMs derives from OMT (Rumbaugh et al., 1991, Chapter 5), and the MDD initiatives of Executable UML (Mellor and Balcer, 2002)

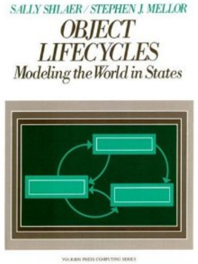










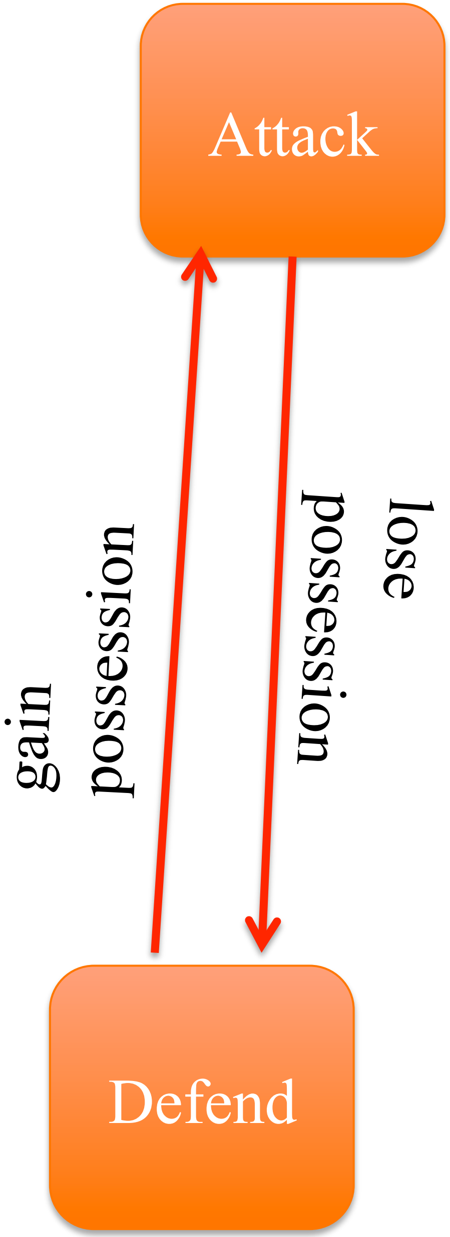


- The original Subsumption Architecture was implemented using the Subsumption Language
- It was based on finite state machines (FSMs) augmented with timers (AFSMs)
- AFSMs were implemented in Lisp

Event-driven FSMs

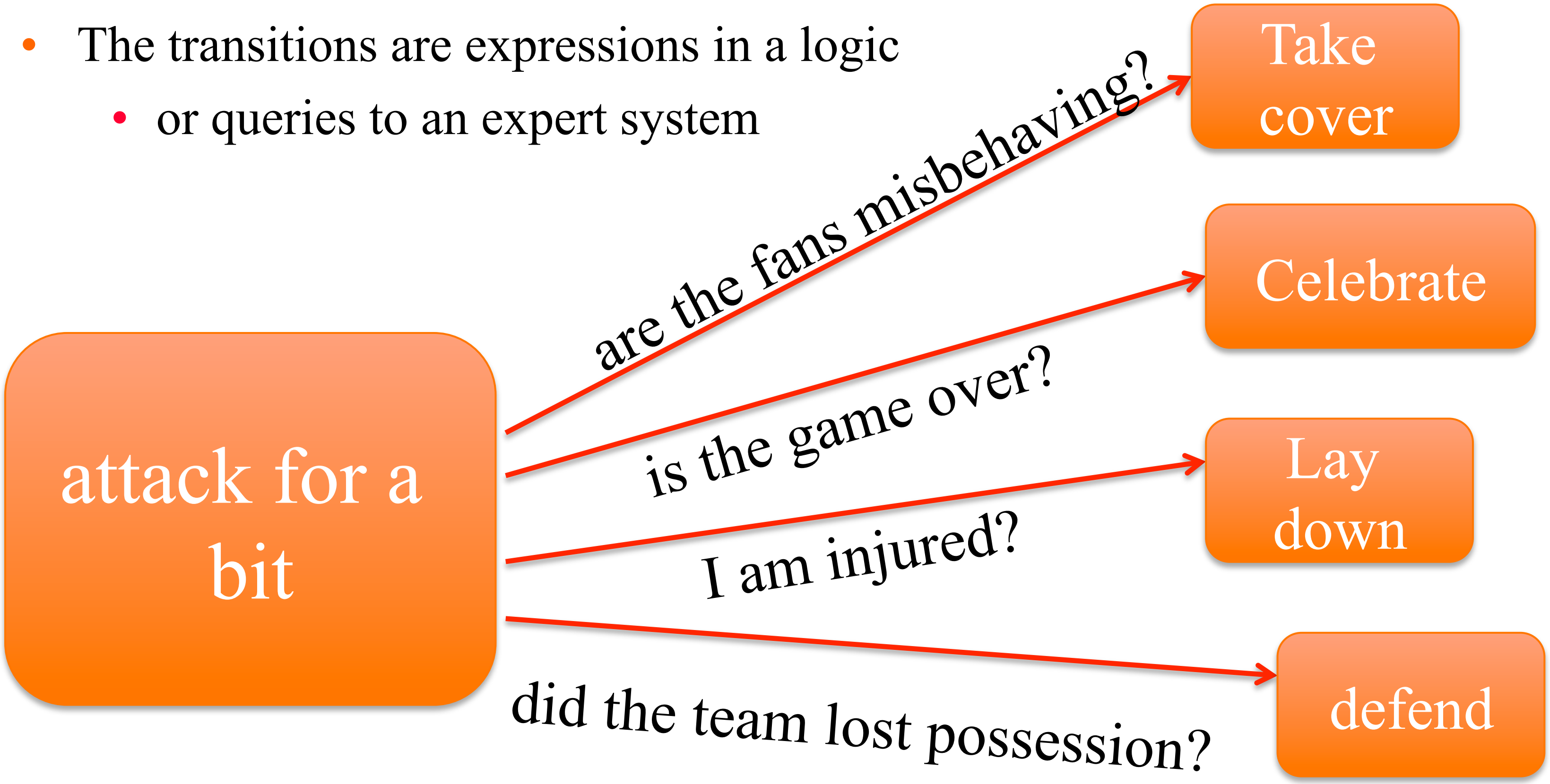
Most common approach

- System is in a state
    - waiting
    - does not change what is
        - doing/happening
    - until event arrives
- Events change the state of the system



Logic-labeled FSMs

- A second view of time (since Harel’s seminal paper)
  - Machines are not waiting in the state for events
  - The machines drive, execute
  - The transitions are expressions in a logic
    - or queries to an expert system



attack for a bit

are the fans misbehaving?

Take cover

is the game over?

Celebrate

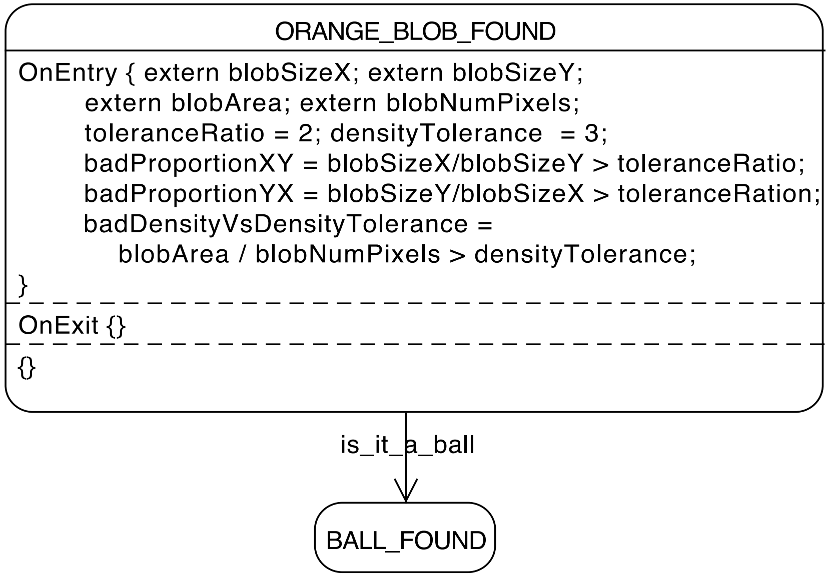
I am injured?

Lay down

did the team lost possession?

defend

Example from robotic soccer



ORANGE_BLOB_FOUND

OnEntry { extern blobSizeX; extern blobSizeY;
extern blobArea; extern blobNumPixels;
toleranceRatio = 2; densityTolerance = 3;
badProportionXY = blobSizeX/blobSizeY > toleranceRatio;
badProportionYX = blobSizeY/blobSizeX > toleranceRation;
badDensityVsDensityTolerance =
blobArea / blobNumPixels > densityTolerance;
}

OnExit {}

{}

```
% BallConditions.d

name{BALLCONDITIONS}.

input{badProportionXY}.
input{badProportionYX}.
input{badDensityVsDensityTolerance}.

BC0: {}              => is_it_a_ball.
BC1: badProportionXY  =>  ~is_it_a_ball. BC1 > BC0.
BC2: badProportionYX  =>  ~is_it_a_ball. BC2 > BC0.
BC3: badDensityVsDensityTolerance  =>  ~is_it_a_ball. BC3 > BC0.

output{b is_it_a_ball, "is_it_a_ball"}.
```

Logic labeled FSMs provide deliverative contro

Arrangements of LLFSM

- Enable MDD
- Provide sequential execution
- Avoid concurrency challenges
- Can be formally verified
- Can be simulated (validated)
- So far, compared directly with
  - Behaviour Trees, Petri nets, Executable UML
- Can perform FMEA
- BUT, how do they compare with UML-B?

One Minute Microwave

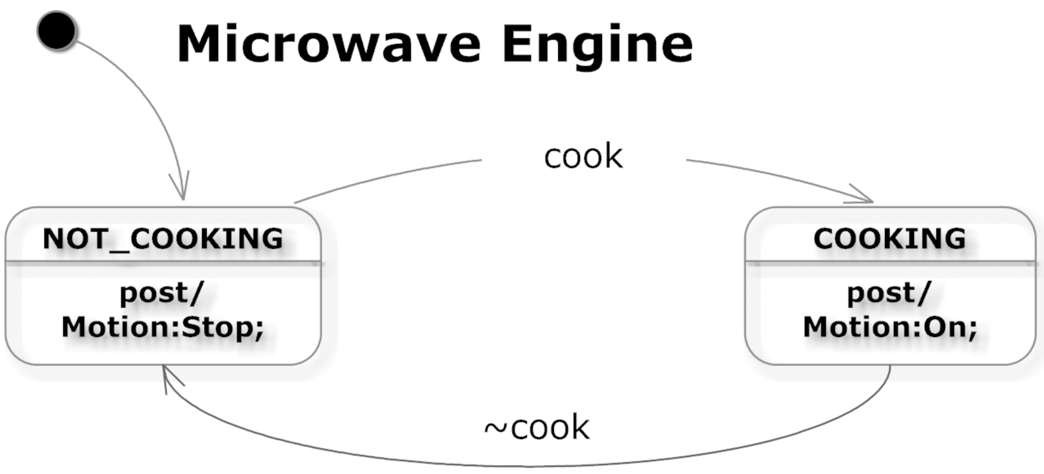
- Widely discussed in the literature of software engineering
- Analogous to the X-Ray machine
  - Therac-25 radiation machine that caused harm to patients
- Important SAFETY feature
  - OPENING THE DOOR SHALL STOP THE COOKING

Requirements

Requirements	Description
R1	There is a single control button available for the use of the oven. If the oven is closed and you push the button, the oven will start cooking (that is, energize the power-tube) for one minute
R2	If the button is pushed while the oven is cooking, it will cause the oven to cook for an extra minute.
R3	Pushing the button when the door is open has no effect.
R4	Whenever the oven is cooking or the door is open, the light in the oven will be on.
R5	Opening the door stops the cooking. and stops the timer and does not clear the timer
R6	Closing the door turns off the light. This is the normal idle state, prior to cooking when the user has placed food in the oven.
R7	If the oven times out, the light and the power-tube are turned off and then a beeper emits a warning beep to indicate that the cooking has finished.

Logic-labeled Finite-state machines with DPL

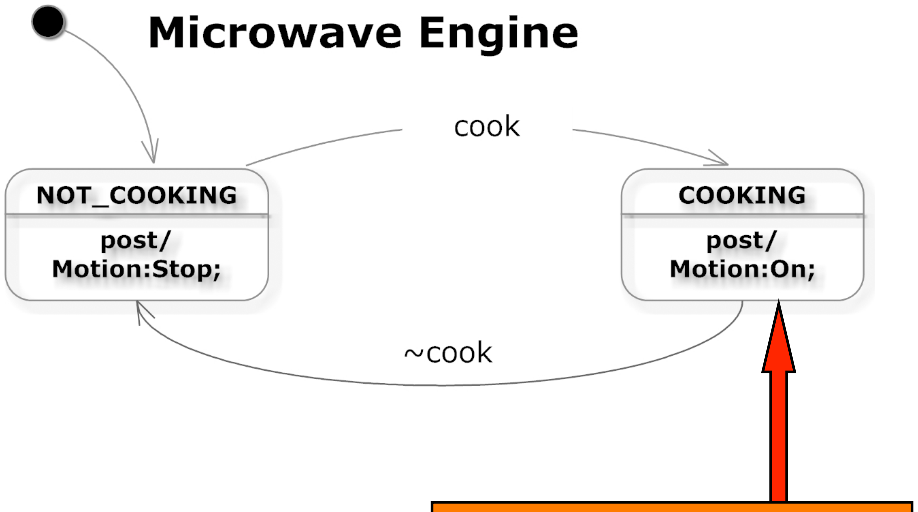
- **Step 1:** Consider writing the script of music for an orchestra. Write individual scripts and place together all actuators that behave with the same actions for the same cues
- Example: The control of the tube (energizing), the fan and the spinning plate



Robots and people

Step 2: Describe the conditions that result in the need to change state

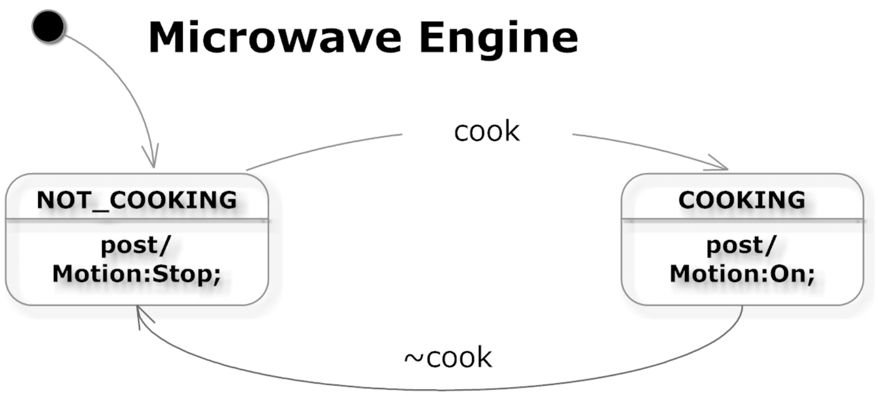
```
% MicrowaveCook.d

name{MicrowaveCook}.

input{timeLeft}.
input{doorOpen}.

C0: {}        => ~cook.
C1: timeLeft =>  cook. C1 > C0.
C2: doorOpen => ~cook. C2 > C1.

output{b cook, "cook"}.
```



Action:
Posting a message to the whiteboard

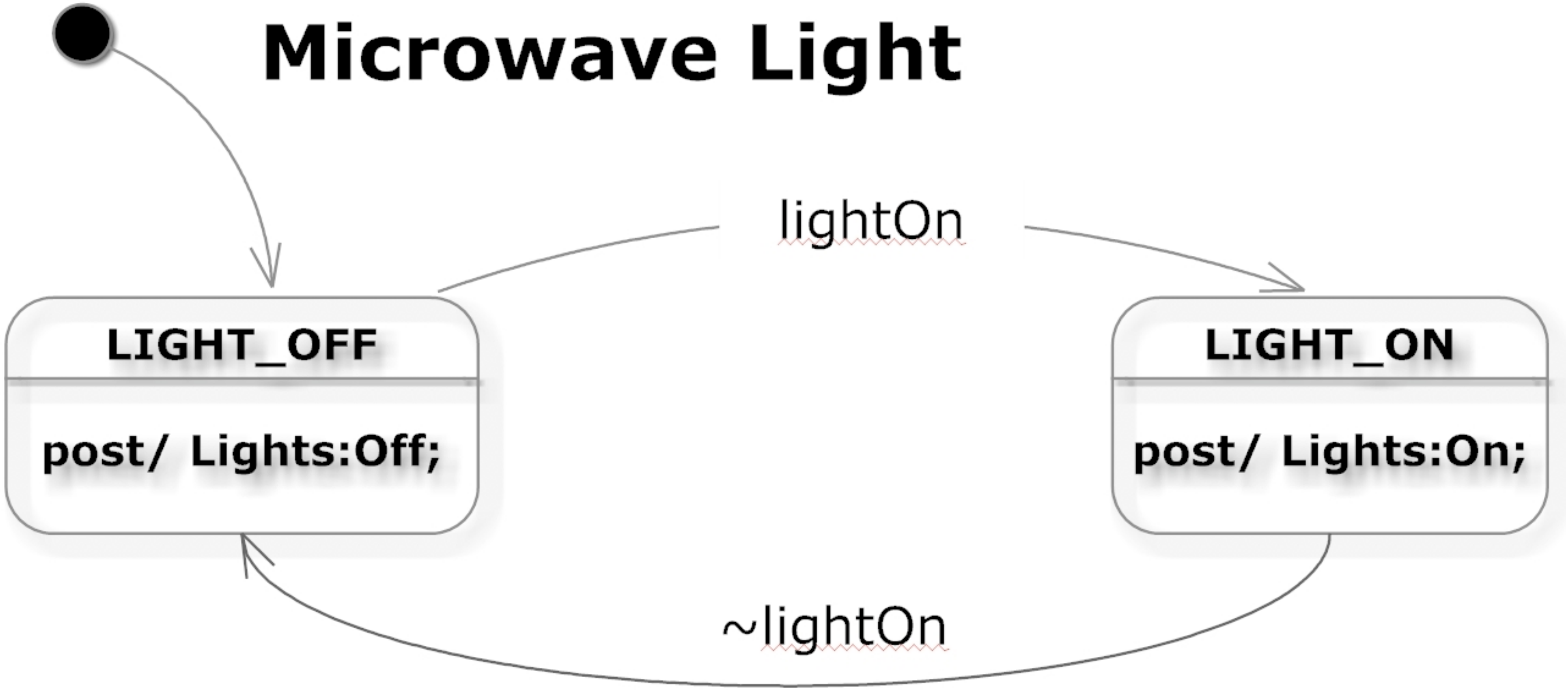
One of the LLFSMs

```
% MicrowaveCook.d

name{MicrowaveCook}.

input{timeLeft}.
input{doorOpen}.

C0: {}       => ~cook.
C1: timeLeft =>  cook. C1 > C0.
C2: doorOpen => ~cook. C2 > C1.

output{b cook, "cook"}.
```



