FAILURE MODE AND EFFECTS ANALYSIS (FMEA) AND MODEL-CHECKING OF SOFTWARE FOR EMBEDDED SYSTEMS BY SEQUENTIAL SCHEDULING OF VECTORS OF LOGIC-LABELLED FINITE-STATE MACHINES

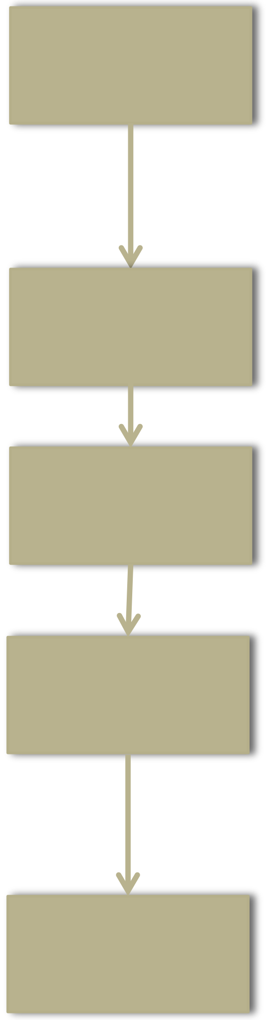
V. ESTIVILL-CASTRO, R. HEXEL*, D.A. ROSENBLUETH†*

**Griffith University, Brisbane, Australia. {v.estivill-castro,r.hexel}@griffith.edu.au,*

†Universidad Nacional Autónoma de Mexico, México City, Mexico. drosenbl@unam.mx

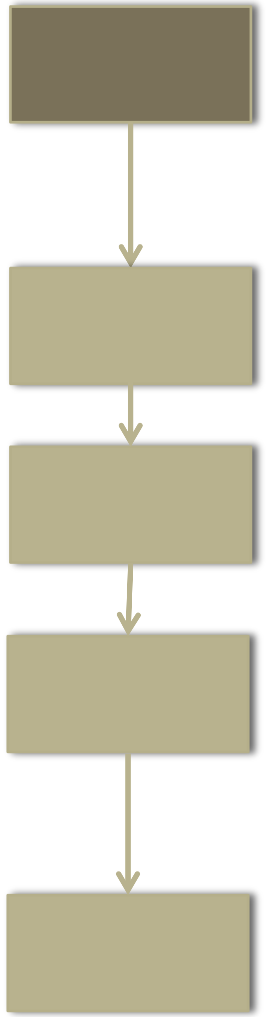
Outline

- Motivation
  - Model-driven development MDD)
- Logic-labeled Finite State Machines
- Vectors of Logic-labeled Machines
- Case studies
  - The mine pump
  - The industrial press
- Conclusions



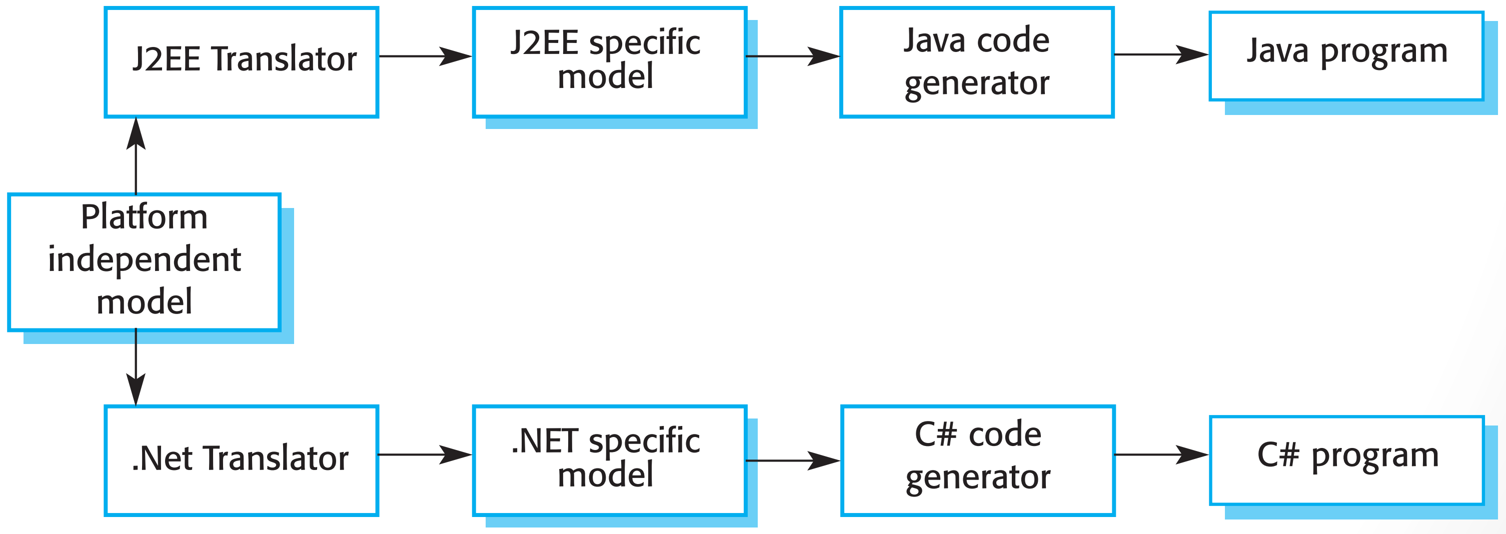
Outline

- Motivation
  - Model-driven development MDD)
- Logic-labeled Finite State Machines
- Vectors of Logic-labeled Machines
- Case studies
  - The mine pump
  - The industrial press
- Conclusions



Model-Driven Engineering

- Approach in Software Engineering
  - Construct software / Safe Software / Quality Software
  - models rather than programs are the principal outputs of the development process (Sommeville, 2009).
  - The programs that execute on a hardware/software platform are then generated automatically from the models.
  - Raises the level of abstraction



Advantages of MDD

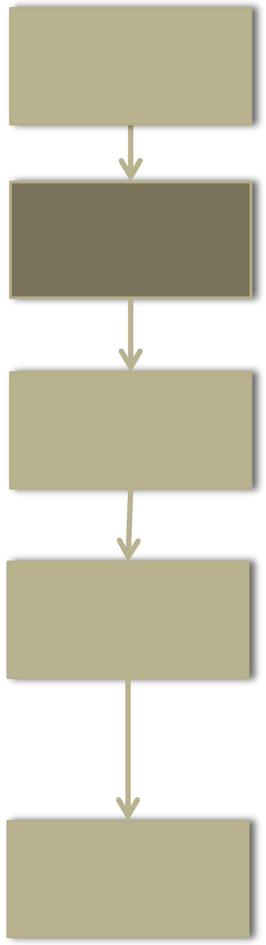
  - direct interpretation (or translation) of the models minimizes faults (with respect to traditional software development and deployment that translates requirements into implementations).
  - offers traceability of requirements as well as rapid refinement and adaption.
- BUT
- all this is spurious if the model is not correct in the first place.
  - inconsistent
  - ambiguous
  - incomplete
  - unsafe

MDD raises the stakes from earlier on

- Importance of Model-Checking
    - Verify the model has correct behavior
- Importance of Failure Modes and Effects Analysis (FMEA)
    - Verify the model is robust and the impact of failures is understood
- NO INTERMIDIATE DEVELOPMENT PHASES
    - WHERE COMMON SENSE OF HUMANS WILL PREVAIL

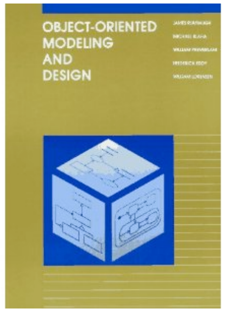
Outline

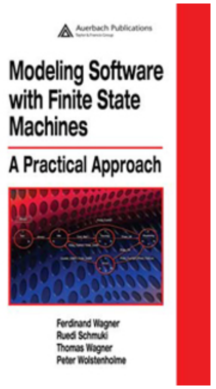
- Motivation
    - Model-driven development (MDD)
- Logic-labeled Finite State Machines
- Vectors of Logic-labeled Machines
- Case studies
    - The mine pump
    - The industrial press
- Conclusions

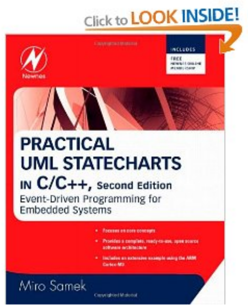


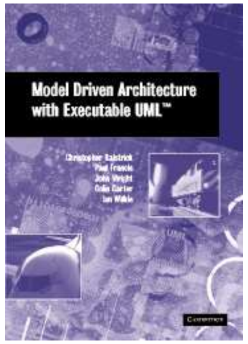
Finite-State Machines (FSM)

