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 INTERMIDIAE 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*© *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).

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

---

(c) Vlad Estivill-Castro
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?
- is the game over?
- I am injured?
- did the team lost possession?
Example from robotic soccer

% 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"}.

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 > toleranceRatio;
  badDensityVsDensityTolerance =
    blobArea / blobNumPixels > densityTolerance;
}

OnExit {}

BALL_FOUND

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

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>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.</td>
</tr>
<tr>
<td>R2</td>
<td>If the button is pushed while the oven is cooking, it will cause the oven to cook for an extra minute.</td>
</tr>
<tr>
<td>R3</td>
<td>Pushing the button when the door is open has no effect.</td>
</tr>
<tr>
<td>R4</td>
<td>Whenever the oven is cooking or the door is open, the light in the oven will be on.</td>
</tr>
<tr>
<td>R5</td>
<td>Opening the door stops the cooking. and stops the timer and does not clear the timer</td>
</tr>
<tr>
<td>R6</td>
<td>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.</td>
</tr>
<tr>
<td>R7</td>
<td>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.</td>
</tr>
</tbody>
</table>
One of the FSMs

% MicrowaveCook.d

name{MicrowaveCook}.

input{timeLeft}.
input{doorOpen}.

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

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

(c) Vlad Estivill-Castro
The complete arrangement
Demo video

http://www.youtube.com/watch?v=t4uel1o67Xk&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

(c) Vlad Estivill-Castro
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

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R1</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>R2</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>R3</strong></td>
<td>Another button accessed by a supervisor can switch the pump on and off independently of the water level.</td>
</tr>
<tr>
<td><strong>R4</strong></td>
<td>The pump will not turn on if the methane sensor detects a high reading.</td>
</tr>
<tr>
<td><strong>R5</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
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.
A2: airFlowLow => alarmOn. A2>A0.

output{b alarmOn,"alarmOn"}.

%MINEPUMP.d
name(MINEPUMP).
input{lowWaterSensorOn}.
input{highWaterSensorOn}.
input{operatorButtonOn}.
input{methaneSensorHigh}.
input{indicateOn}.
input{indicateOff}.

P0: {} => ~pumpShallGoOn.
P1: highWaterSensorOn => pumpShallGoOn.
P5: indicateOn => pumpShallGoOn.
P6: indicateOff => ~pumpShallGoOn.
P7: methaneSensorHigh => ~pumpShallGoOn.

N0: {} => ~pumpShallGoOff.
N1: (~indicateOn,lowWaterSensorOn) => pumpShallGoOff.
N3: indicateOff => pumpShallGoOff.
N4: methaneSensorHigh => pumpShallGoOff.

output{b pumpShallGoOn,"pumpShallGoOn"}. output{b pumpShallGoOff,"pumpShallGoOff"}.
The complete model

CO2SensorHigh \&\& airFlowLow \&\& !CO2SensorHigh

1 RINGING
OnEntry {bell=1;}
OnExit {0}

2 INACTIVE
OnEntry {extern supervisorButtonOn; extern supervisorButtonOff; extern supervisorButtonInactive; indicateOn=0; indicateOff=0;}
OnExit {0}

supervisorButtonOff \&\& !supervisorButtonOn

1 INDICTAE_ON
OnEntry {indicateOn=1;}
OnExit {indicateOn=0;}

supervisorButtonOn \&\& !supervisorButtonOff

1 INDICTAE_OFF
OnEntry {indicateOff=1;}
OnExit {indicateOff=0;}

supervisorButtonOn \&\& !supervisorButtonOff

supervisorButtonOff \&\& !supervisorButtonOn

1 RUNNING
OnEntry {motor=1;}
OnExit {0}

(indicateOn \&\& (lowWaterSensorOn \&\& (highWaterSensorOn \&\& operatorButtonOn)) \&\& !indicateOff \&\& !methaneSensorHigh

![Diagram](image-url)
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

```plaintext
2 SILENT
OnEntry {noise=0;}
OnExit {}
{}

1 MAKING_NOISE
OnEntry {noise=1;}
OnExit {}
{}
```

(c) Vlad Estivill-Castro
<table>
<thead>
<tr>
<th>Failures</th>
<th>Consequences Property that fails</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>CO2-sensor stuck high</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CO2-sensor stuck low</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Airflow sensor stuck high</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Airflow sensor stuck low</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bell stuck ringing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bell stuck not ringing</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Supervisor button stuck in on</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Supervisor button stuck in off</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operator button stuck in on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operator button stuck in off</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Methane sensor stuck in high</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Methane sensor stuck in low</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>(High water) sensor stuck in on</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(High water) sensor stuck in off</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Low water) sensor stuck in on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Low water) sensor stuck in off</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Motor stuck running</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Motor stuck not running</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Industrial Press Requirements