Step 1 (again): Analyze another actuator

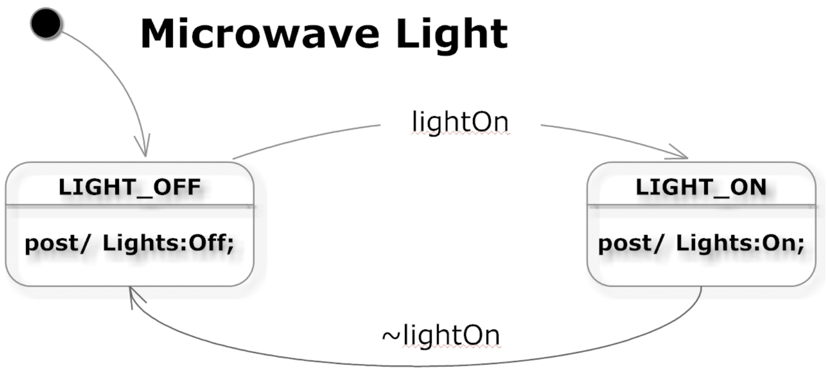
- Illustration: The light



Microwave Light

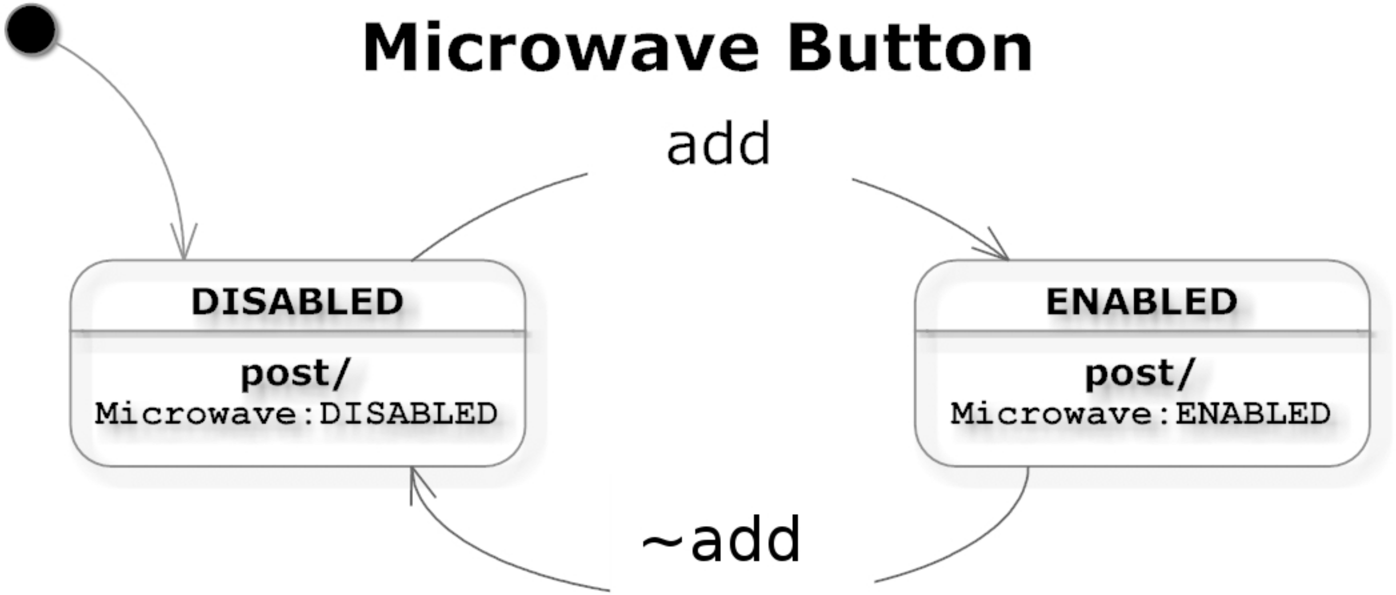
Step 2 (again): Describe the conditions that result in the need to change state

```
% MicrowaveLight.d

name{MicrowaveLight}.

input{timeLeft}.
input{doorOpen}.

L0: {}        => ~lightOn.
L1: timeLeft =>  lightOn. L1 > L0.
L2: doorOpen =>  lightOn. L2 > L0.

output{b lightOn, "lightOn"}.
```

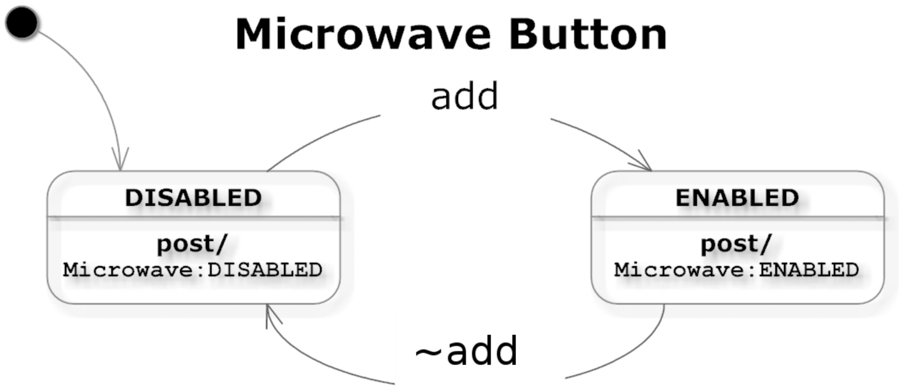


Step 1 (again): Analyze another actuator

- Illustration: The button



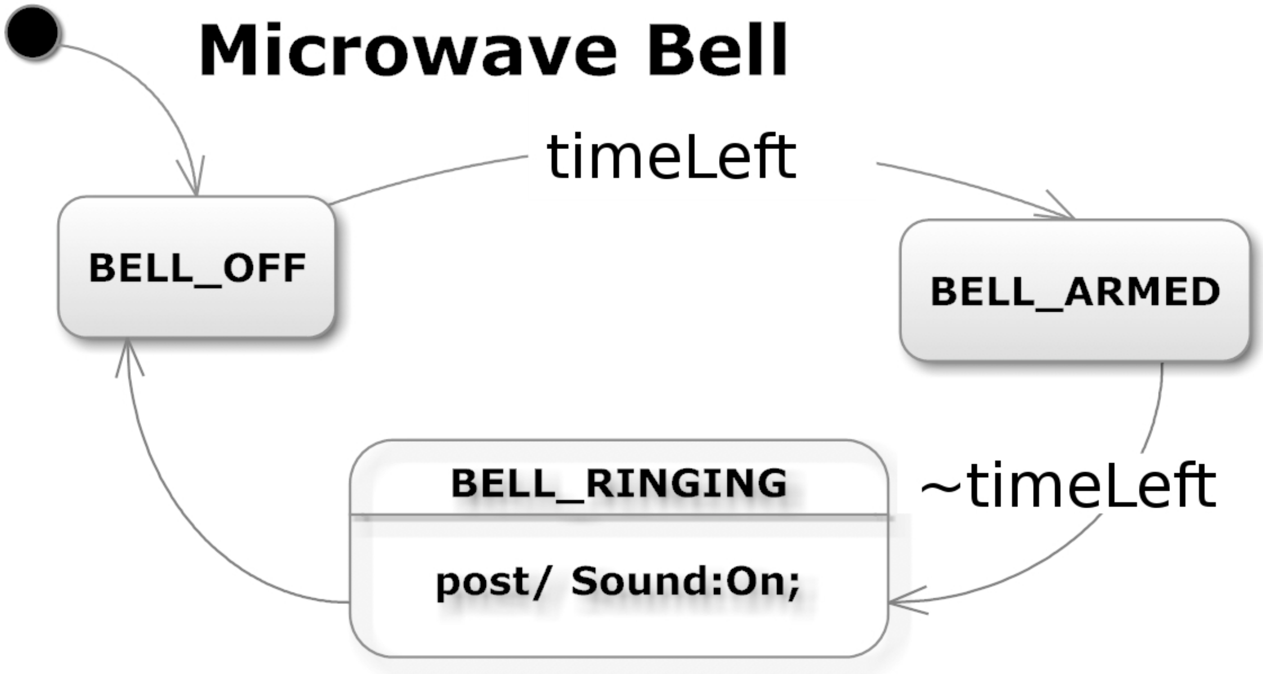
Step 2 (again): Describe the conditions that result in the need to change state



```
% MicrowaveButton.d

name{MicrowaveButton}.

input{doorOpen}.
input{buttonPushed}.


CB0: {}            => ~add.
CB1: buttonPushed =>  add. CB1 > CB0.
CB2: doorOpen      => ~add. CB2 > CB1.


output{b add, "add"}.
```

Step 1 (again): Analyze another actuator

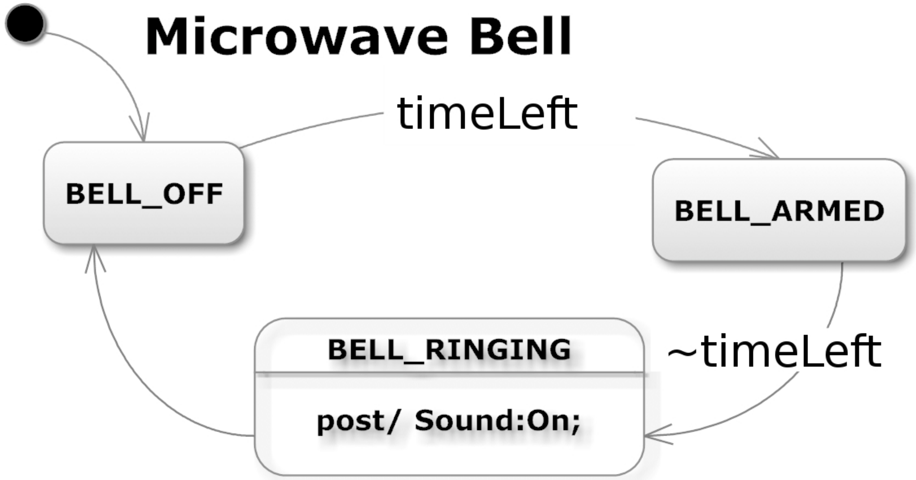
- Illustration: The bell



Step 2 (again): Describe the conditions that result in the need to change state

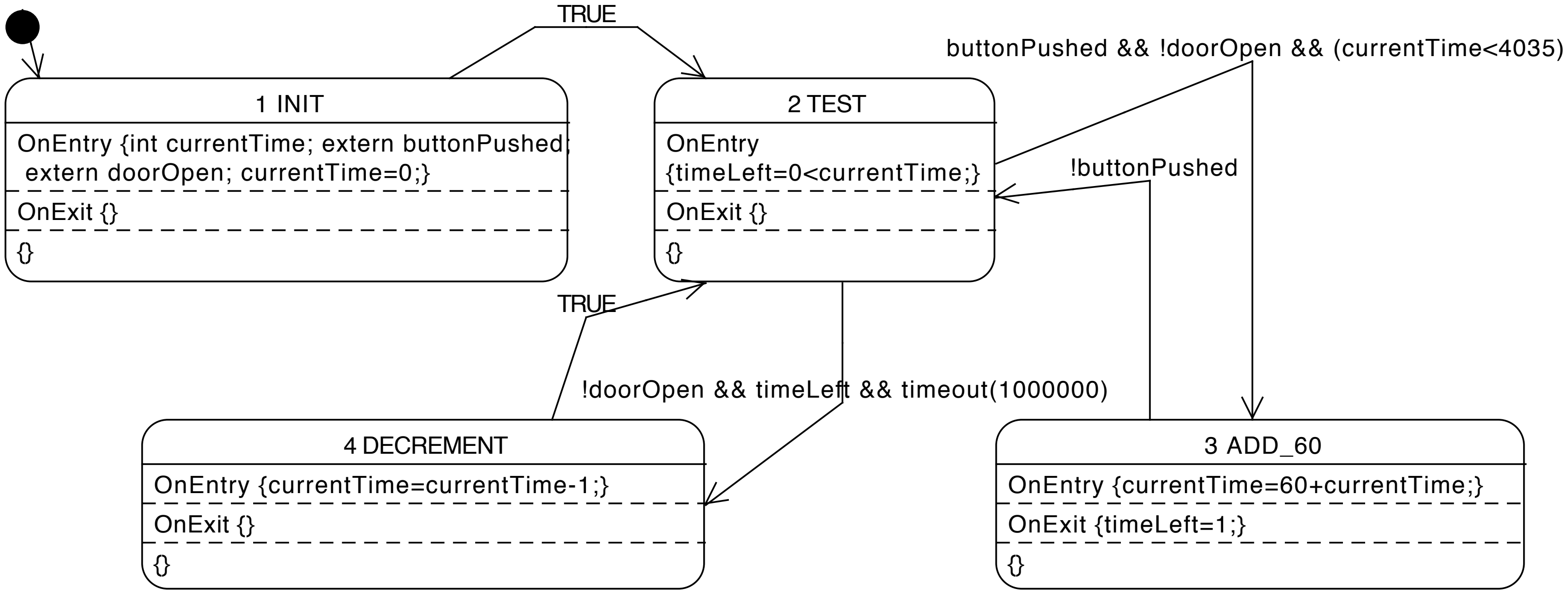
No need for a logic: `timeLeft`

- posted by another module
- does not require a proof



Step 1 (again): Analyze another actuator

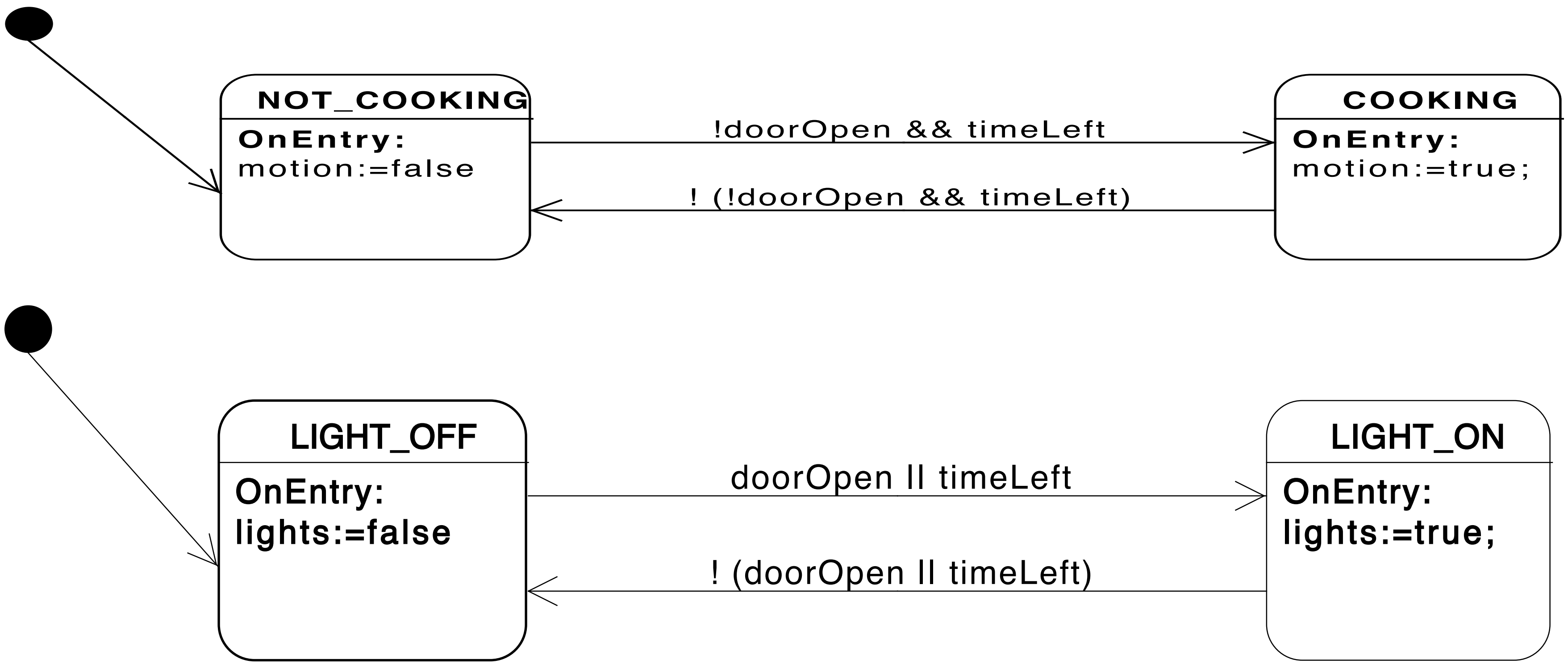
- Illustration: The timer



Embedded systems are performing several things

- The models is made of several finite state-machines
  - Behavior-based control
- With a rich language of logic, the modeling aspect is decomposed
  - the action /reaction part of the system
    - the states and transitions of the finite-state machine
  - the declarative knowledge of the world
    - the logic system