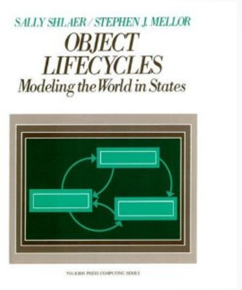
- Widely used model of behavior in embedded systems
  - *QP* (Samek, 2008), *Bot- Studio* (Michel, 2004) *StateWORKS* (Wagner et al., 2006) and *MathWorks*◯R *StateFlow*. The UML form of FSMs derives from OMT (Rumbaugh et al., 1991, Chapter 5), and the MDD initiatives of Executable UML (Mellor and Balcer,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



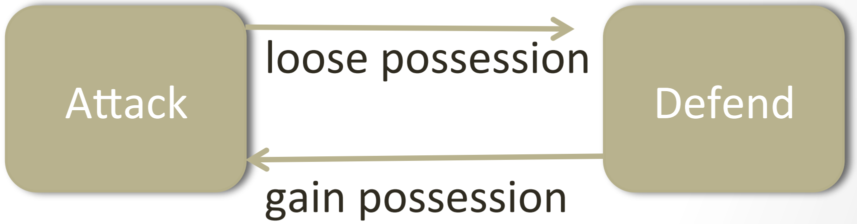






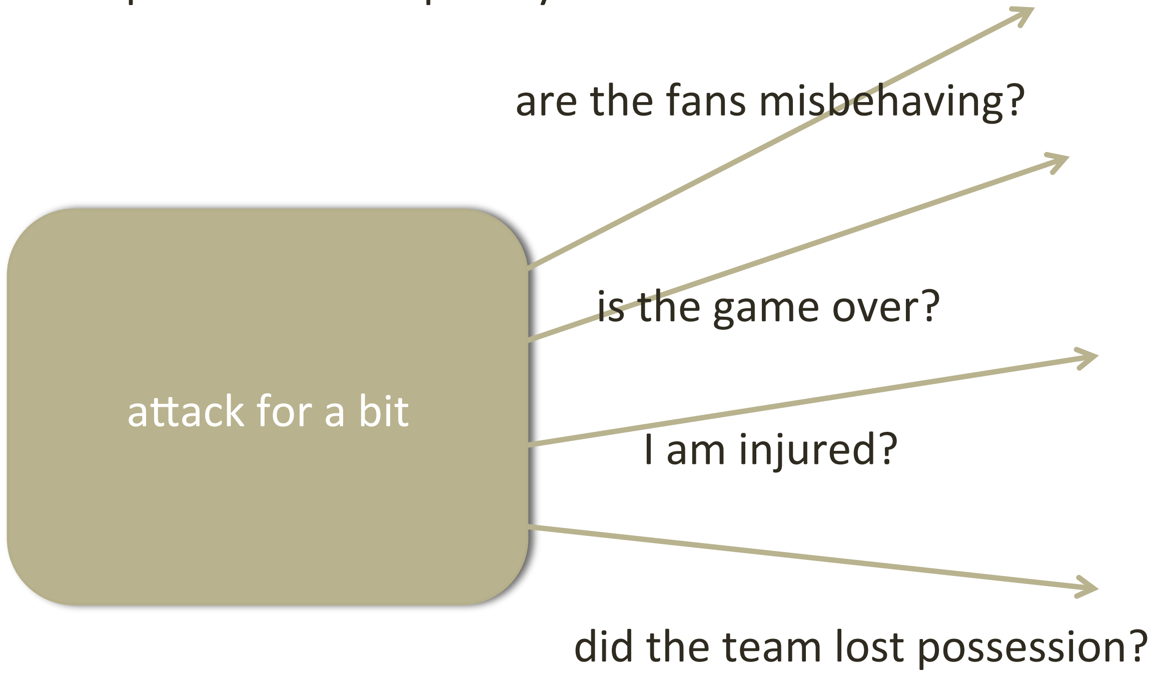

SALLY SHLAER / STEPHEN J. MELLOR
OBJECT LIFECYCLES
Modeling the World in States

- Most common approach
  - System is in a state
  - Events change the state of the system

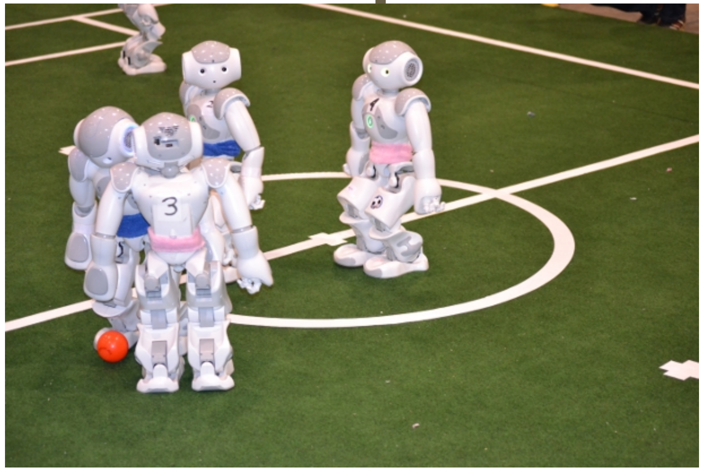


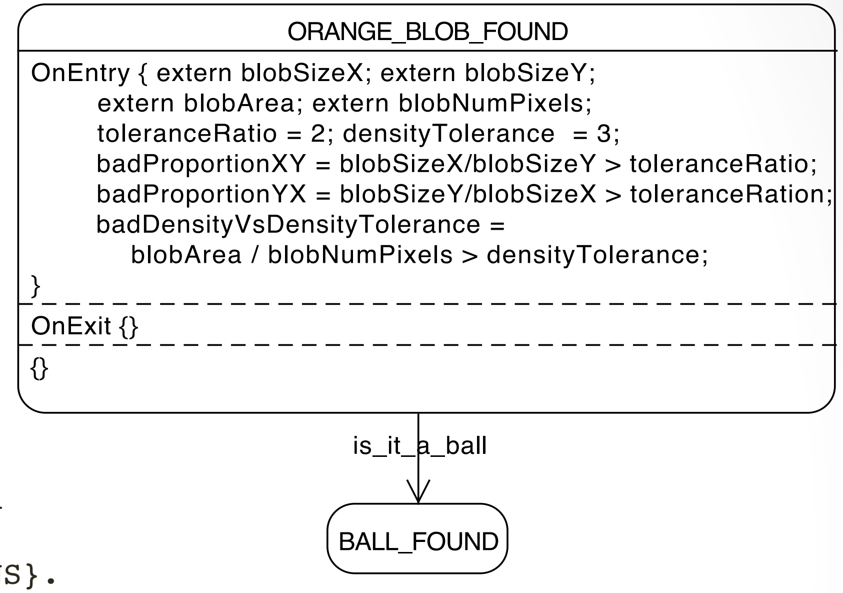
Logic-labeled FSMs

- 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

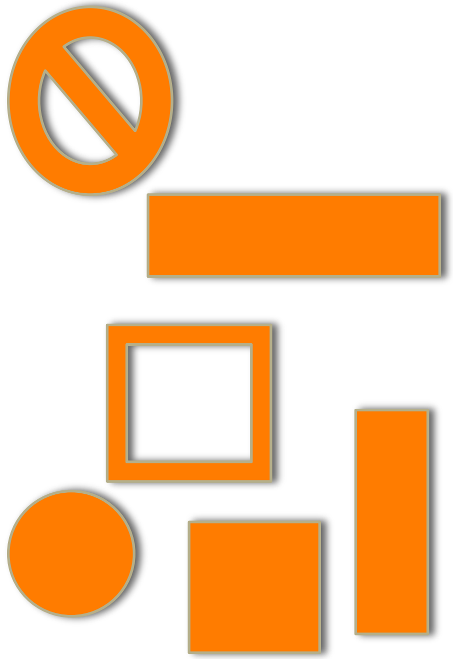


Example from robotic soccer





```
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"}.
```

One Minute Microwave

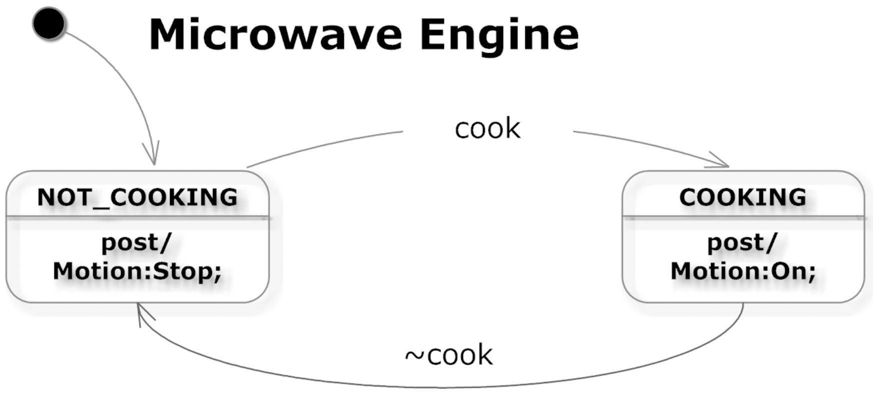
- Widely discussed in the literature of software engineering
- Analogous to the X-Ray machine
  - Therac-25 radiation machine that caused harm to patients
- Important SAFETY feature
  - OPENING THE DOOR SHALL STOP THE COOKING

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.

