Modelling and Analysing a Mechanical Lung Ventilator in mCRL2

Danny van Dortmont Tim Willemse Jeroen Keiren

27 June 2024 (ABZ 2024)

Department of Mathematics & Computer Science

Introduction

- Mechanical Lung Ventilator (MLV) Case study for ABZ 2024
- Simplified but realisitic requirements document
- Modelling and verification based on requirements using mCRL2

Source: `https://www.umbriaecultura.it/mvm-milano-ventilatore-meccanico/`
CC-BY-NC-ND 4.0

mCRL2 by Example

Data and Processes

Data types

mCRL2 by Example

Data and Processes

Data types

- `Bool`, `Nat`, `List(S)`, ... predefined
- Used-defined types:

```
sort SensorState = struct Working | Error | sFaulty;
```

mCRL2 by Example

Data and Processes

Data types

- `Bool`, `Nat`, `List(S)`, ... predefined
- Used-defined types:

```
sort SensorState = struct Working | Error | sFaulty;
```

Processes:

-
  ```
  act get_var_r: Nat;
      set_var_r: Nat;
  proc M(var: Bool) =
          get_var_r(var) . M()
        + sum b: Bool . set_var_r(b)
          . M(var = b);
  init M(false);
  ```

mCRL2 by Example

Data and Processes

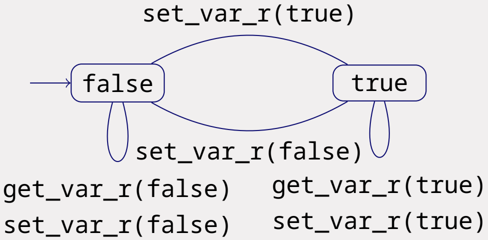
Data types

- Bool, Nat, List(S), ... predefined
- Used-defined types:

```
sort SensorState = struct Working | Error | sFaulty;
```

Processes:

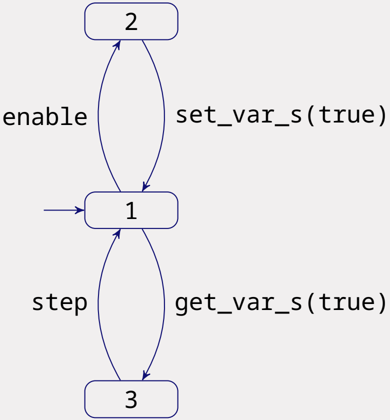
-
  ```
  act get_var_r: Nat;
      set_var_r: Nat;
  proc M(var: Bool) =
          get_var_r(var) . M()
        + sum b: Bool . set_var_r(b)
          . M(var = b);
  init M(false);
  ```



mCRL2 by Example

Processes

```
act enable, step;
    get_var_s: Nat;
    set_var_s: Nat;
proc C() =
       enable . set_var_s(true) . C()
     + get_var_s(true) . step . C();
init C();
```



mCRL2 by Example

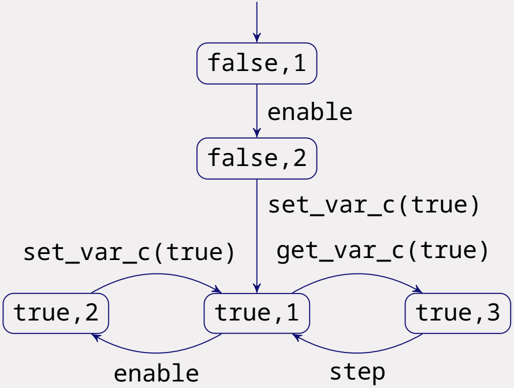
Communicating Processes

```
act enable, step;
    get_var_r, get_var_s, get_var_c: Nat;
    get_var_s, set_var_s, set_var_c: Nat;
proc C() =
       enable . set_var_s(true) . C()
     + get_var_s(true) . step . C();
     M(var: Bool) =
       get_var_r(var) . M()
     + sum b: Bool . set_var_r(b) . M(var = b);
init allow({enable, step, get_var, set_var},
       comm({get_var_r|get_var_s
                -> get_var_c,
              set_var_r|set_var_s
                -> set_var_c},
             C() || M(false)));
```