The Microwave example --- We can translate DPL to propositions



NOT_COOKING
OnEntry:
motion:=false

!doorOpen && timeLeft

! (!doorOpen && timeLeft)

COOKING
OnEntry:
motion:=true;

LIGHT_OFF
OnEntry:
lights:=false

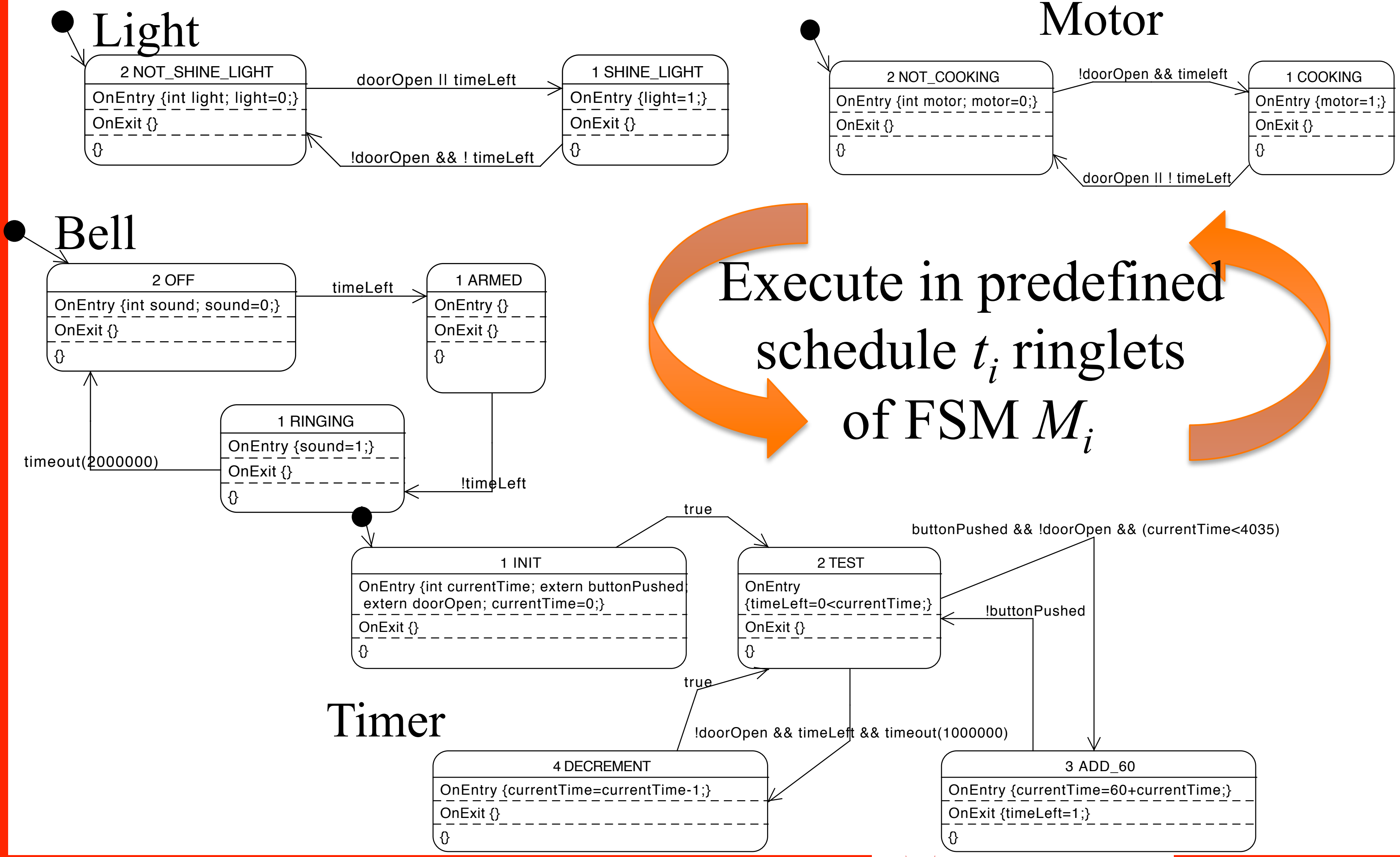
doorOpen || timeLeft

! (doorOpen || timeLeft)

LIGHT_ON
OnEntry:
lights:=true;

Robots and people

The complete arrangement



Light

2 NOT_SHINE_LIGHT
OnEntry {int light; light=0;}
OnExit {}
{}

doorOpen II timeLeft

!doorOpen && ! timeLeft

1 SHINE_LIGHT
OnEntry {light=1;}
OnExit {}
{}

Motor

2 NOT_COOKING
OnEntry {int motor; motor=0;}
OnExit {}
{}

!doorOpen && timeleft

doorOpen II ! timeLeft

1 COOKING
OnEntry {motor=1;}
OnExit {}
{}

Bell

2 OFF
OnEntry {int sound; sound=0;}
OnExit {}
{}

timeLeft

1 ARMED
OnEntry {}
OnExit {}
{}

!timeLeft

1 RINGING
OnEntry {sound=1;}
OnExit {}
{}

timeout(2000000)

Execute in predefined schedule t_i ringlets of FSM M_i

Timer

1 INIT
OnEntry {int currentTime; extern buttonPushed; extern doorOpen; currentTime=0;}
OnExit {}
{}

true

2 TEST
OnEntry {timeLeft=0<currentTime;}
OnExit {}
{}

buttonPushed && !doorOpen && (currentTime<4035)

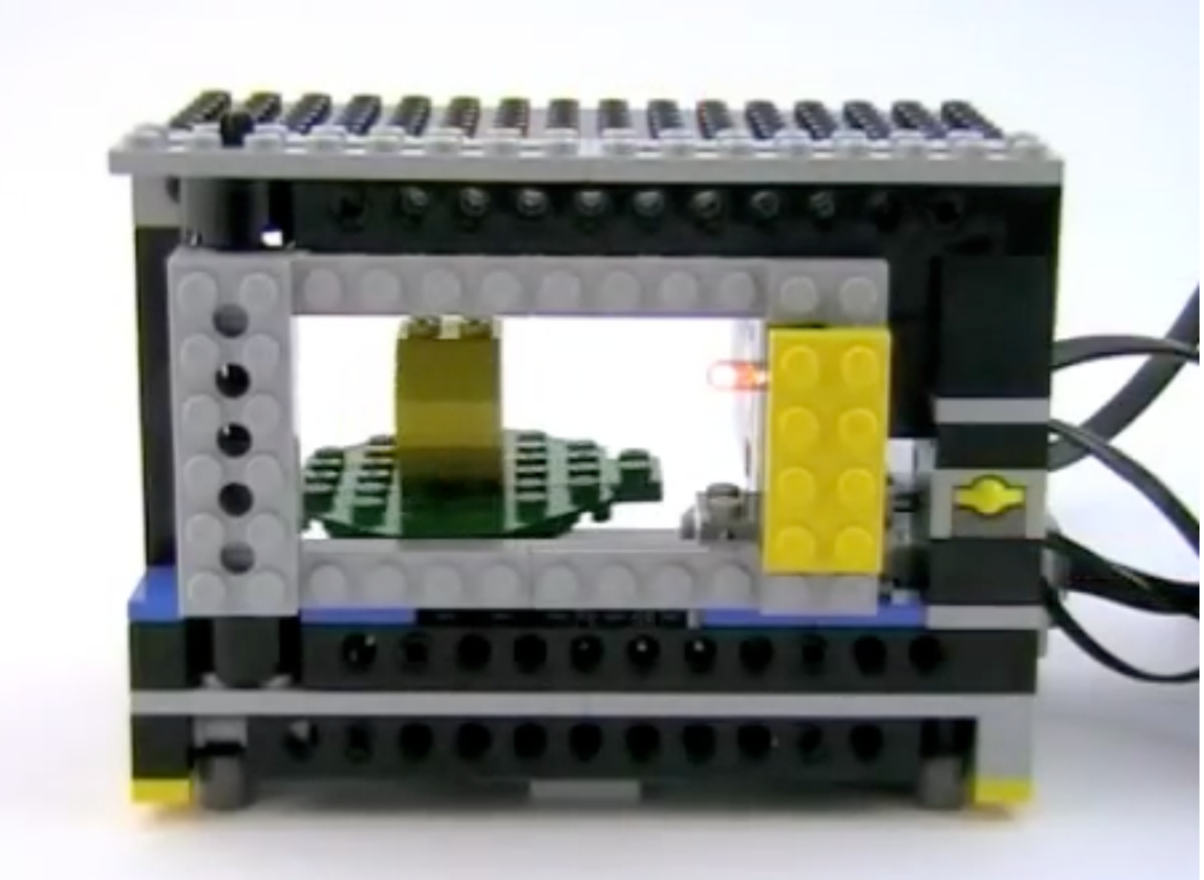
!buttonPushed

true

!doorOpen && timeLeft && timeout(1000000)

4 DECREMENT
OnEntry {currentTime=currentTime-1;}
OnExit {}
{}

3 ADD_60
OnEntry {currentTime=60+currentTime;}
OnExit {timeLeft=1;}
{}

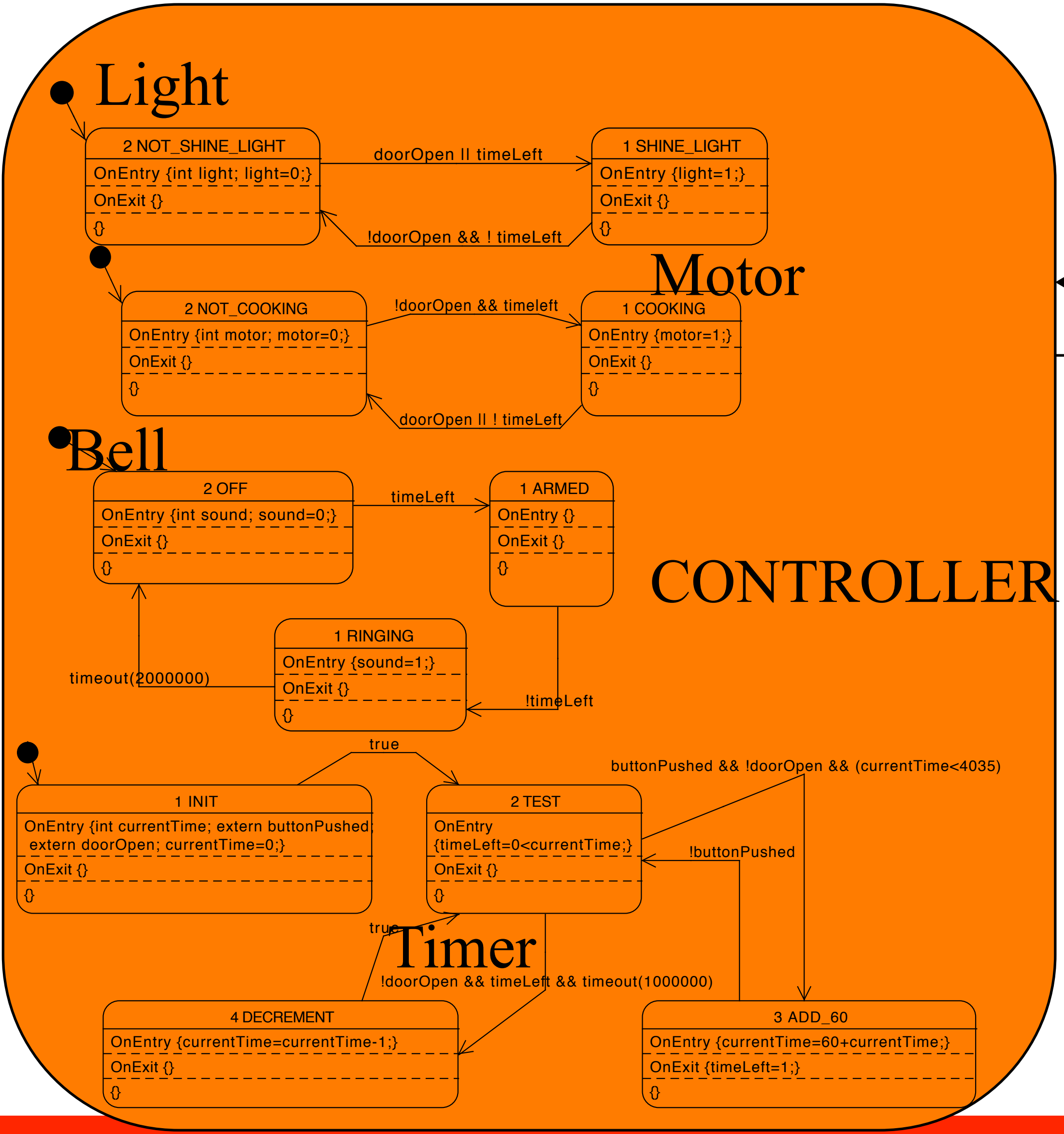


Demo video (`java`)

- http://www.youtube.com/watch?v=t4ueI1o67Xk&feature=relmfu

Robots and people

SIMULATION



Light

2 NOT_SHINE_LIGHT
OnEntry {int light; light=0;}
OnExit {}
{}

doorOpen || timeLeft

!doorOpen && ! timeLeft

1 SHINE_LIGHT
OnEntry {light=1;}
OnExit {}
{}

Motor

2 NOT_COOKING
OnEntry {int motor; motor=0;}
OnExit {}
{}

!doorOpen && timeleft

doorOpen || ! timeLeft

1 COOKING
OnEntry {motor=1;}
OnExit {}
{}

Bell

2 OFF
OnEntry {int sound; sound=0;}
OnExit {}
{}

timeLeft

1 ARMED
OnEntry {}
OnExit {}
{}

1 RINGING
OnEntry {sound=1;}
OnExit {}
{}

timeout(2000000)

!timeLeft

CONTROLLER

1 INIT
OnEntry {int currentTime; extern buttonPushed;
extern doorOpen; currentTime=0;}
OnExit {}
{}

true

2 TEST
OnEntry
{timeLeft=0<currentTime;}
OnExit {}
{}

buttonPushed && !doorOpen && (currentTime<4035)

!buttonPushed

true

Timer

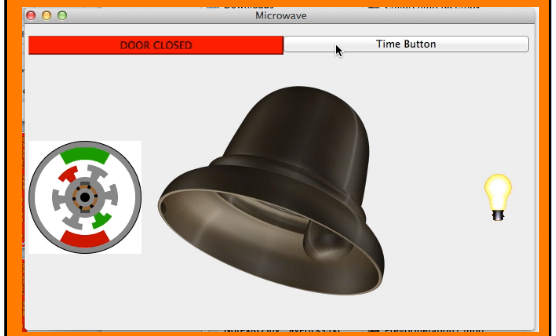
!doorOpen && timeLeft && timeout(1000000)