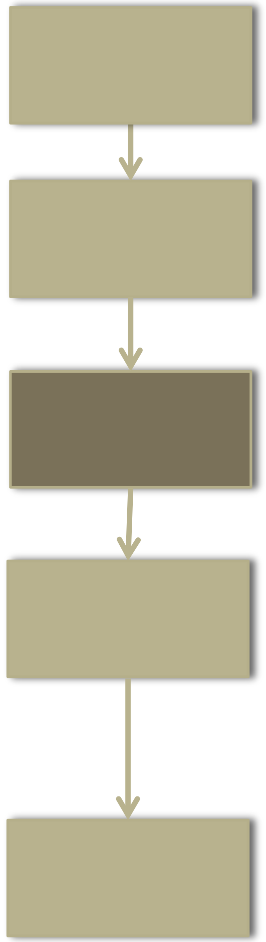
One of the FSMs

```
% MicrowaveCook.d

name{MicrowaveCook}.

input{timeLeft}.
input{doorOpen}.

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

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



Outline

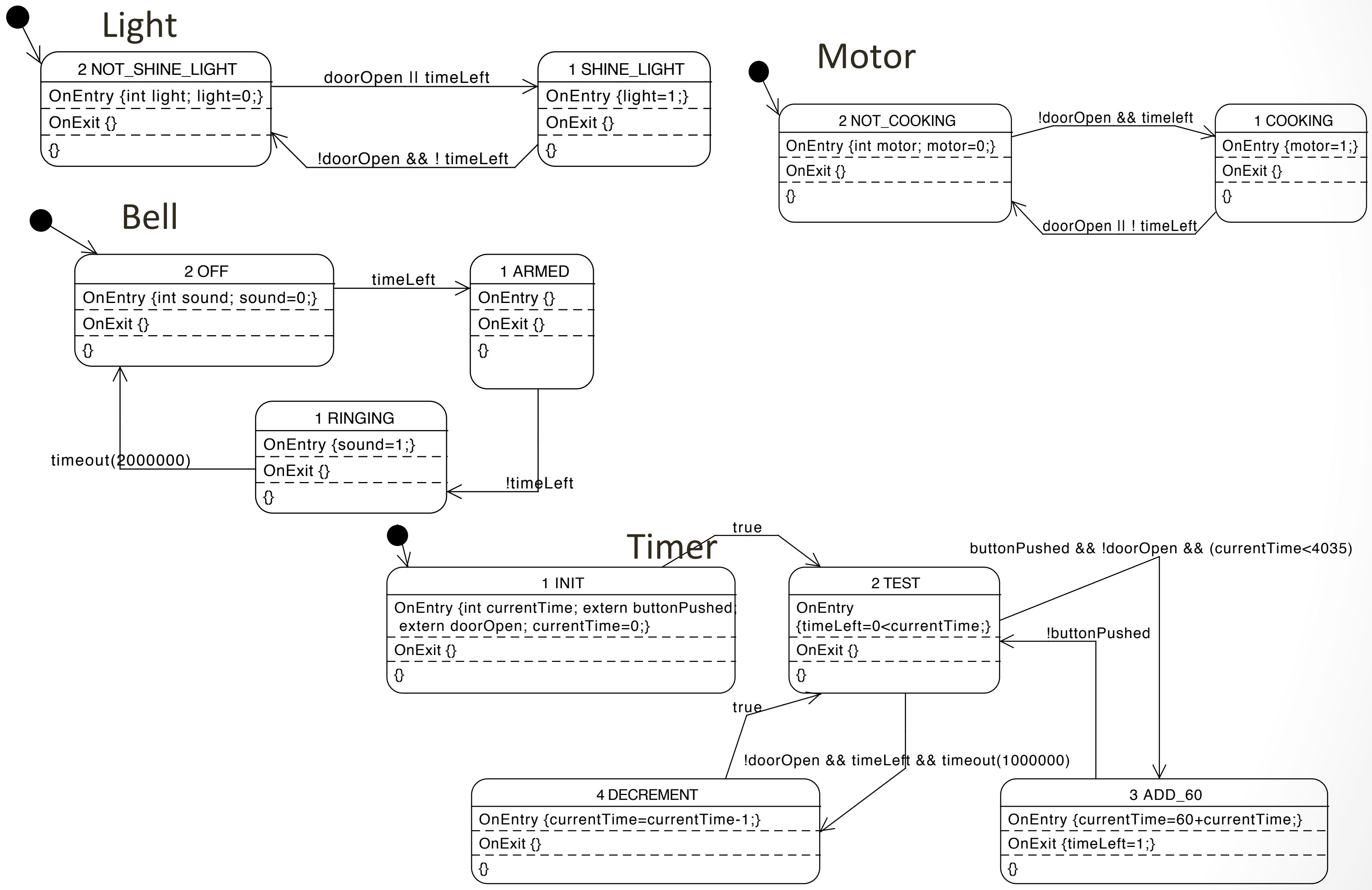
- Motivation
  - Model-driven development MDD)
- Logic-labeled Finite State Machines
- Vectors of Logic-labeled Machines
- Case studies
  - The mine pump
  - The industrial press
- Conclusions

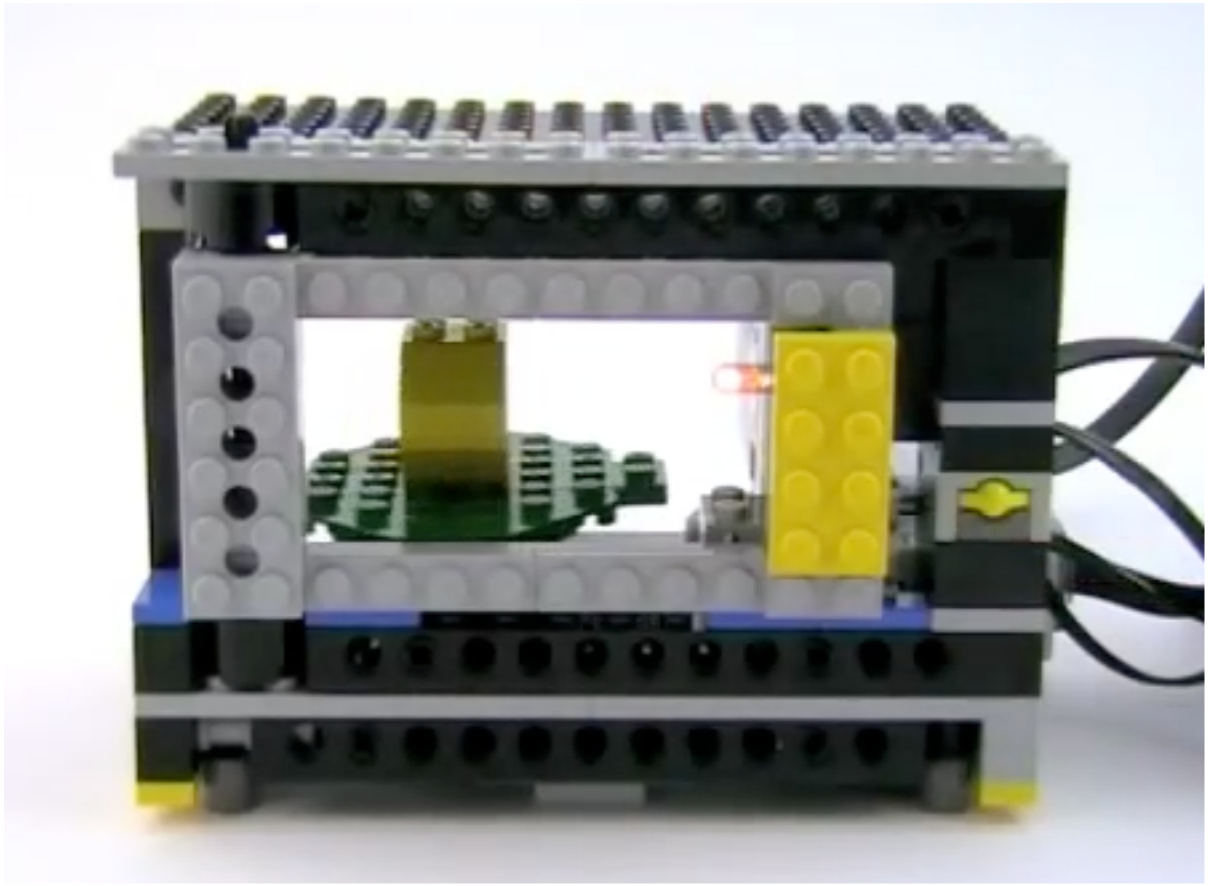


Embedded systems are performing several things

- The models is made of several finite state-machines
- 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 complete arrangement





Demo video

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

Contrast of sequential execution

event-driven models

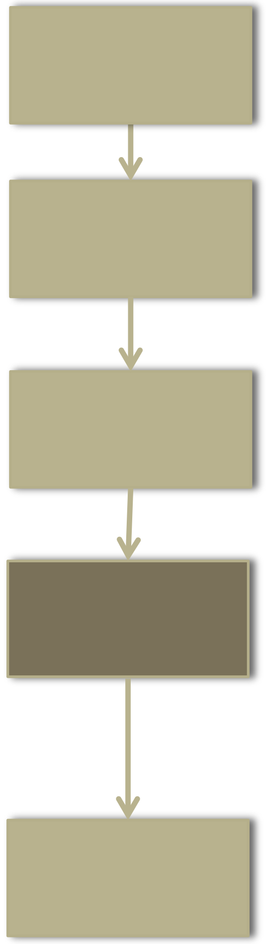
- allow open concurrency
- this means the state of the system are all combinations of states of each thread
- models become complex
  - language constructs for consistency
- model-checking becomes unfeasible
- simulation is not repeatable

time-triggered architecture

- prescribes the scheduling
- reduced space of states of the system
- models are simpler