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>The plunger is initially resting at the bottom with the motor off.</td>
</tr>
<tr>
<td>R2</td>
<td>When power is supplied, the controller shall turn the motor on, causing the plunger to rise.</td>
</tr>
<tr>
<td>R3</td>
<td>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.</td>
</tr>
<tr>
<td>R4</td>
<td>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.</td>
</tr>
<tr>
<td>R5</td>
<td>If the plunger is falling fast (below PONR) then the controller shall leave the motor off until the plunger reaches the bottom.</td>
</tr>
<tr>
<td>R6</td>
<td>When the plunger is at the bottom the controller shall turn the motor on: the plunger will rise again.</td>
</tr>
</tbody>
</table>
The complete model

--- with peripherals for model checking and FMEA

**bottom sensor**

- Indicating Press Away From Bottom
  - On Entry: \( \text{sensorAtBottomActive} = 0 \)

- Indicating Press At Bottom
  - On Entry: \( \text{sensorAtBottomActive} = 1 \)

**button**

- Button Pressed
  - On Entry: \( \text{buttonPushed} = 1 \)

- Button Is Released
  - On Entry: \( \text{buttonPushed} = 0 \)

**operator**

- Operator Pushing Button
  - On Entry: \( \text{operatorPushingButton} \)

**motor**

- Electric Motor On
  - On Entry: \( \text{motorOn} = 1 \)

- Electric Motor Off
  - On Entry: \( \text{motorOn} = 0 \)

**controller**

- Power On
  - On Entry: \( \text{sensorAtTopActive} = 1 \)

- Opening Press
  - On Entry: \( \text{signalMotorOn} = 1 \)

- Closing
  - On Entry: \( \text{sensorAtBottomActive} = 0 \)

- Press At Top
  - On Entry: \( \text{sensorAtTopActive} = 1 \)

- Press Away From Top
  - On Entry: \( \text{sensorAtTopActive} = 0 \)

**plunger**

- Plunger At Bottom
  - On Entry: \( \text{plungerRisingBelowPONR} = 0 \)

- Plunger Rising Below PONR
  - On Entry: \( \text{plungerRisingBelowPONR} = 1 \)

- Plunger Rising Above PONR
  - On Entry: \( \text{plungerRisingAbovePONR} = 1 \)

- Plunger Falling Fast
  - On Exit: \( \text{plungerRisingBelowPONR} = 0 \)

- Plunger Falling Slow
  - On Exit: \( \text{plungerRisingBelowPONR} = 0 \)

- Plunger At Top
  - On Entry: \( \text{plungerAtTop} \)

- ! signal Plunger At Bottom
  - On Entry: \( \text{signalPlungerAtBottom} \)

- button Pushed & ! low
  - On Entry: \( \text{buttonPushed} \)

- sensor At Bottom Active
  - On Entry: \( \text{sensorAtBottomActive} \)

- ! signal Motor On
  - On Entry: \( \text{signalMotorOn} \)

- ! signal Motor Off
  - On Entry: \( \text{signalMotorOn} \)
Contrast with Behavior Trees

Incorrect modeling of sequence of events after the press falls down

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

<table>
<thead>
<tr>
<th>Failures</th>
<th>Consequences Property that fails</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Bottom sensor stuck indicating press away from bottom</td>
<td></td>
</tr>
<tr>
<td>Bottom sensor stuck indicating press at bottom</td>
<td></td>
</tr>
<tr>
<td>PONR sensor stuck on above PONR</td>
<td>X</td>
</tr>
<tr>
<td>PONR sensor stuck on below PONR</td>
<td></td>
</tr>
<tr>
<td>Top sensor stuck indicating press away from top</td>
<td>X</td>
</tr>
<tr>
<td>Top sensor stuck indicating press at top</td>
<td></td>
</tr>
<tr>
<td>Operator button stuck on pressed</td>
<td>X</td>
</tr>
<tr>
<td>Operator button stuck on released</td>
<td></td>
</tr>
<tr>
<td>Motor fails, leaves motor stuck on running</td>
<td></td>
</tr>
<tr>
<td>Motor fails, leaves motor stuck on off</td>
<td>X</td>
</tr>
<tr>
<td>Power switch button stuck to supply power</td>
<td></td>
</tr>
<tr>
<td>Power switch button stuck to no power</td>
<td>X</td>
</tr>
</tbody>
</table>
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?