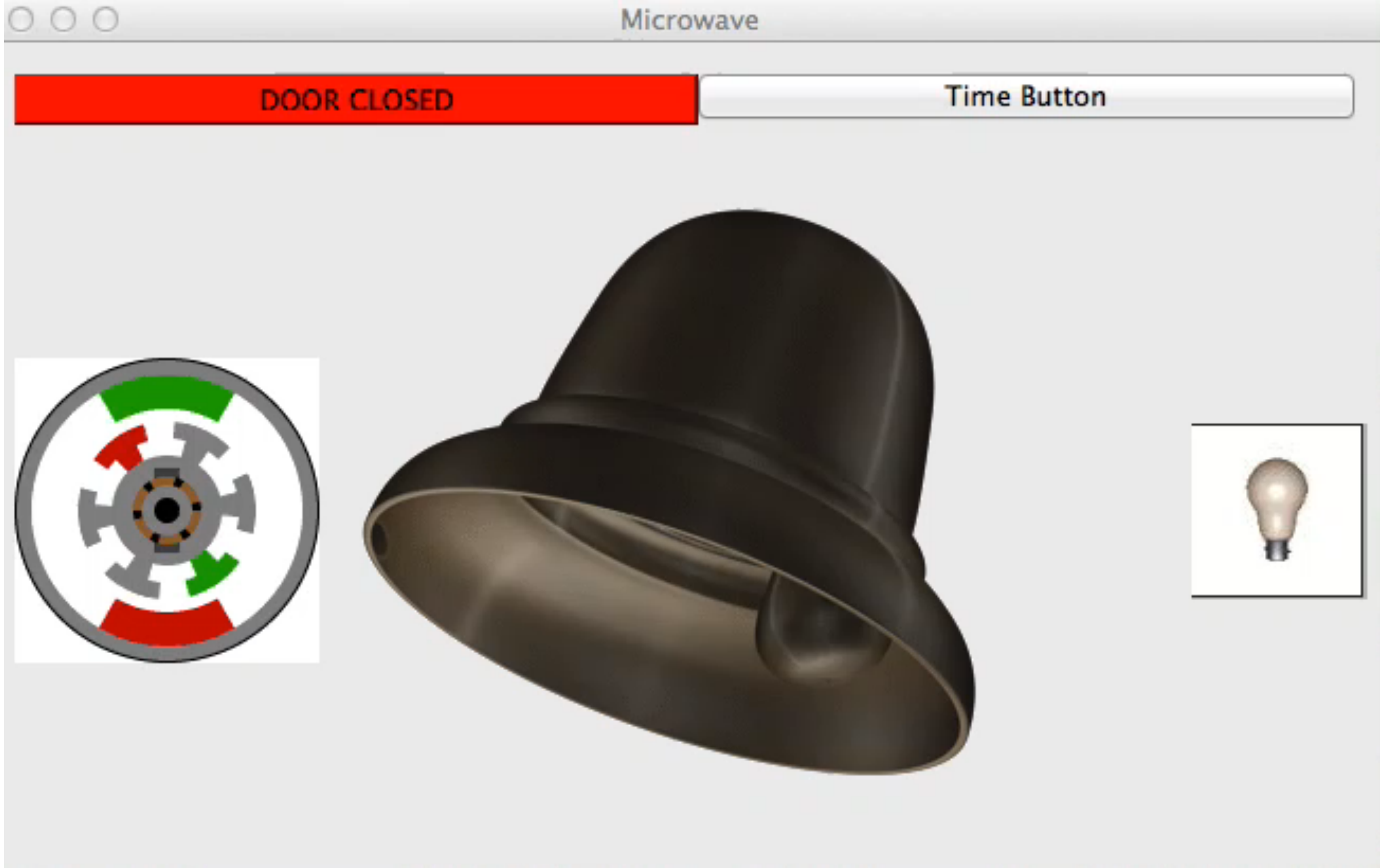
4 DECREMENT
OnEntry {currentTime=currentTime-1;}
OnExit {}
{}

3 ADD_60
OnEntry {currentTime=60+currentTime;}
OnExit {timeLeft=1;}
{}

HARDWARE

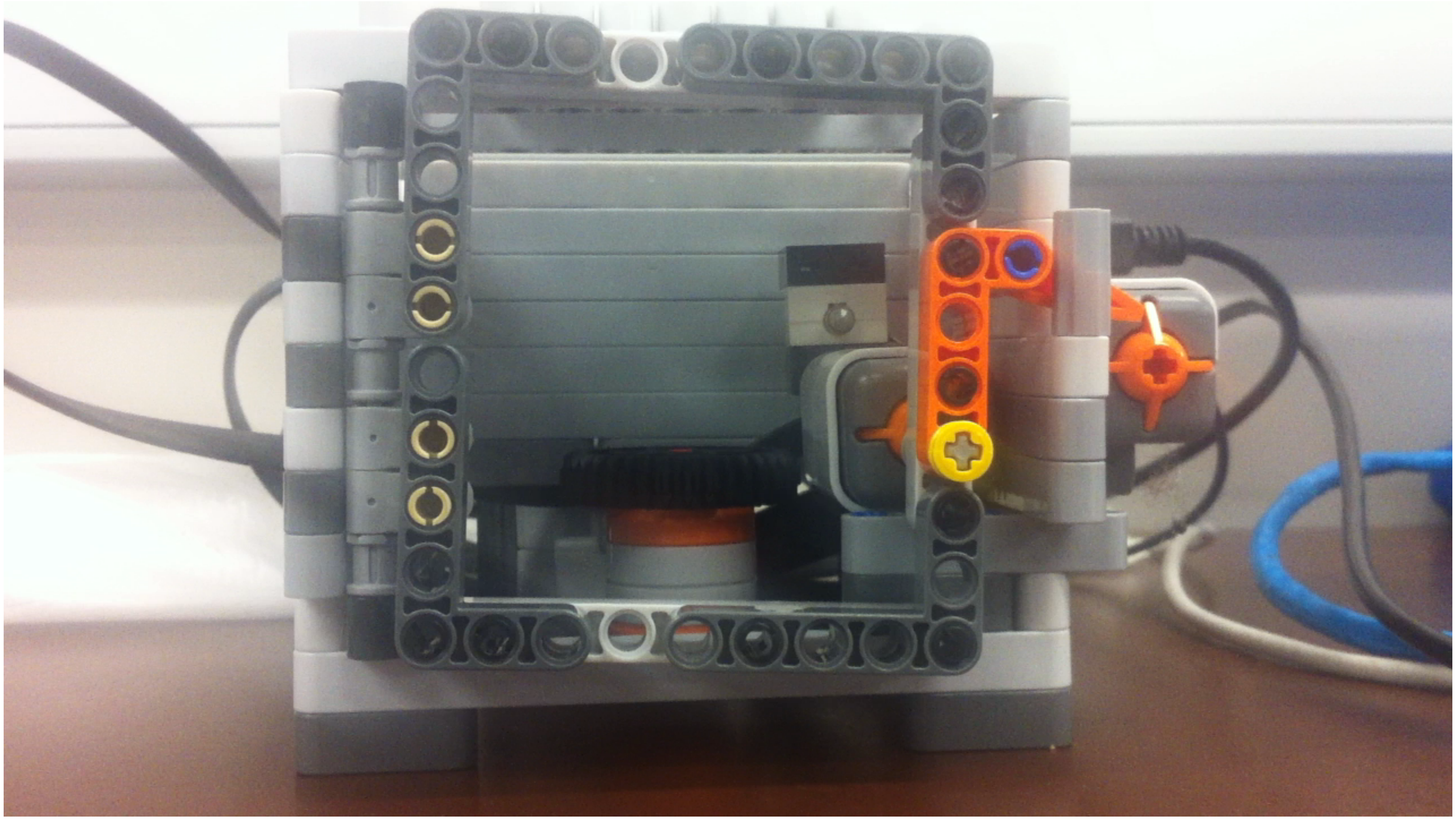
Qt GUI





SIMULATION demo video (**C++**)

http://www.youtube.com/watch?v=Dm3SP3q9_VE



Demo video (C++)

Model Checking and Validation

- Properties
  - **Property 1**: *Necessarily, the oven stops (after* several *steps, i.e. a small, finite number of transitions in the Kripke structure) after the door opens.*"
  - **Property-2:** "*It is necessary to pass through a state in which the door is closed to reach a state in which the motor is working and the machine has started.*"
  - **Property-3:** "*Necessarily, the oven stops(after* several *steps, i.e. again, a small, finite number of transitions in the Kripke structure) after the timer has expired.*"
  - **Property-4:** "*Cooking may go on for ever (e.g. if the user repeatedly keeps pressing the* add *button while the timer is still running).*"

Formal description of the Property in LTL

- Using NUSMV's code
  - "***the cooking must stop if the door is held open***"

```
SPEC
AG( (E$$doorOpen=1 & M0$$motor=1) ->
        AX(  (E$$doorOpen=1 -> M0$$motor=0) | AX(
               (E$$doorOpen=1 -> M0$$motor=0) | AX(
              (E$$doorOpen=1 -> M0$$motor=0) | AX(
               (E$$doorOpen=1 -> M0$$motor=0) | AX(
               (E$$doorOpen=1 -> M0$$motor=0) | AX(
               (E$$doorOpen=1 -> M0$$motor=0) | AX(
               (E$$doorOpen=1 -> M0$$motor=0) | AX(
               M0$$motor=0)))))))))
```

Failure Mode Analysis

- New components come into place

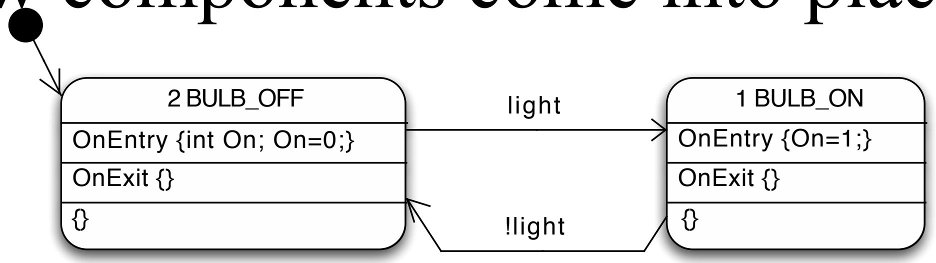
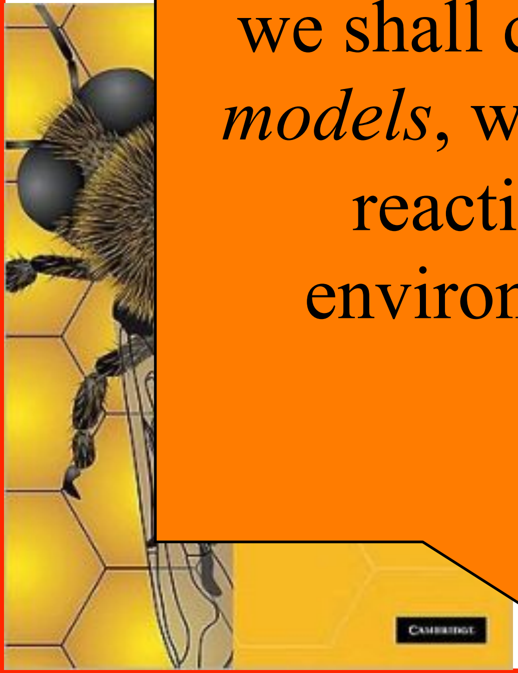


Figure 3: A model of the light bulb hardware component.

- Fault injection determines the effects
  1. to remove behavior from the model (an *omission failure*) and test all properties, and
  2. to modify (a *value failure*) behavior and test all properties.

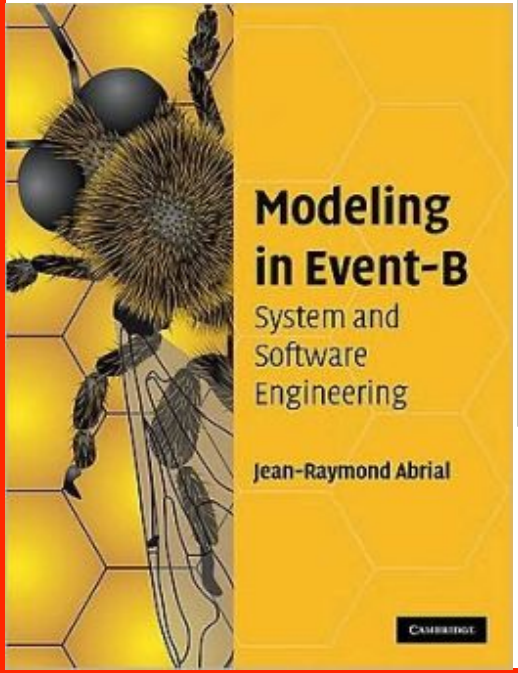
The comparison with UML-B?



Note that the models we are going to construct will not just describe the control part of our intended system, they will also contain a certain **representation of the environment** within which the system we build is supposed to behave. In fact, we shall quite often essentially construct *closed models*, which are able to exhibit the actions and reactions taking place between a certain environment and a corresponding, possibly distributed, controller [8]

Potentially carry out FMEA

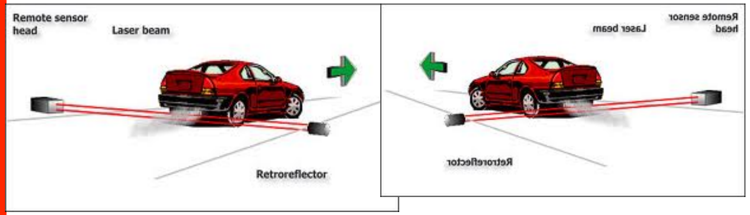
The negation of MDD

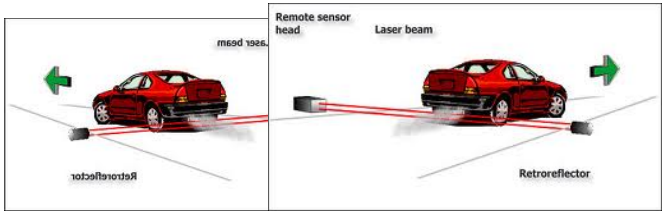

Modeling in Event-B
System and Software Engineering
Jean-Raymond Abrial

"In no way is the model of a program, the program itself. But the model of a program and more generally of a complex computer system, although **not executable**, allows us to clearly identify the properties of the future system and to prove that they will be present in it." [8, Page 13].

"Note again that, as with blueprints, the basis is lacking: our model will **thus not in general be executable**" [8, Page 17].

Case Study I: Bridge to the Island (64-page chapter [8])





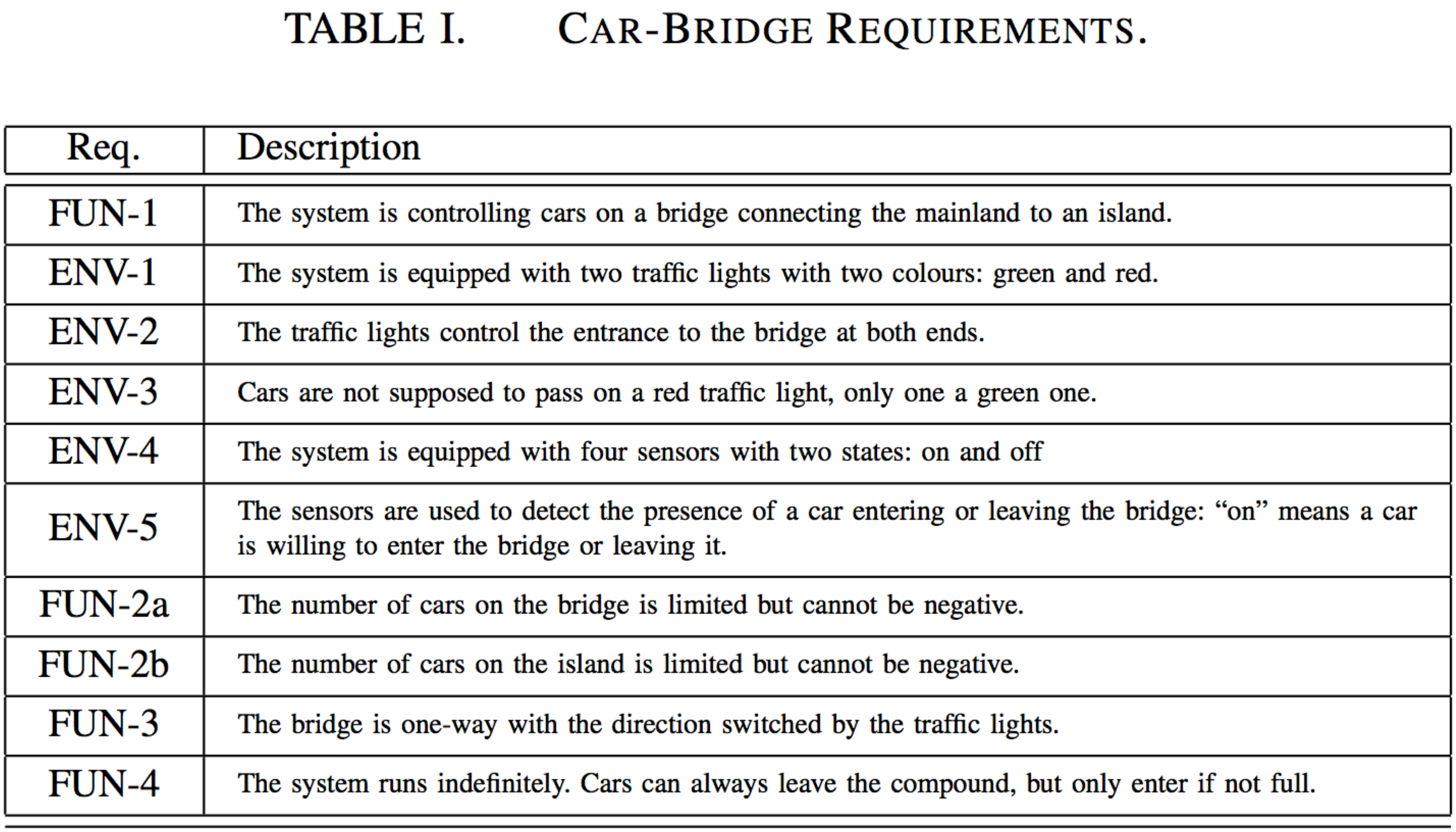


TABLE I. CAR-BRIDGE REQUIREMENTS.

Req.	Description
FUN-1	The system is controlling cars on a bridge connecting the mainland to an island.
ENV-1	The system is equipped with two traffic lights with two colours: green and red.
ENV-2	The traffic lights control the entrance to the bridge at both ends.
ENV-3	Cars are not supposed to pass on a red traffic light, only one a green one.
ENV-4	The system is equipped with four sensors with two states: on and off
ENV-5	The sensors are used to detect the presence of a car entering or leaving the bridge: “on” means a car is willing to enter the bridge or leaving it.
FUN-2a	The number of cars on the bridge is limited but cannot be negative.
FUN-2b	The number of cars on the island is limited but cannot be negative.
FUN-3	The bridge is one-way with the direction switched by the traffic lights.
FUN-4	The system runs indefinitely. Cars can always leave the compound, but only enter if not full.