- model checking becomes feasible
- SIMULATIONS are repeatable

Outline

- Motivation
  - Model-driven development MDD)
- Logic-labeled Finite State Machines
- Vectors of Logic-labeled Machines
- Case studies
  - The mine pump
  - The industrial press
- Conclusions

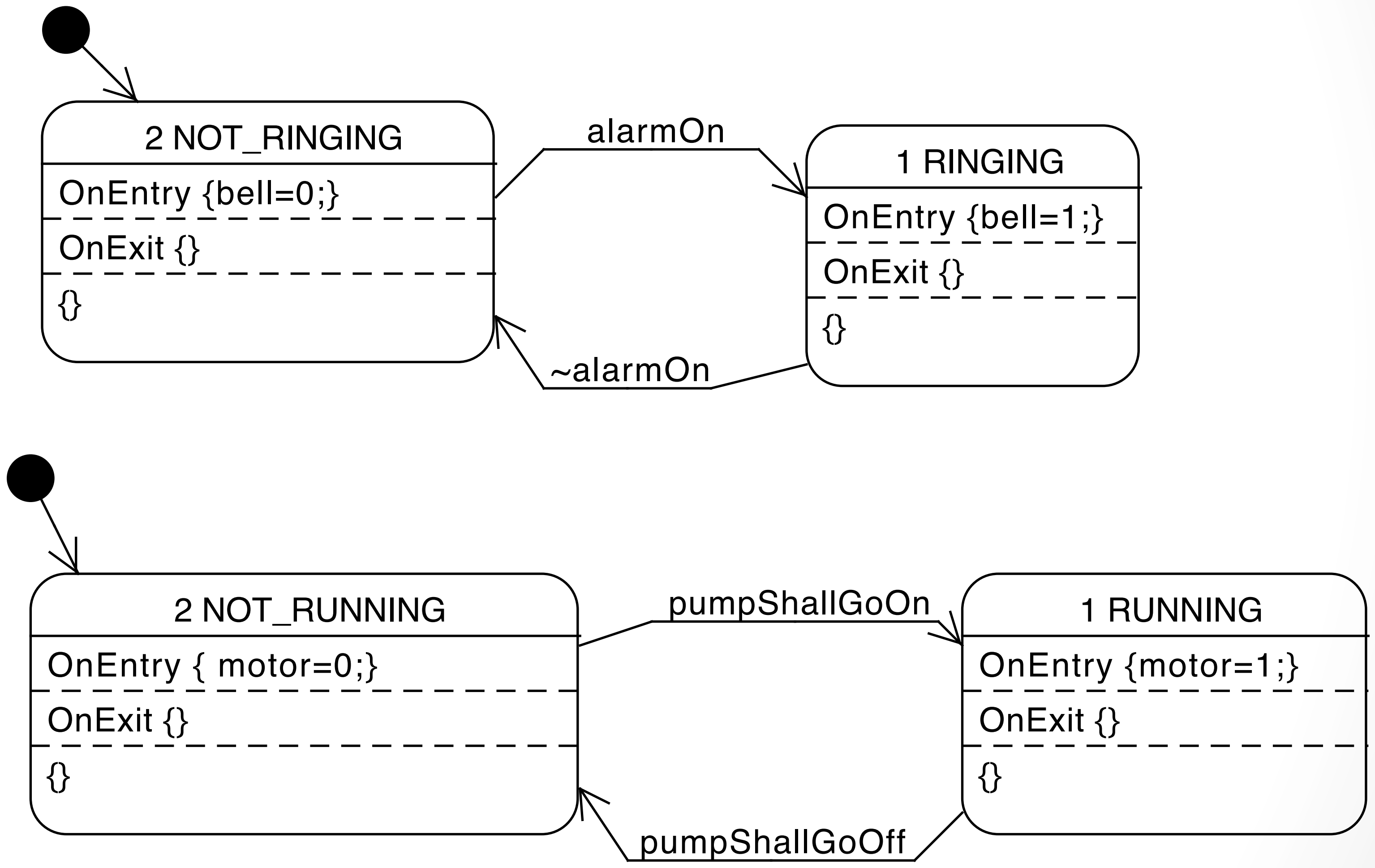


Mine Pump

Requirements	Description
R1	The pump extracts water from a mine shaft. When the water volume has been reduced below the low-water sensor, the pump is switched off. When the water raises above the high-water sensor it shall switch on.
R2	An human operator can switch the pump on and off provided the water level is between the high-water sensor and the low-water sensor.
R3	Another button accessed by a supervisor can switch the pump on and off independently of the water level.
R4	The pump will not turn on if the methane sensor detects a high reading.
R5	There are two other sensors, a carbon monoxide sensor and an air-flow sensor, and if carbon monoxide is high or air-flow is low, and alarm rings to indicate evacuation of the shaft.

Models are two FSMs

- the logic part not illustrated

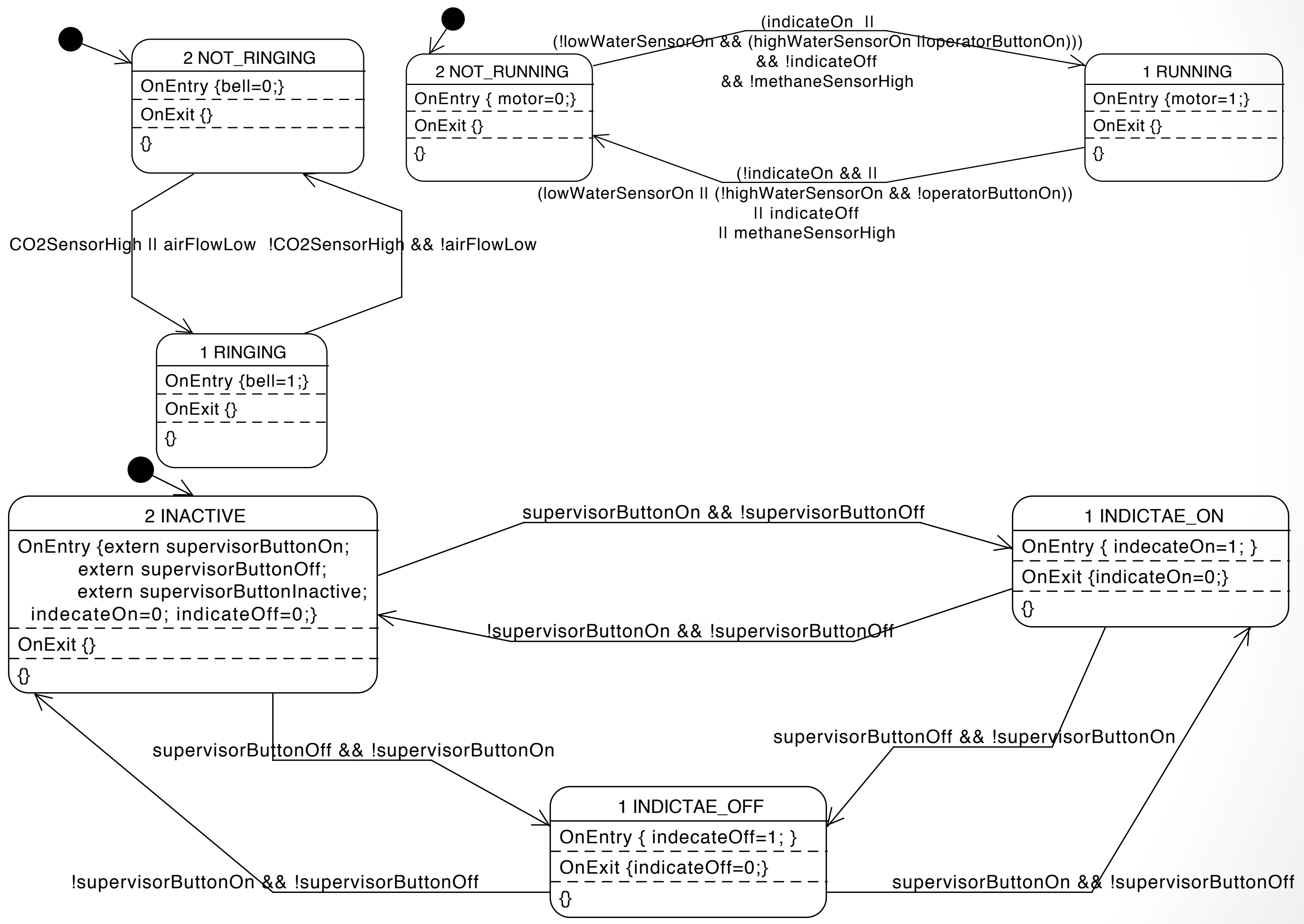


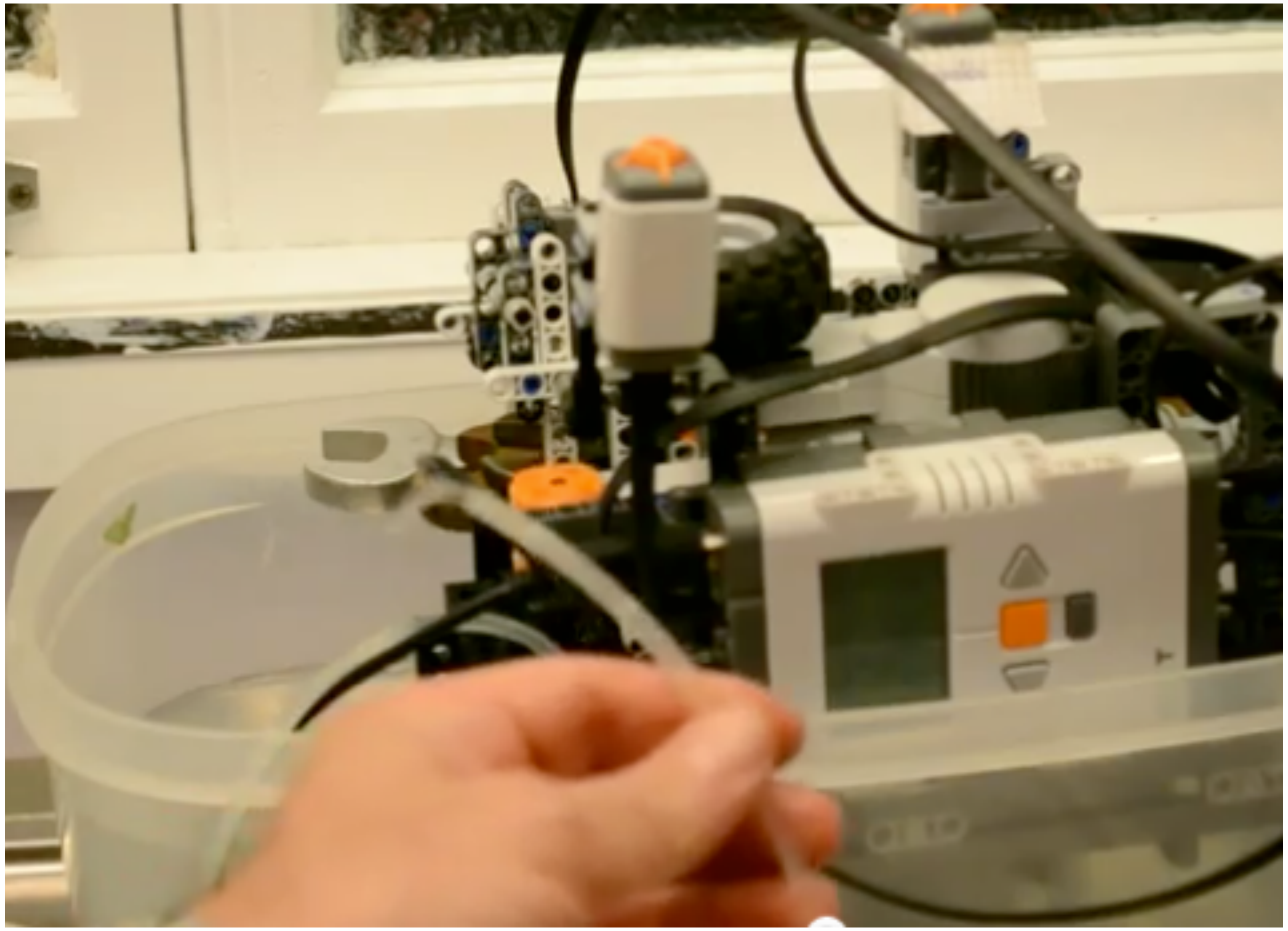
The logic part of the models