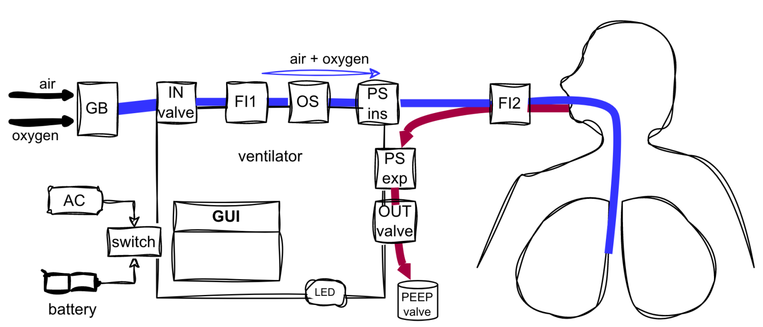
mCRL2 by Example

Communicating Processes

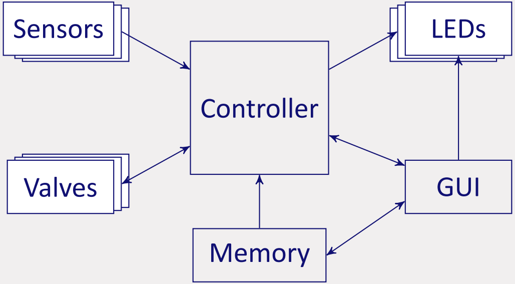
```
act enable, step;
    get_var_r, get_var_s, get_var_c: Nat;
    get_var_s, set_var_s, set_var_c: Nat;
proc C() =
       enable . set_var_s(true) . C()
     + get_var_s(true) . step . C();
     M(var: Bool) =
       get_var_r(var) . M()
     + sum b: Bool . set_var_r(b) . M(var = b);
init allow({enable, step, get_var, set_var},
       comm({get_var_r|get_var_s
                -> get_var_c,
              set_var_r|set_var_s
                -> set_var_c},
             C() || M(false)));
```



Mechanical Lung Ventilator



Architecture



Formalizing MLV in mCRL2

Memory and Alarms

- `Memory` modelled as process `M`
  - One process parameter for each configuration parameter
  - Only allow setting valid values, e.g.

    ```
    sum v: Nat . (4 <= v && v <= 50) -> set_RR_PCV_r(v) ...
    ```

Formalizing MLV in mCRL2

Memory and Alarms

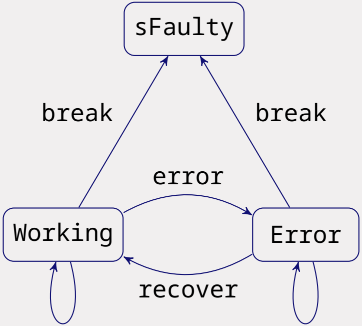
- Memory modelled as process `M`
  - One process parameter for each configuration parameter
  - Only allow setting valid values, e.g.
    `sum v: Nat . (4 <= v && v <= 50) -> set_RR_PCV_r(v) ...`
- LEDs triggered from `Controller` synchronizing on `alarm_r` action:

```
proc LEDs = VisualAlarms(false,false,false);
     VisualAlarms(low, medium, high: Bool) =
         alarm_r(Low). VisualAlarms(low = true)
       + ...
       + snooze_alarm_r(Low). VisualAlarms(low = false)
       + ...
       + low -> LowAlarm. VisualAlarms()
     + ...
```

Formalizing MLV in mCRL2

Sensors and Valves

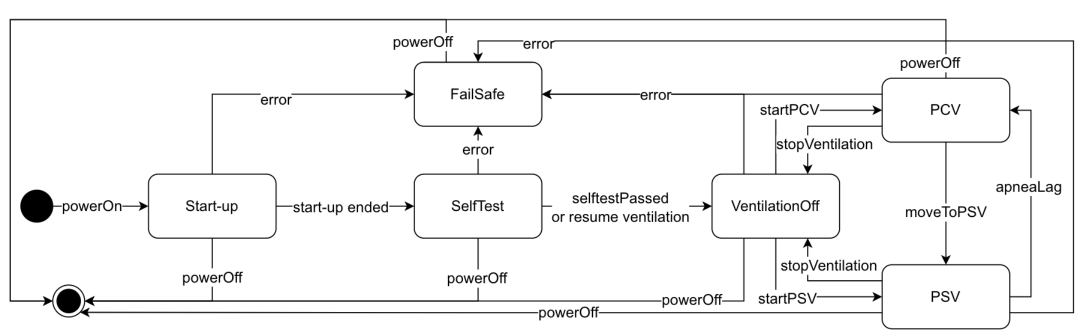
```
Sensor(id: SensorId, state: SensorState,
       currVal: Int, validValues: List(Int)) =
    sum value: Int .
      (value in validValues && state == Working)
        -> updateSensorValue(id, value)
      . Sensor(currVal = value)
  + (state != sFaulty)
    -> getSensorState_r(id, state) . Sensor()
  + (state == Working)
    -> error. Sensor(state=Error)
  + (state == Error)
    -> recover. Sensor(state=Working)
  + (state != sFaulty)
    -> break. Sensor(state=sFaulty)
  + (state != sFaulty)
    -> getSensorValue_r(id, currVal)
       . Sensor();
```



getSensorState_r, getSensorState_r,
getSensorValue_r, getSensorValue_r
updateSensorValue

Controller and GUI

- UML state machine diagram
- Further restrictions from textual requirements



Controller

Design choices:

- Every mode is separate process
- Signal mode changes by going through `Setup` process

```
Controller_StartUp_Setup =
  powerOff . ControllerSwitcher(Stop)
+ setValveState_s(In, Closed)
  . setValveState_s(Out, Open)
  . emitMode_s(StartUp)
  . Controller_StartUp(InitialSensorStatus, InitialValveStatus, 0, 0, true);
```