As usual, requirements are refined

Correctness by construction

- Build the model incrementally
- Always (formally) verifying correctness
  - Events
    - `ML_Out`: A car has gone out of the mainland onto the compound of the bridge and the island.
    - `ML_In` : A car has moved away from the island/ bridge compound onto the mainland.
  - Variables
    - d capacity of the island (and bridge)
    - n cars in compound (bridge & island)

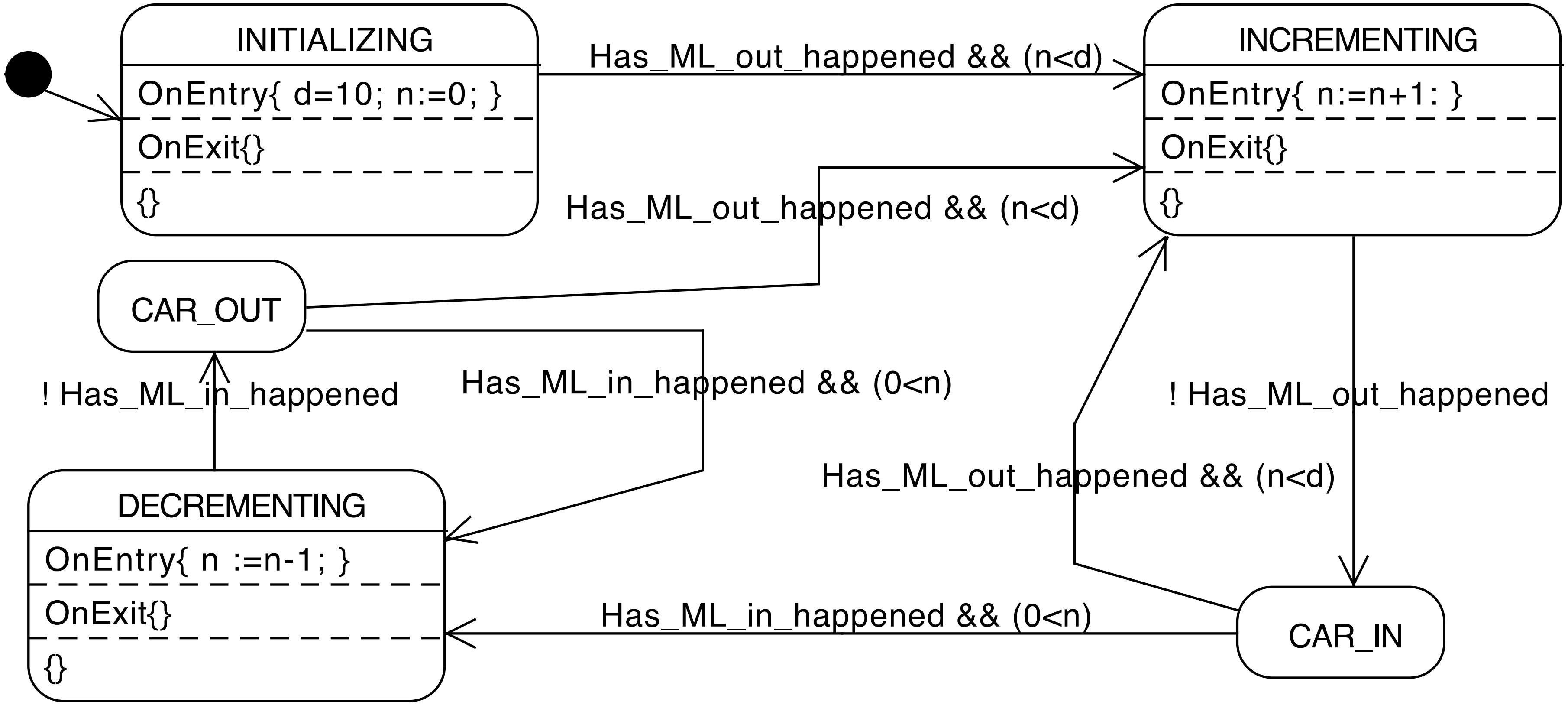


Figure 1: LLFSM model for level one of the car-bridge.

Verified properties

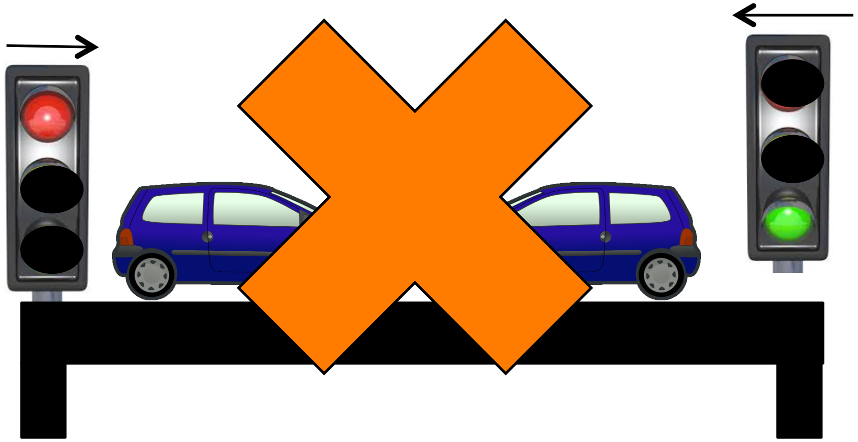
- always $0 \leq n \wedge n < d$
- d remains constant
- once $n = d-1$, any alternation of the values of `has_ML_Out_happened` (between TRUE and FALSE) will not change the value of n.
- `has_ML_Out_happened` must be set to FALSE before the setting of `has_ML_In_happened` to TRUE causes the value of n to decrease
- under the assumption $0<d$, the statement $(n<d \vee 0 \leq n)$ in all future states.

The second level

- New events:
  - `IL_In` : A car has gone out of the bridge onto the island.
  - `IL_Out` : A car has moved off the island onto the bridge.
- New variables
  - *n_b2i* : the number of cars on the bridge heading towards the island.
  - *ni* : the number of cars on the island.
  - *n_b2m* : the number of cars on the bridge heading towards the mainland.

The invariant

- $(n\_b2i == 0) \lor (n\_b2m == 0)$.
  - That is, enforcing the bridge is one-way (in one direction or the other).



Still prove deadlock free

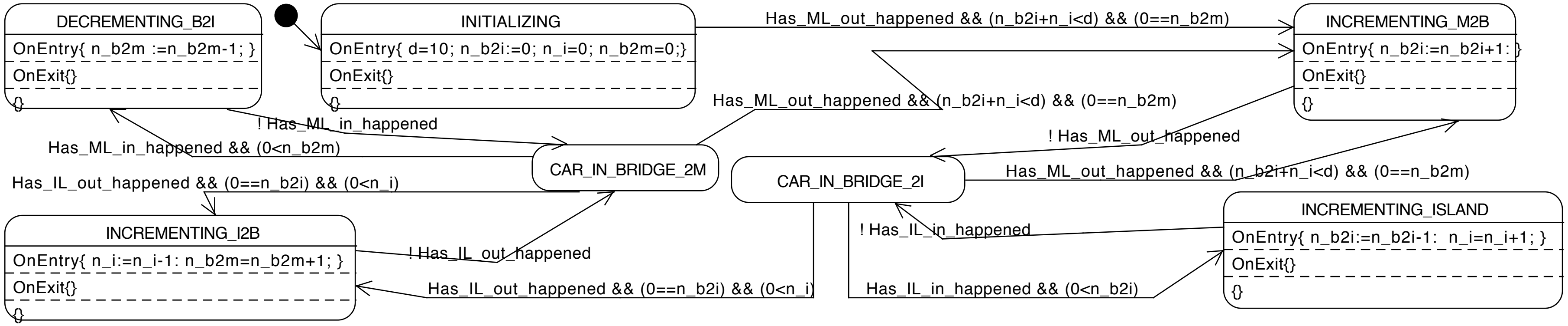
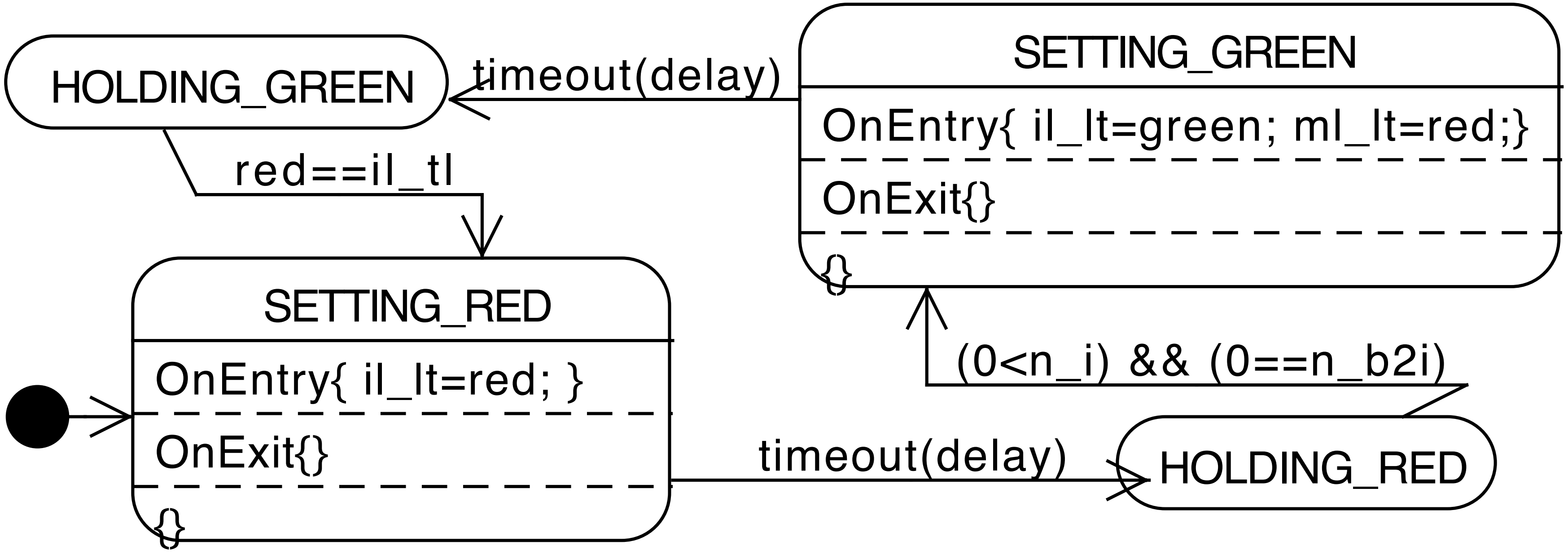


Fig. 2. LLFSM model for level two of the car-bridge.

- $(n\_b2m>0) \vee ((n\_b2i+ni<d) \wedge (n\_b2m==0)) \vee (n\_b2i>0) \vee ((ni>0) \wedge (n\_b2i==0))$.

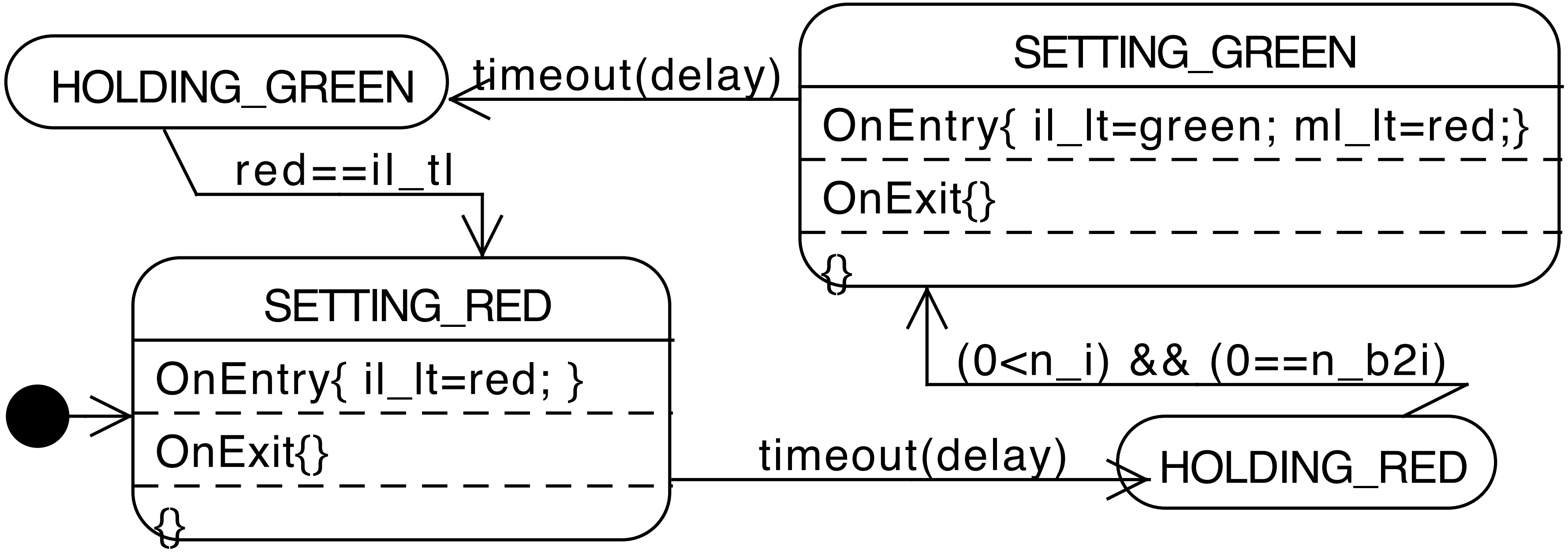
Level three

- Introducing the lights
  - Variables
    - *ml_lt* : The colour of the light on the mainland side.
    - *il_lt* : The colour of the light on the island side.



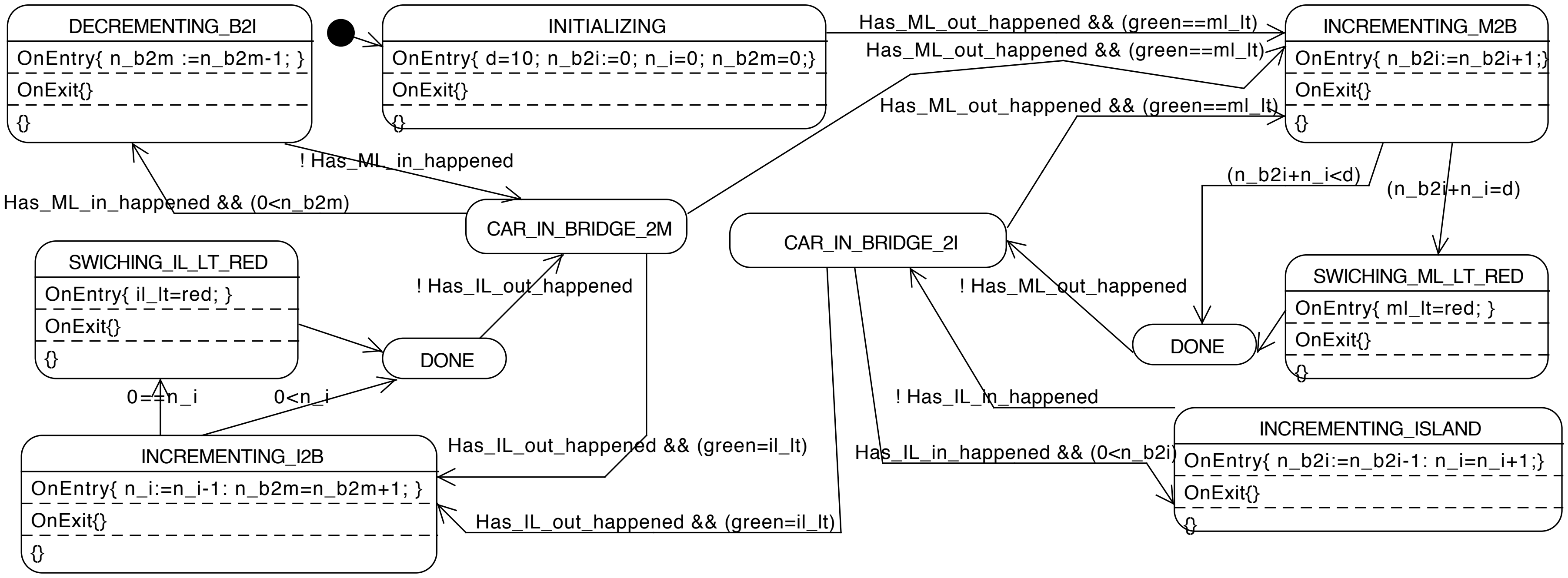
(a) The mainland light.

Figure 3. Versatile model at the level of lights for the car-bridge controller.



(b) The island light.

Figure 3. Versatile model at the level of lights for the car-bridge controller.