```
%Alarm.d

name{ALARM}.

input{CO2SensorHigh}.
input{airFlowLow}.

A0: {} =>  ~alarmOn.
A1: CO2SensorHigh =>  alarmOn. A1>A0.
A2: airFlowLow =>  alarmOn. A2>A0.

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

```
name{MINEPUMP}.
input{lowWaterSensorOn}.  input{highWaterSensorOn}.  input{operatorButtonOn}.
input{methaneSensorHigh}.  input{indicateOn}.  input{indicateOff}.

P0: {} =>  ~pumpShallGoOn.
P1: highWaterSensorOn =>  pumpShallGoOn.                                          P1>P0.
P2: lowWaterSensorOn =>  ~pumpShallGoOn.                                          P2>P1.
P3: {~lowWaterSensorOn,~highWaterSensorOn,operatorButtonOn}=> pumpShallGoOn.      P3>P2. P3>P0.
P4: {~lowWaterSensorOn,~highWaterSensorOn,~operatorButtonOn}=> ~pumpShallGoOn.    P4>P3.
P5: indicateOn => pumpShallGoOn.                                                  P5>P2. P5>P4. P5>P0.
P6: indicateOff => ~pumpShallGoOn.                                                P6>P5.
P7: methaneSensorHigh => ~pumpShallGoOn.                                          P7>P5. P7>P3. P7>P1.

N0: {} =>  ~pumpShallGoOff.
N1: {~indicateOn,lowWaterSensorOn} =>  pumpShallGoOff.                             N1>N0.
N2: {~indicateOn,~lowWaterSensorOn,~highWaterSensorOn,~operatorButtonOn}=> pumpShallGoOff.   N2>N0.
N3: indicateOff => pumpShallGoOff.                                                N3>N0.
N4: methaneSensorHigh => pumpShallGoOff.                                          N4>N0.

output{b pumpShallGoOn,"pumpShallGoOn"}.  output{b pumpShallGoOff,"pumpShallGoOff"}.
```

The complete model


2 NOT_RINGING
OnEntry {bell=0;}
OnExit {}
{}
CO2SensorHigh || airFlowLow
!CO2SensorHigh && !airFlowLow
1 RINGING
OnEntry {bell=1;}
OnExit {}
{}
2 NOT_RUNNING
OnEntry { motor=0;}
OnExit {}
{}
(indicateOn ||
(!lowWaterSensorOn && (highWaterSensorOn ||operatorButtonOn)))
&& !indicateOff
&& !methaneSensorHigh
1 RUNNING
OnEntry {motor=1;}
OnExit {}
{}
(!indicateOn && ||
(lowWaterSensorOn || (!highWaterSensorOn && !operatorButtonOn))
|| indicateOff
|| methaneSensorHigh
2 INACTIVE
OnEntry {extern supervisorButtonOn;
extern supervisorButtonOff;
extern supervisorButtonInactive;
indecateOn=0; indicateOff=0;}
OnExit {}
{}
supervisorButtonOn && !supervisorButtonOff
!supervisorButtonOn && !supervisorButtonOff
1 INDICTAE_ON
OnEntry { indecateOn=1; }
OnExit {indicateOn=0;}
{}
supervisorButtonOff && !supervisorButtonOn
supervisorButtonOff && !supervisorButtonOn
1 INDICTAE_OFF
OnEntry { indecateOff=1; }
OnExit {indicateOff=0;}
{}
!supervisorButtonOn && !supervisorButtonOff
supervisorButtonOn && !supervisorButtonOff



Demo video

http://www.youtube.com/watch?v=y4muLP0jA8U

Properties demonstrated by model-checking

Property-1 *“If* CO_2 *is high, the alarm must ring.”*

Property-2 *“If air flow is low, the alarm must ring.”*

Property-3 *“If methane levels are high, the pump must be turned off.”*

Property-4 *“If the supervisor switches off when running, the pump will be turned off.”*

Property-5 *“If the operator turns her switch off when the pump is running and the water level is neither low nor high, then the pump motor goes off.”*

Property-6 *“The pump is turned on when the water is above the high water sensor (and the low-water sensor’s signal is consistent with this), unless the supervisor turns it off or methane levels are high.”*