(c) The central controller

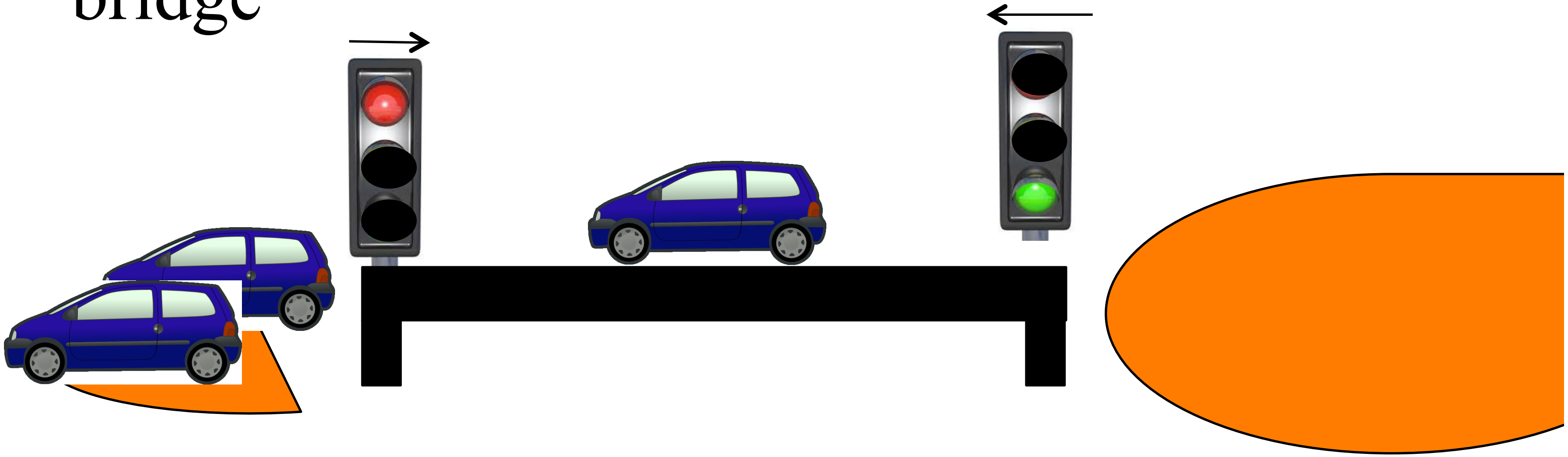
Figure 3. Versatile model at the level of lights for the car-bridge controller.

UML-B does not model delays

- As a consequence, to ensure that the lights alternates, it forcers cars to alternate.
- Does not even notice this awkward behaviour of the system (model).

Scenario where the UML-B model fails

- Capacity is no more than 4 cars
- Two (2) cars to go onto the bridge
- One (1) car to go from the bridge onto the island
- A car from the mainland can still go on the bridge

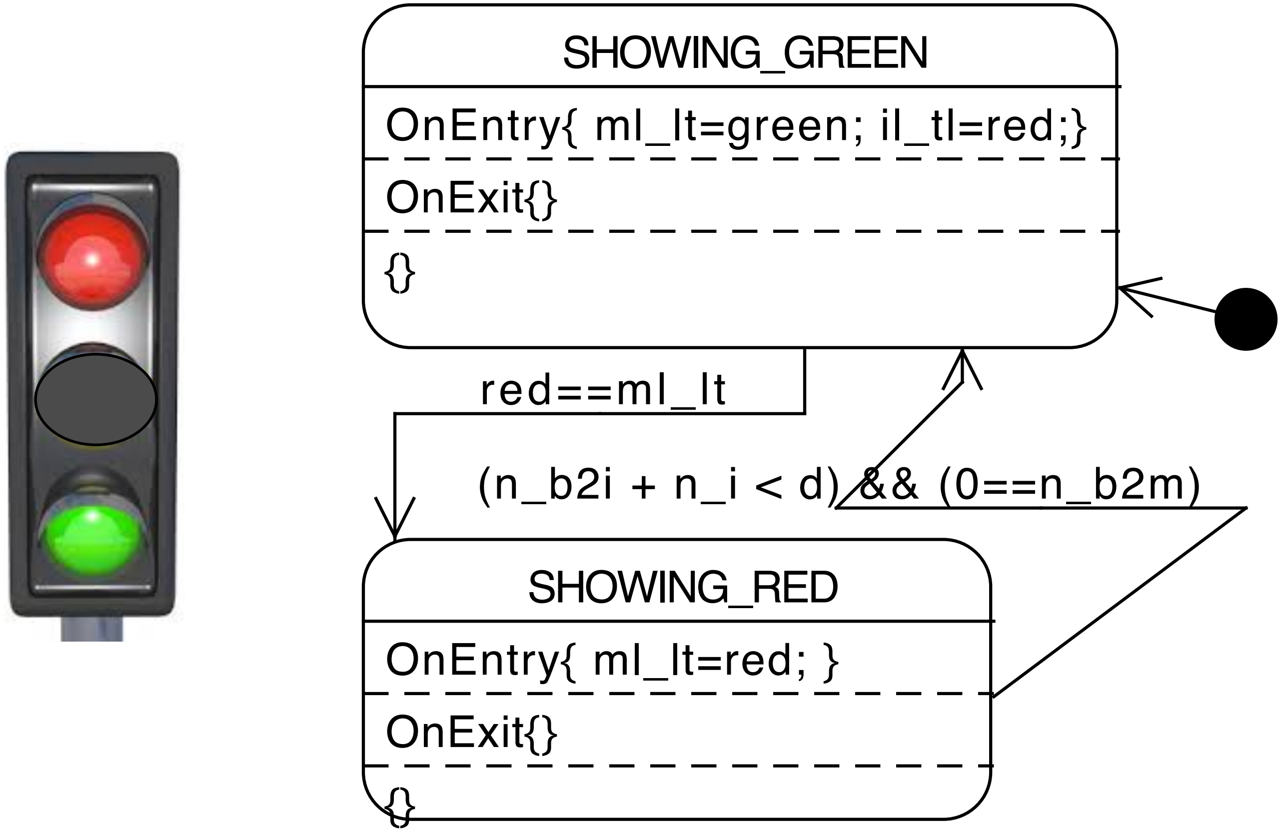


Scenario where the UML-B model fails

- If the third car reaches the island and no other car takes the bridge

Scenario where the UML-B model fails

- If the third car reaches the island and no other car takes the bridge
- A fourth car cannot go to the island!!
- Has to wait for a car out, and there is still capacity



SHOWING_GREEN

OnEntry{ ml_lt=green; il_tl=red;}

OnExit{}

{}

red==ml_lt

(n_b2i + n_i < d) && (0==n_b2m)

SHOWING_RED

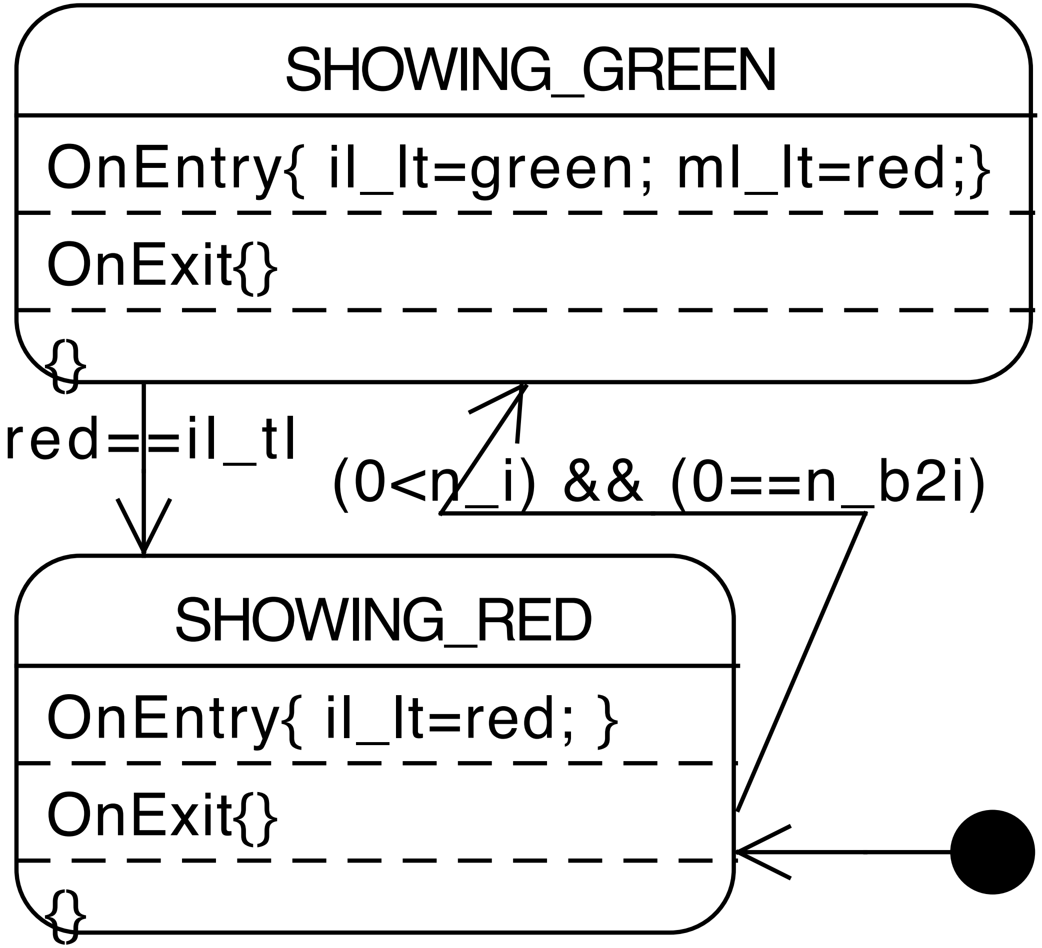
OnEntry{ ml_lt=red; }

OnExit{}

{}

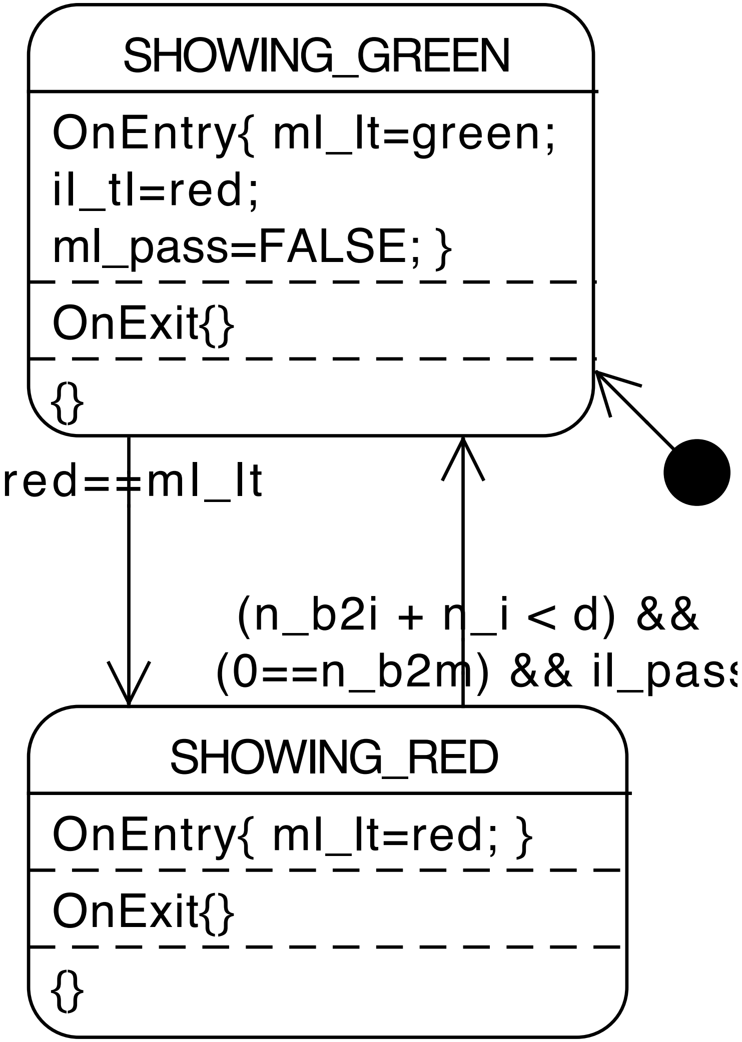
(a) The mainland light.

Fig4. Lights models with no delay as [8, Sec. 2.6.1 to 2.6.7].



(b) The island light.

Fig4. Lights models with no delay as [8, Sec. 2.6.1 to 2.6.7].



SHOWING_GREEN

OnEntry{ ml_lt=green;
il_tl=red;
ml_pass=FALSE; }

OnExit{}

{}

red==ml_lt

(n_b2i + n_i < d) &&
(0==n_b2m) && il_pas

SHOWING_RED

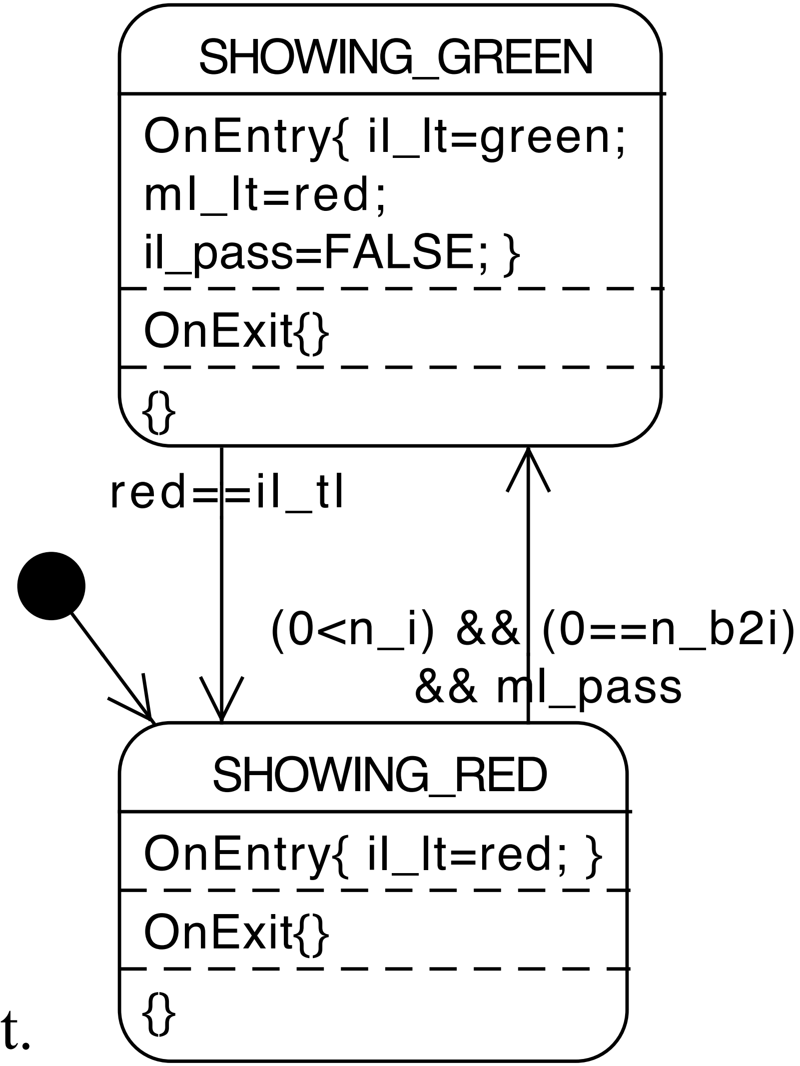
OnEntry{ ml_lt=red; }

OnExit{}

{}

(a) The mainland light.

Figure 5 Modification to Fig. 3 to create the forced-alternation model of [8, Sec. 2.8.8].



SHOWING_GREEN

OnEntry{ il_lt=green;
ml_lt=red;
il_pass=FALSE; }

OnExit{}

{}

red==il_tl

(0<n_i) && (0==n_b2i)
&& ml_pass

SHOWING_RED

OnEntry{ il_lt=red; }

OnExit{}

{}

(b) The island light.

Figure 5 Modification to Fig. 3 to create the forced-alternation model of [8, Sec. 2.8.8].

INCREMENTING_M2B

OnEntry{
n_b2i:=n_b2i+1:
ml_pass=TRUE; }

OnExit{}

{}

INCREMENTING_I2B

OnEntry{ n_i:=n_i-1:
n_b2m=n_b2m+1;
il_pass=TRUE;}

OnExit{}

{}

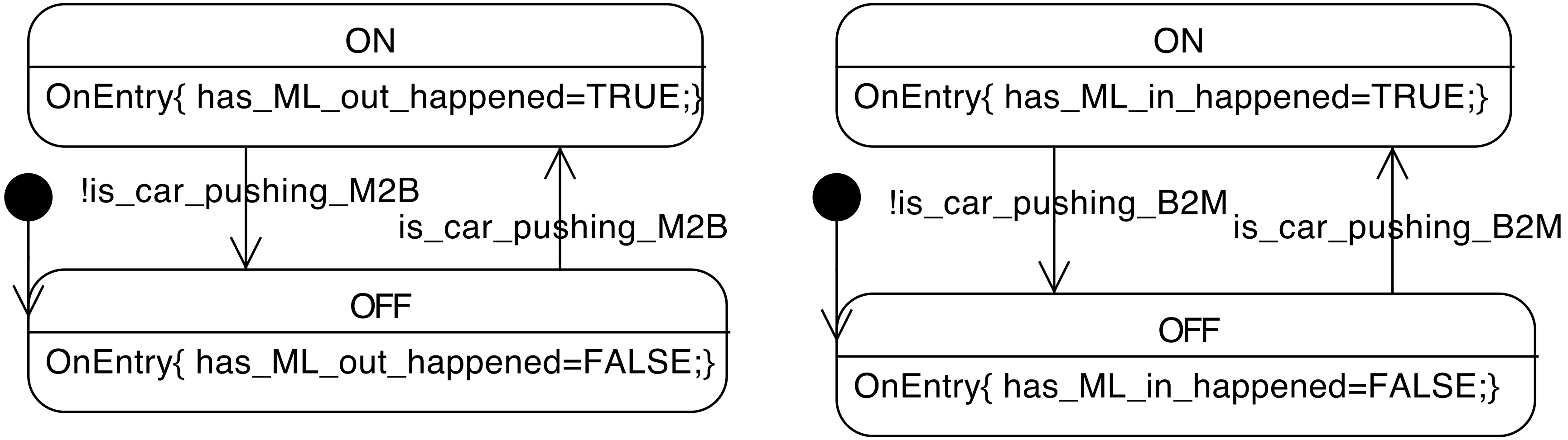
(c) The replaced states.

Figure 5 Modification to Fig. 3 to create the forced-alternation model of [8, Sec. 2.8.8].

Event-B considerations match Logic-Labelled FMS considerations

- “make clearer the separation between the *software controller* and the *physical environment*” [8, Page 89].
- “a *closed model* corresponding to the complete mathematical simulation of the pair formed by the software controller and the environment” [8, Page 89].

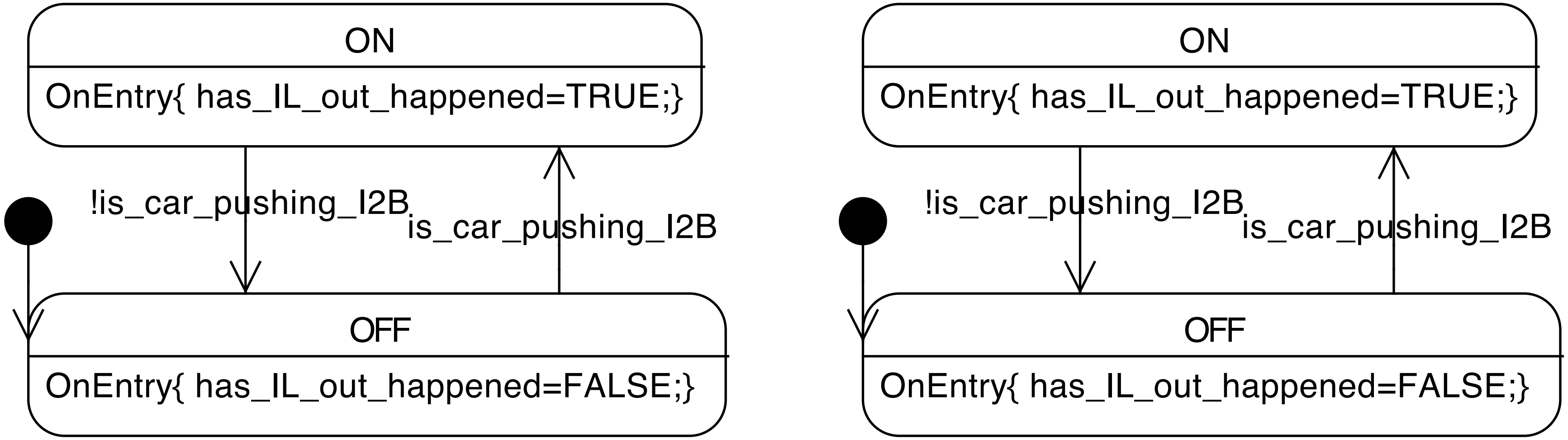
Robots and people


ON
OnEntry{ has_ML_out_happened=TRUE;}
!is_car_pushing_M2B
is_car_pushing_M2B
OFF
OnEntry{ has_ML_out_happened=FALSE;}

ON
OnEntry{ has_ML_in_happened=TRUE;}
!is_car_pushing_B2M
is_car_pushing_B2M
OFF
OnEntry{ has_ML_in_happened=FALSE;}

(a)Mainland to bridge

(b)Island to bridge

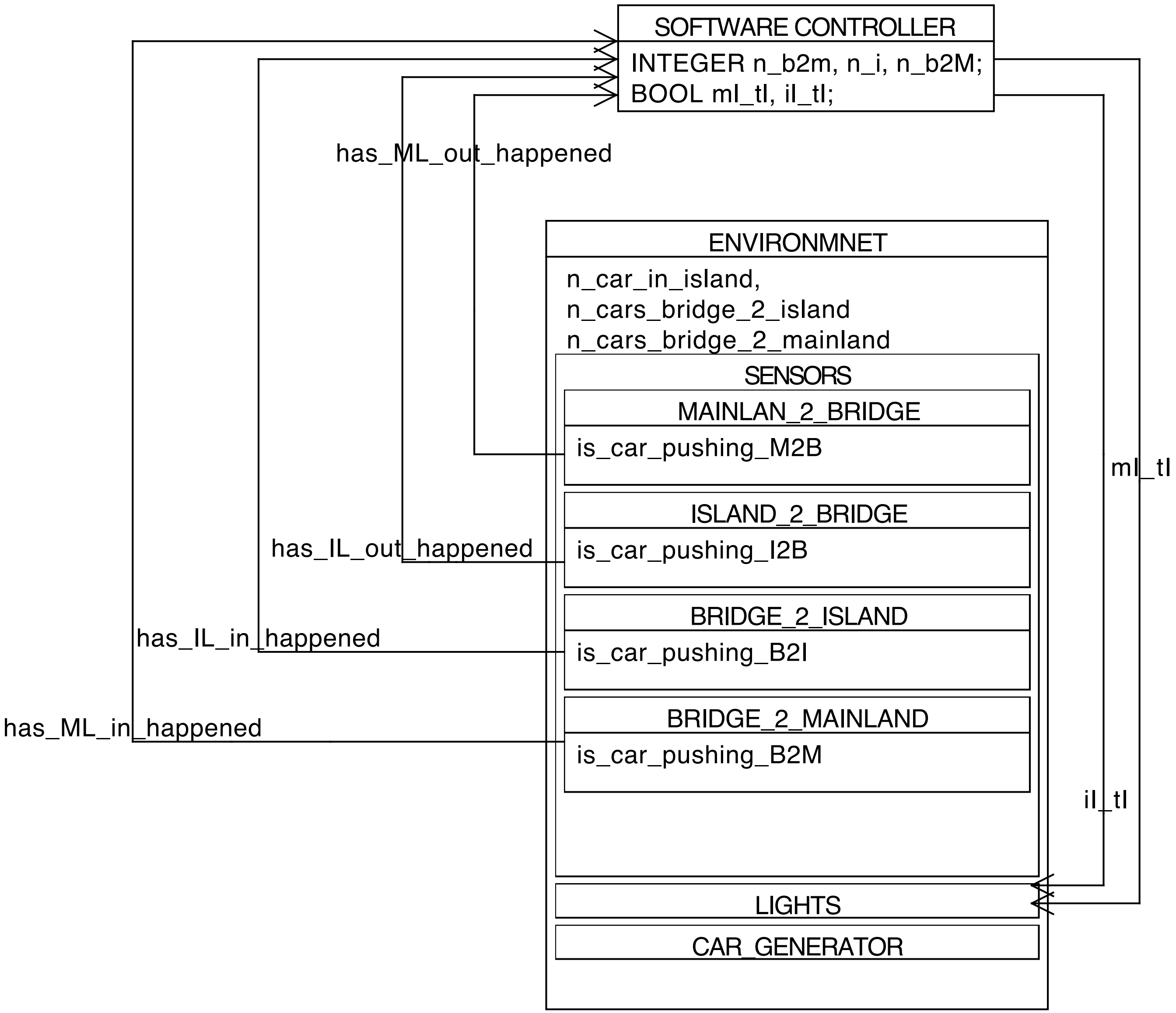

ON
OnEntry{ has_IL_out_happened=TRUE;}
!is_car_pushing_I2B
is_car_pushing_I2B
OFF
OnEntry{ has_IL_out_happened=FALSE;}

ON
OnEntry{ has_IL_out_happened=TRUE;}
!is_car_pushing_I2B
is_car_pushing_I2B
OFF
OnEntry{ has_IL_out_happened=FALSE;}

(c) Bridge to mainland

(d)Bridge to island

Fig. 6 The 4 sensor behaviour models.



SOFTWARE CONTROLLER
INTEGER n_b2m, n_i, n_b2M;
BOOL ml_tl, il_tl;

has_ML_out_happened

ENVIRONMNET
n_car_in_island,
n_cars_bridge_2_island
n_cars_bridge_2_mainland

SENSORS

MAINLAN_2_BRIDGE
is_car_pushing_M2B

ISLAND_2_BRIDGE
is_car_pushing_I2B

BRIDGE_2_ISLAND
is_car_pushing_B2I

BRIDGE_2_MAINLAND
is_car_pushing_B2M

has_IL_out_happened

has_IL_in_happened

has_ML_in_happened

ml_tl

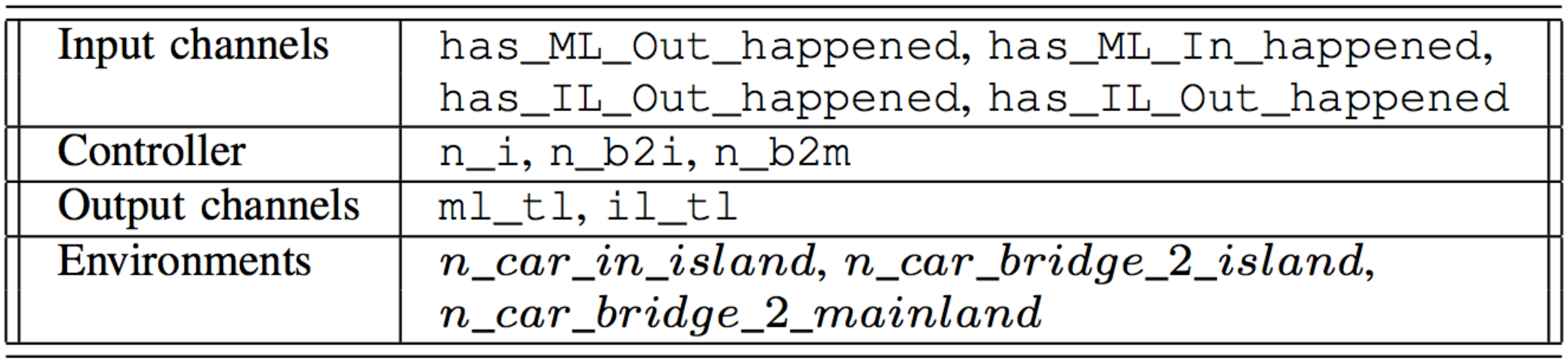
il_tl

LIGHTS

CAR_GENERATOR

Figure 7. The communication channels between the environment and the software controller in Fig 3.

TABLE II. THE TYPES OF VARIABLES USED.



Input channels	`has_ML_Out_happened`, `has_ML_In_happened`, `has_IL_Out_happened`, `has_IL_Out_happened`
Controller	`n_i`, `n_b2i`, `n_b2m`
Output channels	`ml_tl`, `il_tl`
Environments	$n\_car\_in\_island$, $n\_car\_bridge\_2\_island$, $n\_car\_bridge\_2\_mainland$

UML-B requirements

- *Event-B* uses a model of control over environment variables
  - (making sure the environment plays fair).
  - *Event-B*, a driver will never run a red light, for example.
- This does not happen with LLFSMs
  - non-deterministic aspect that is captured in the Kripke structure in NuSMV, capable of reacting to the environment changing the corresponding external variables at any time in any way.

The powered-window in the car

- Driver and passenger can control a passenger window
- Obstacles when going up halt the movement.
- Short push moves the window all the way (down or up)
- Long push regulates the final position (when the long push terminates)

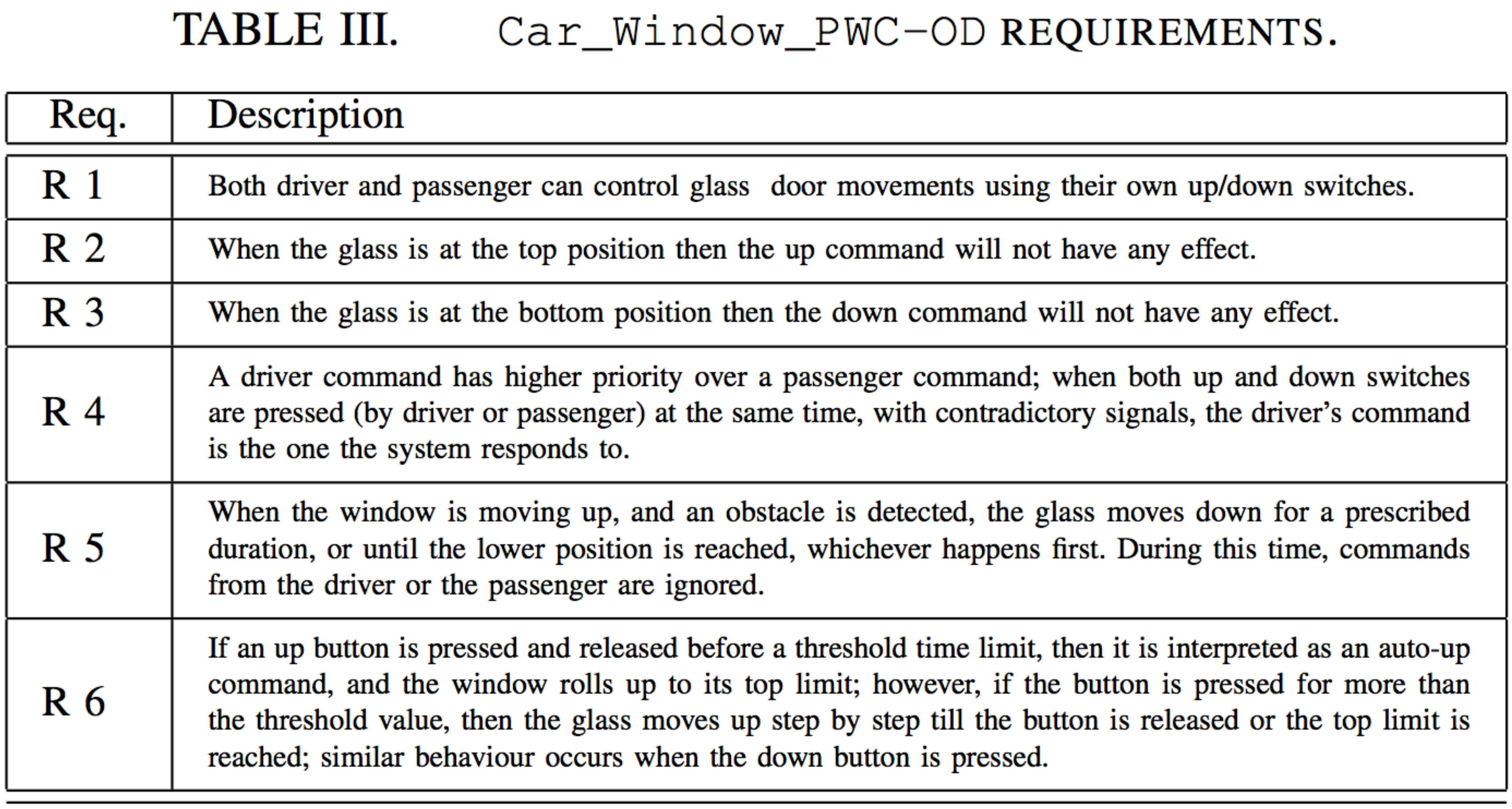
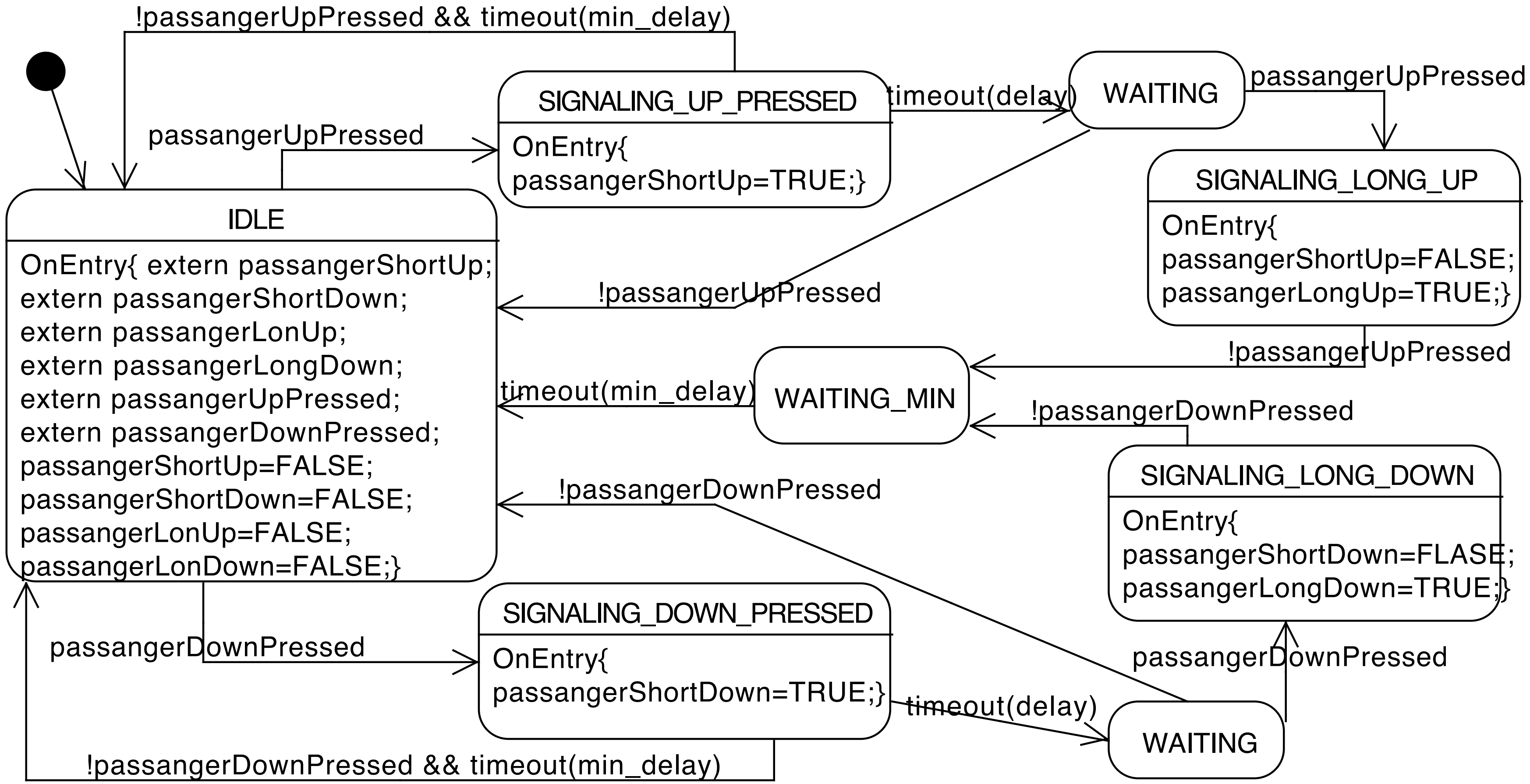


TABLE III. `Car_Window_PWC-OD` REQUIREMENTS.