Property-7 *“If the supervisor sets the switch to inactive and the pump is running when the water is not above the high water sensor and the low-water sensor indicates a low level, the pump turns off.”*

Property-8 *“If there is low methane, low water, and the pump is not running, but the supervisor puts the switch to on, then the pump is turned on.”*

FMEA

- Perform fault-injection
  - Simulate the bell (although not part of the software)
    - everything that can go wrong here
      - missing transition
      - missing statement
      - incorrect statement

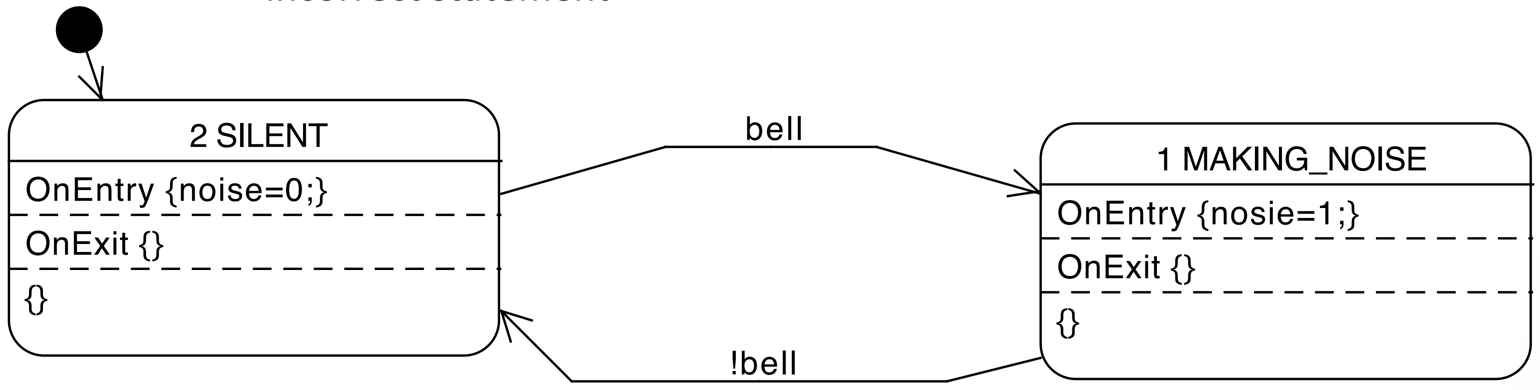
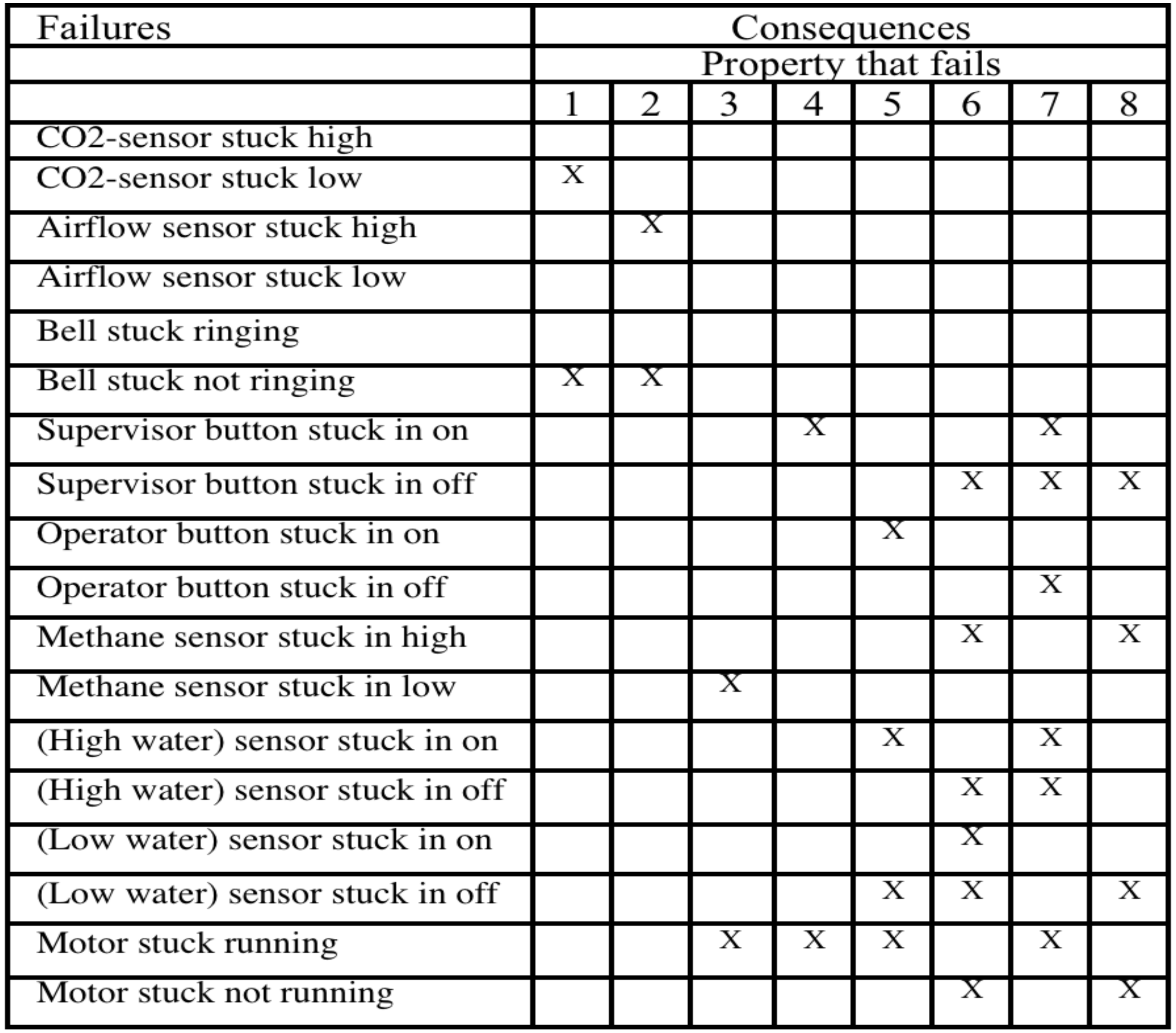


Table at level 1



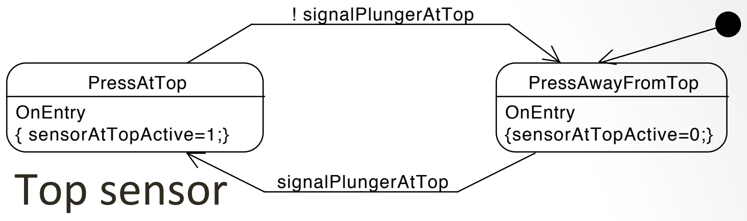
Failures	Consequences							
	Property that fails							
	1	2	3	4	5	6	7	8
CO2-sensor stuck high								
CO2-sensor stuck low	X							
Airflow sensor stuck high		X						
Airflow sensor stuck low								
Bell stuck ringing								
Bell stuck not ringing	X	X						
Supervisor button stuck in on				X			X	
Supervisor button stuck in off						X	X	X
Operator button stuck in on					X			
Operator button stuck in off							X	
Methane sensor stuck in high						X		X
Methane sensor stuck in low			X					
(High water) sensor stuck in on					X		X	
(High water) sensor stuck in off						X	X	
(Low water) sensor stuck in on						X		
(Low water) sensor stuck in off					X	X		X
Motor stuck running			X	X	X		X	
Motor stuck not running						X		X

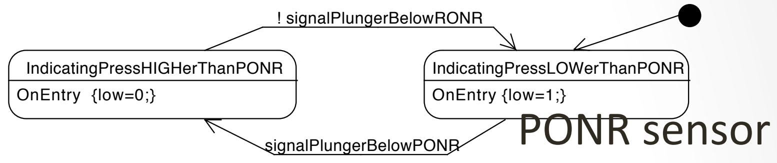
Industrial Press Requirements

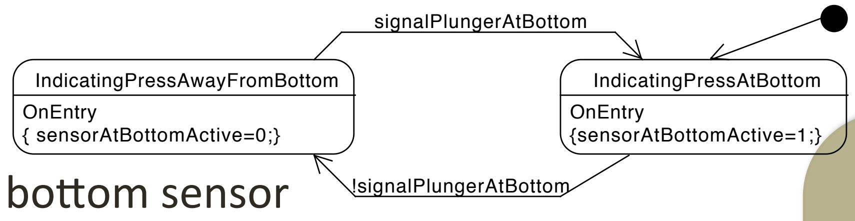
Requirements	Description
R1	The plunger is initially resting at the bottom with the motor off.
R2	When power is supplied, the controller shall turn the motor on, causing the plunger to rise.
R3	When at the top, the plunger shall be held there until the operator pushes and holds down the button. This shall cause the controller to turn the motor off and the plunger will begin to fall.
R4	If the operator releases the button while the plunger is falling slowly (above PONR), the controller shall turn the motor on again, causing the plunger to start rising again, without reaching the bottom.
R5	If the plunger is falling fast (below PONR) then the controller shall leave the motor off until the plunger reaches the bottom.
R6	When the plunger is at the bottom the controller shall turn the motor on: the plunger will rise again.