Req.	Description
R 1	Both driver and passenger can control glass door movements using their own up/down switches.
R 2	When the glass is at the top position then the up command will not have any effect.
R 3	When the glass is at the bottom position then the down command will not have any effect.
R 4	A driver command has higher priority over a passenger command; when both up and down switches are pressed (by driver or passenger) at the same time, with contradictory signals, the driver's command is the one the system responds to.
R 5	When the window is moving up, and an obstacle is detected, the glass moves down for a prescribed duration, or until the lower position is reached, whichever happens first. During this time, commands from the driver or the passenger are ignored.
R 6	If an up button is pressed and released before a threshold time limit, then it is interpreted as an auto-up command, and the window rolls up to its top limit; however, if the button is pressed for more than the threshold value, then the glass moves up step by step till the button is released or the top limit is reached; similar behaviour occurs when the down button is pressed.



!passangerUpPressed && timeout(min_delay)

passangerUpPressed

SIGNALING_UP_PRESSED

OnEntry{
passangerShortUp=TRUE;}

timeout(delay)

WAITING

passangerUpPressed

SIGNALING_LONG_UP

OnEntry{
passangerShortUp=FALSE;
passangerLongUp=TRUE;}

IDLE

OnEntry{ extern passangerShortUp;
extern passangerShortDown;
extern passangerLonUp;
extern passangerLongDown;
extern passangerUpPressed;
extern passangerDownPressed;
passangerShortUp=FALSE;
passangerShortDown=FALSE;
passangerLonUp=FALSE;
passangerLonDown=FALSE;}

!passangerUpPressed

!passangerUpPressed

timeout(min_delay)

WAITING_MIN

!passangerDownPressed

SIGNALING_LONG_DOWN

OnEntry{
passangerShortDown=FLASE;
passangerLongDown=TRUE;}

!passangerDownPressed

passangerDownPressed

SIGNALING_DOWN_PRESSED

OnEntry{
passangerShortDown=TRUE;}

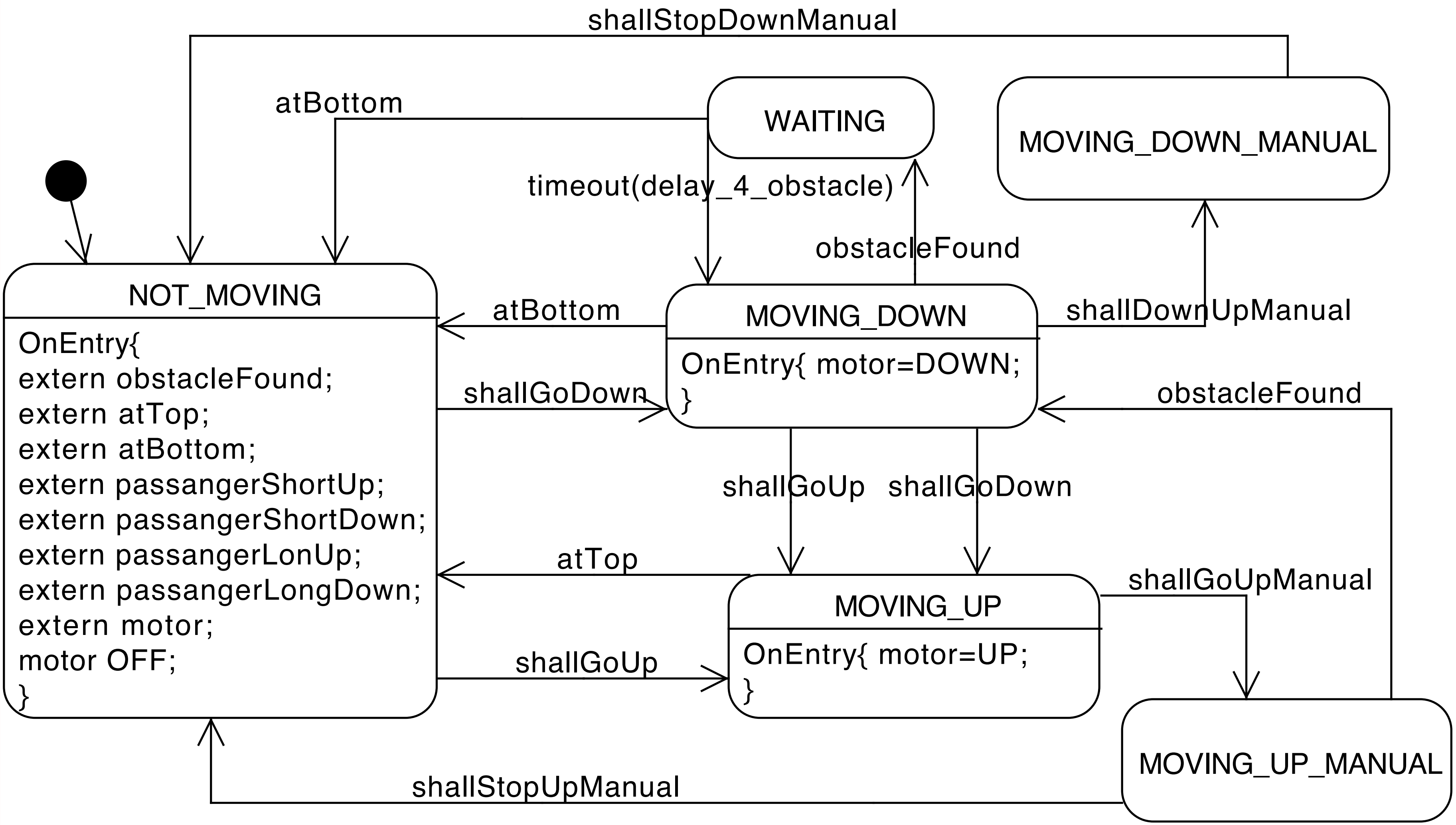
timeout(delay)

WAITING

passangerDownPressed

!passangerDownPressed && timeout(min_delay)

Figure 8. The LLFSM for the button of the passenger.



NOT_MOVING

```
OnEntry{
extern obstacleFound;
extern atTop;
extern atBottom;
extern passangerShortUp;
extern passangerShortDown;
extern passangerLonUp;
extern passangerLongDown;
extern motor;
motor OFF;
}
```

MOVING_DOWN

```
OnEntry{ motor=DOWN;
}
```

MOVING_UP

```
OnEntry{ motor=UP;
}
```

Figure 10. The LLFSM for the motor of the window.

```
name{SHALLGOUP}.
input{passangerLongUp}. input{passangerShortUp}.
input{driverLongUp}.    input{driverShortUp}.
input{driverLongDown}.  input{driverShortDown}.
input{obstacleFound}.
input{atTop}.

UP0: {} => ~shallGoUp.

UP1: passangerLongUp => shallGoUp.  UP1>UP0.
UP2: passangerShortUp => shallGoUp. UP2>UP0.
UP3: driverLongUp => shallGoUp.     UP3>UP0.
UP4: driverShortUp => shallGoUp.    UP4>UP0.

UP5: driverLongDown => ~shallGoUp.  UP5>UP1. UP5>UP2.
UP6: driverShortDown => ~shallGoUp. UP6>UP1. UP6>UP2.
UP7: obstacleFound => ~shallGoUp.   UP7>UP1. UP7>UP2. UP7>UP3. UP7>UP4.
UP8: atTop => ~shallGoUp.           UP8>UP1. UP8>UP2. UP8>UP3. UP8>UP4.

output{b shallGoUp,"shallGoUp"}.
```

Figure 9. DPL coding for the predicate `ShallGoUp`.

```
name{SHALLGODOWN}.
input{passangerLongDown}.  input{passangerShortDown}.
input{driverLongUp}.        input{driverShortUp}.
input{driverLongDown}.      input{driverShortDown}.
input{obstacleFound}.
input{atBottom}.

DN0: {} => ~shallGoDown.

DN1:passangerLongDown => shallGoDown.  DN1>DN0.
DN2:passangerShortDown => shallGoDown. DN2>DN0.
DN3:driverLongDown => shallGoDown.     DN3>DN0.
DN4:driverShortDown => shallGoDown.    DN4>DN0.

DN5:driverLongUp => ~shallGoDown.      DN5> DN1. DN5>DN2.
DN6:driverShortUp => ~shallGoDown.     DN6> DN1. DN6>DN2.

DN7:obstacleFound => shallGoDown.      DN7> DN6. DN7>DN5. DN7>DN0.

DN8:atBottom => ~shallGoDown.          DN8> DN7. DN8>DN4. DN8>DN3.
                                       DN8>DN2. DN8>DN1.
output{b shallGoDown,"shallGoDown"}..
```

Figure 11. DPL coding for the predicate `ShallGoDown`.

```
name{SHALLGOUPMANUAL}.
input{passangerLongUp}.  input{driverLongUp}.

MUP0: {} => ~shallGoUpManual.

MUP1: passangerLongUp => shallGoUpManual.  MUP1>MUP0.
MUP2: driverLongUp => shallGoUpManual.     MUP2>MUP0.

output{b shallGoUpManual,"shallGoUpManual"}.
```

Figure 12. DPL coding for the predicate `shallGoUpManual.`

```
name{SHALLSTOPUPMANUAL}.
input{passangerLongUp}.  input{driverLongDown}.
input{driverShortDown}.  input{driverLongUp}.  input{atTop}.

SUP0: {} => shallStopUpManual.

SUP1: passangerLongUp => ~shallStopUpManual.  SUP1> SUP0.

SUP2:driverLongDown => shallStopUpManual.     SUP2> SUP1.
SUP3:driverShortDown => shallStopUpManual.    SUP3> SUP1.

SUP4:driverLongUp => ~shallStopUpManual.      SUP4> SUP2. SUP4>SUP3.

SUP5: atTop => shallStopUpManual.             SUP5>SUP4.

output{b shallStopUpManual,"shallStopUpManual"}.
```

Figure 13. DPL coding for the predicate `shallStopUpManual.`

```
shallGoUp≡
       ( driverLongUp
             || driverShortUp
             || passangerLongUp || passangerShortUp)
       &&!
       ( atTop
             || obstacleFound
             || driverLongDown || driverShortDown).
```

Figure 14. Simple-C expression for the logic theory in Fig. 9.

```
SPEC AG (
(passangerShortUp = 1 -> ((passangerShortDown = 0
       & passangerLongDown = 0)
       & passangerLongUp = 0)) | pc = M0S0R0)
```

Figure 15. CTL formula that verifies that only `passangerShortUp` is `TRUE` once some computation has happened

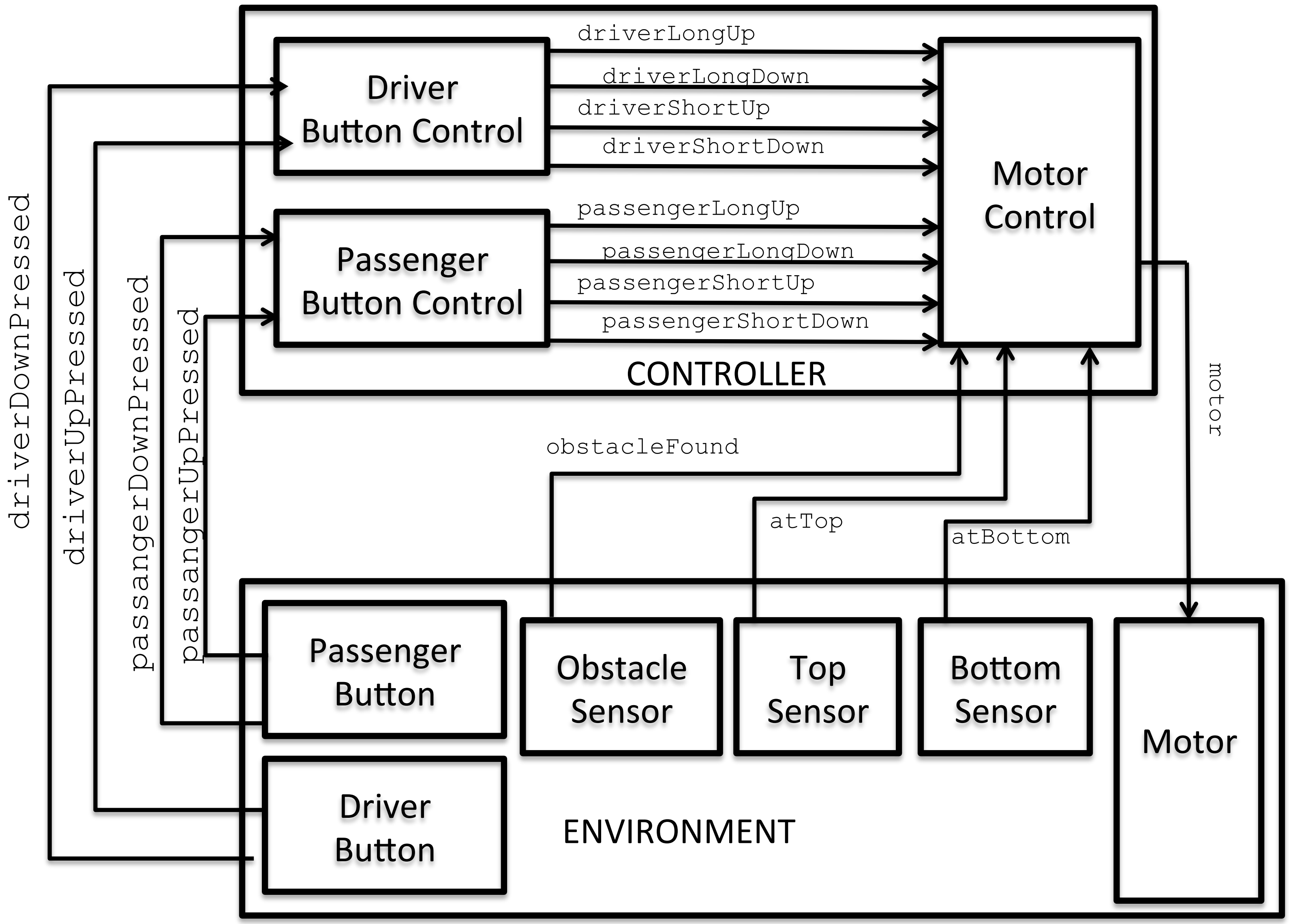


Figure 16. Communication channels of the car-window controller and its environment; the closed world model.

```
LTLSPEC
G ( ( obstacleFound=1 & atTop=0 & driverLongUp=1 & motor=Up ) ->
   X ( obstacleFound=0 | atTop=1 | driverLongUp=0 | motor=Down |
      X ( obstacleFound=0 | atTop=1 | driverLongUp=0 | motor=Down |
         X ( obstacleFound=0 | atTop=1 | driverLongUp=0 | motor=Down |
            X ( obstacleFound=0 | atTop=1 | driverLongUp=0 | motor=Down |
               X ( obstacleFound=0 | atTop=1 | driverLongUp=0 | motor=Down
                 )
              )
           )
        )
      )
  ))
```

Figure 17. Structure of the LTL formula that encodes that and obstacle found will cause the motor to switch direction to going down.

Comparison with Event-B (and its tool UML-B)

- Event-B results in 30-Page *Event-B* specification [18]
- It is incomprehensible to the average learned software engineer
- The specification is longer than the actual code.

Summary

- Logic-labelled finite-state machines are very effective models of behaviour
  - Significantly well established event-driven version
  - But the logic-labelled reduces many complexities without loosing expressive power
- We can simulate behaviours (detect faults)
- We can formally verify models
- We can perform fault injection and FMEA
- Complete Model-Driven Development

THANK YOU