The complete model

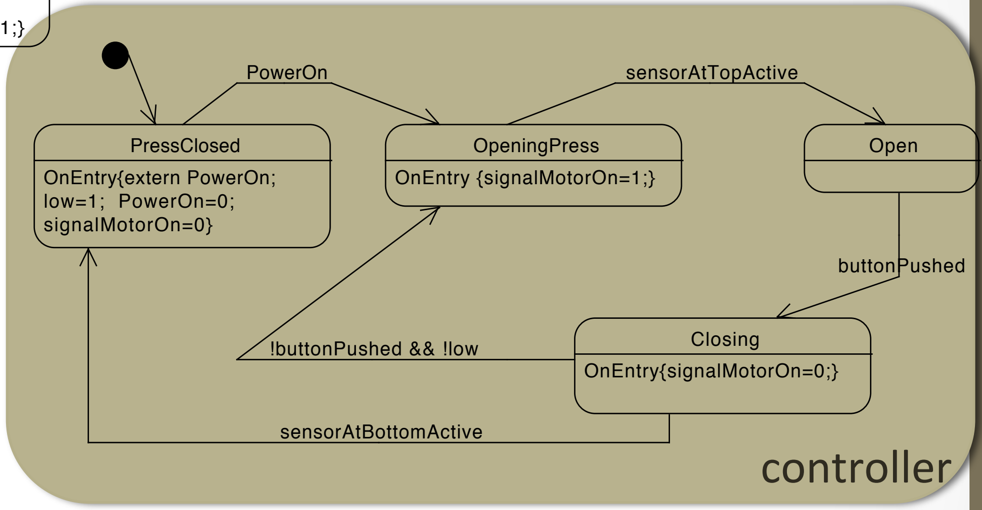
--- with peripherals for model checking and FMEA

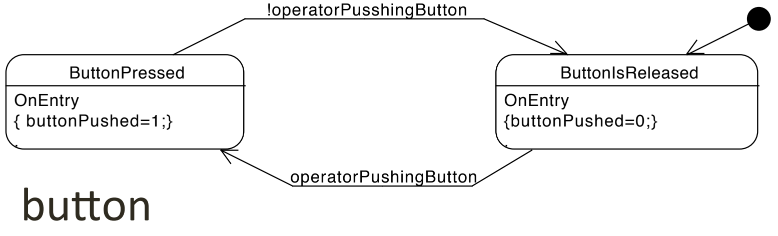


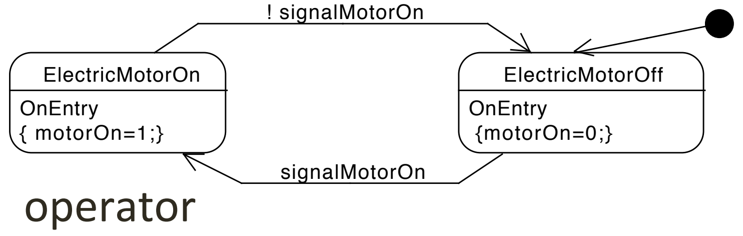


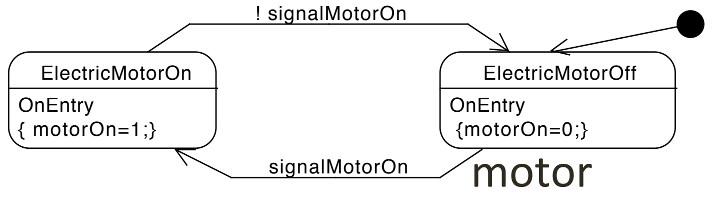


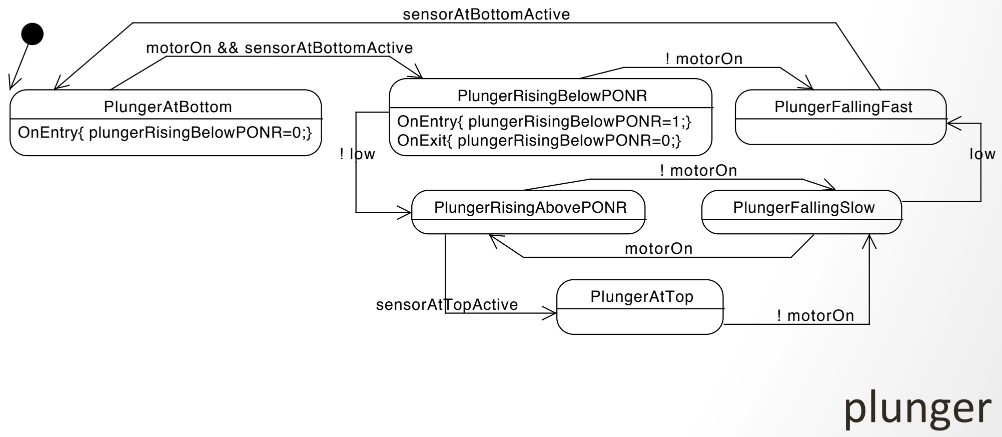
signalPlungerAtBottom
IndicatingPressAwayFromBottom
OnEntry { sensorAtBottomActive=0;}
IndicatingPressAtBottom
OnEntry {sensorAtBottomActive=1;}
!signalPlungerAtBottom
bottom sensor



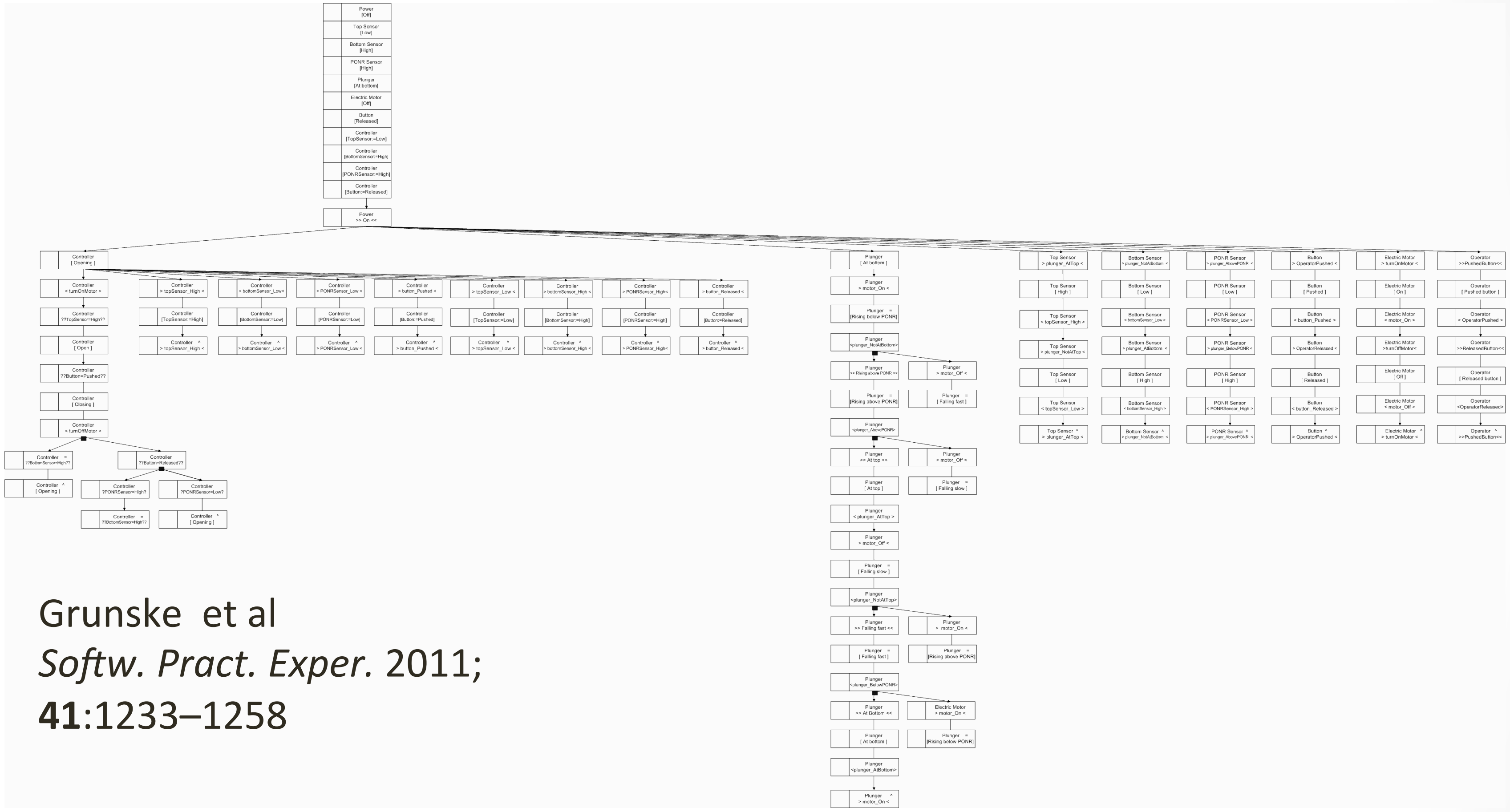






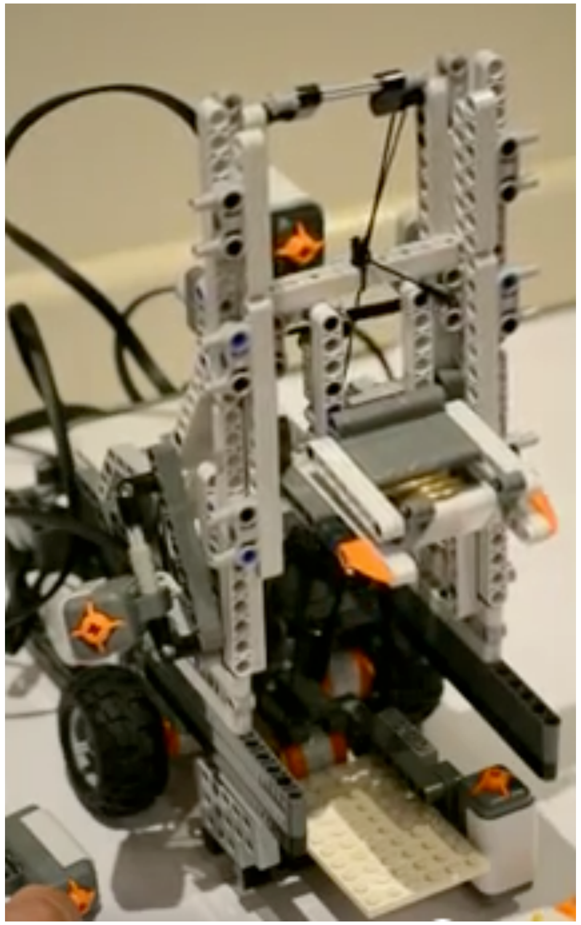


Contrast with Behavior Trees



Grunske et al
Softw. Pract. Exper. 2011;
41:1233–1258

Incorrect modeling of sequence of events after the press falls down



Demo

http://www.youtube.com/watch?v=blUpMdH14pM

Properties demonstrated by model-checking

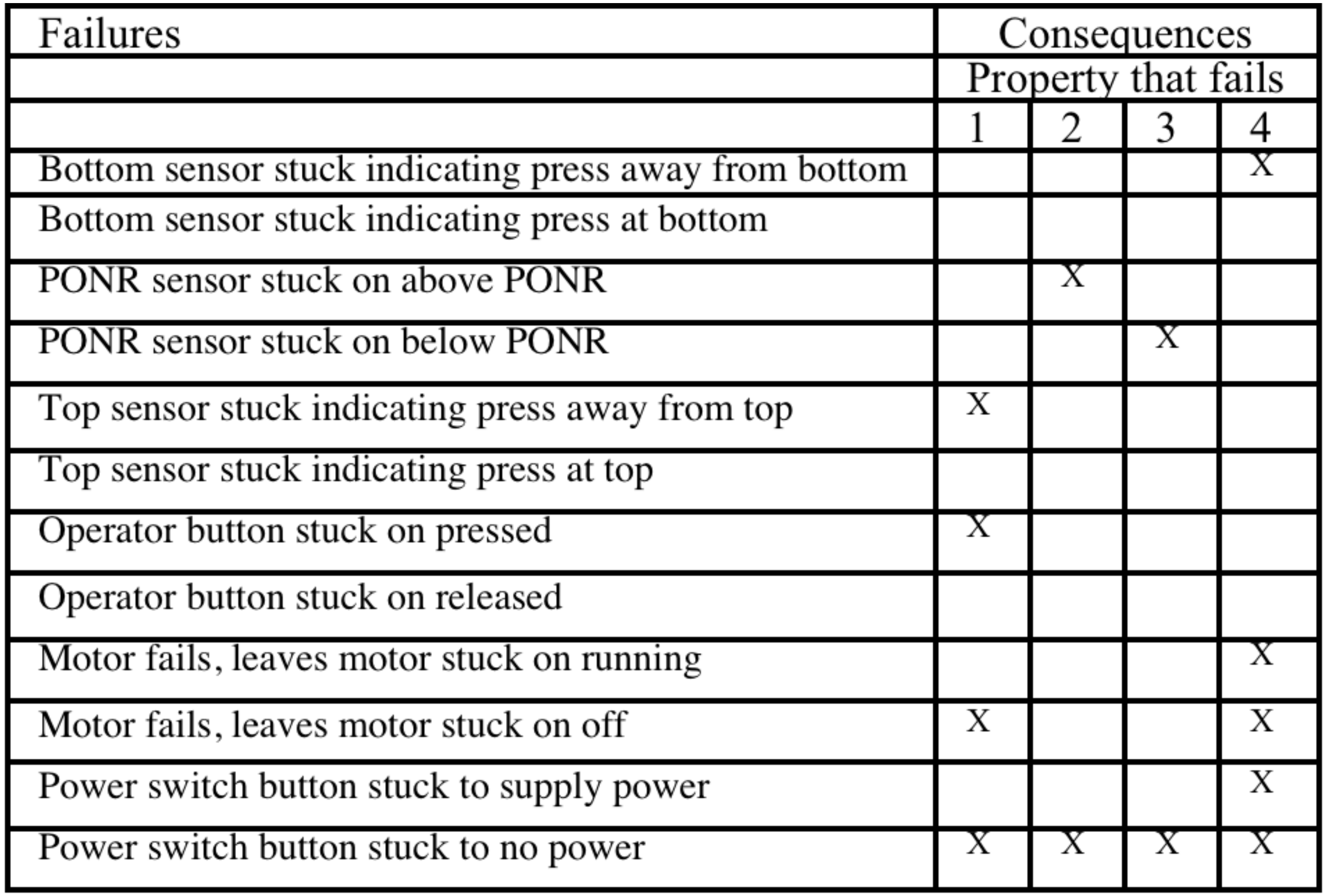
Property-1 *"If the operator is not pushing the button and the plunger is at the top, the motor should remain on".*

Property-2 *"If the plunger is falling below the PONR, a state modeled by the plunger falling fast, then the motor should remain off."*

Property-3 *"If the plunger is falling above the PONR, a state modeled by falling slow, and the operator releases the button, the motor should turn on, before the plunger changes state."*

Property-4 *"Once the plunger is down, a new signal is needed to turn the motor on and raise the plunger again."*

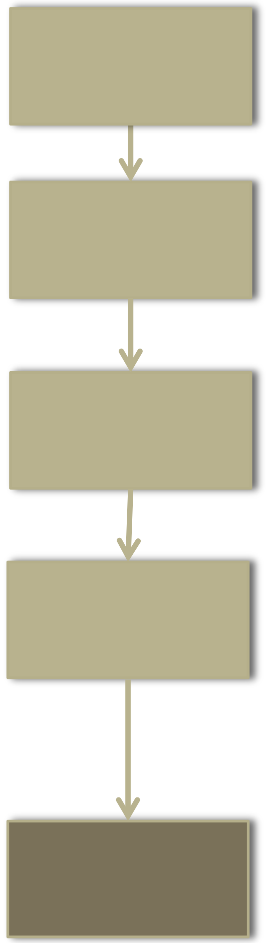
Table level 1



Failures	Consequences			
	Property that fails			
	1	2	3	4
Bottom sensor stuck indicating press away from bottom				X
Bottom sensor stuck indicating press at bottom				
PONR sensor stuck on above PONR		X		
PONR sensor stuck on below PONR			X	
Top sensor stuck indicating press away from top	X			
Top sensor stuck indicating press at top				
Operator button stuck on pressed	X			
Operator button stuck on released				
Motor fails, leaves motor stuck on running				X
Motor fails, leaves motor stuck on off	X			X
Power switch button stuck to supply power				X
Power switch button stuck to no power	X	X	X	X

Outline

- Motivation
  - Model-driven development MDD)
- Logic-labeled Finite State Machines
- Vectors of Logic-labeled Machines
- Case studies
  - The mine pump
  - The industrial press
- Conclusions



Simulation, Model-checking and FMEA

- Ensure quality and safety
  - software in embedded controllers
- Logic-labeled vectors of FSM, sequentially scheduled
  - provide more succinct models
    - validated
  - with clear semantics
  - that
    - can be simulated
    - can be exported to various platforms
      - (model-driven development)
    - can be model-checked
      - (in a matter of seconds, as opposed to days of CPU time)
    - can be examined with fault-injection

Thank you

Any Questions?