- Details follow requirements
- PCV, PSV modes: abstract from actual ventilation control

Controller and GUI

Enabled GUI actions depend on Controller Mode

Controller and GUI

Enabled GUI actions depend on Controller Mode

Example: Switch from PSV to PCV mode onlly allowed if MLV is in PSV mode

Controller and GUI

Enabled GUI actions depend on Controller Mode

Example: Switch from PSV to PCV mode onlly allowed if MLV is in PSV mode

Problem: Reading controller mode from GUI then switching . . . . race condition

Controller and GUI

Enabled GUI actions depend on Controller Mode

Example: Switch from PSV to PCV mode onlly allowed if MLV is in PSV mode

Problem: Reading controller mode from GUI then switching . . . . race condition

Solution

Some actions happen simultaneously with reading controller mode using multi-actions.

Controller and GUI

Enabled GUI actions depend on Controller Mode

Example: Switch from PSV to PCV mode onlly allowed if MLV is in PSV mode

Problem: Reading controller mode from GUI then switching . . . . race condition

Solution

Some actions happen simultaneously with reading controller mode using multi-actions.

```
ExposeControllerMode(mode: OperationMode) =
    sum m: OperationMode. emitMode_r(m). ExposeControllerMode(mode = m)
+ controller_Mode_s(mode). ExposeControllerMode()
+ emitMode(mode). ExposeControllerMode();
```

Controller and GUI

Enabled GUI actions depend on Controller Mode

Example: Switch from PSV to PCV mode onlly allowed if MLV is in PSV mode

Problem: Reading controller mode from GUI then switching . . . . race condition

Solution

Some actions happen simultaneously with reading controller mode using multi-actions.

```
ExposeControllerMode(mode: OperationMode) =
        sum m: OperationMode. emitMode_r(m). ExposeControllerMode(mode = m)
+ controller_Mode_s(mode). ExposeControllerMode()
+ emitMode(mode). ExposeControllerMode();
```

- Synchronize with Controller on `emitMode_s`
- Synchronize with GUI on `controller_Mode_s`
- `emitMode` for verification purposes

Formalizing MLV in mCRL2

State space size

- Induced LTS contains $1.5 \cdot 10^{23}$ reachable states
- Symbolic reachability in mCRL2 in 13 seconds

Verification

- Formalize requirements and scenarios using modal μ-calculus
- Verify using mCRL2's symbolic model checker

Example (Cont.38)

> "when the ventilator is in Start-up or VentilationOff mode the valve pressure shall be set to close and the out valve shall be open".

Verification

- Formalize requirements and scenarios using modal μ-calculus
- Verify using mCRL2's symbolic model checker

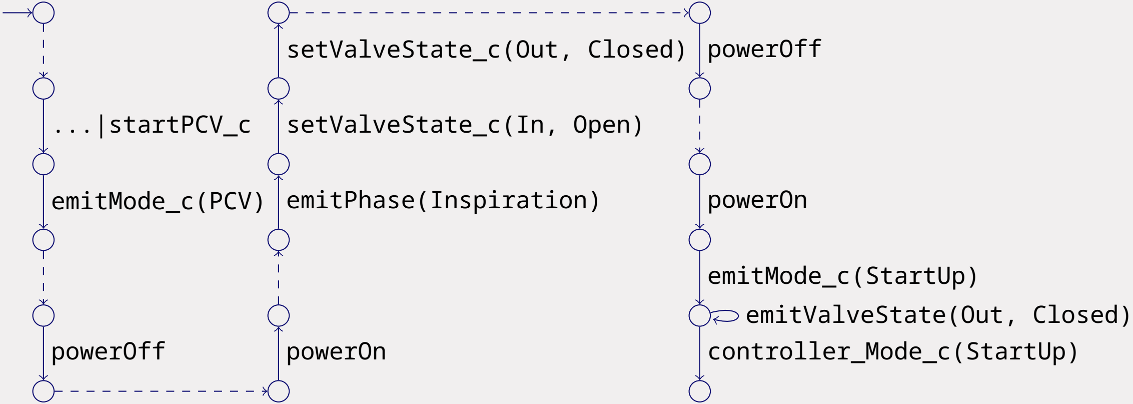
Example (Cont.38)

"when the ventilator is in Start-up or VentilationOff mode the valve pressure shall be set to close and the out valve shall be open".

```
[true*]
  (<emitMode(VentilationOff) || emitMode(StartUp)>true
  =>
   [emitValveState(In,Open) || emitValveState(Out,Closed)] false
  )
```

Verification

Early version of model violated requirement


...|startPCV_c
emitMode_c(PCV)
powerOff
powerOn
emitPhase(Inspiration)
setValveState_c(In, Open)
setValveState_c(Out, Closed)
powerOff
powerOn
emitMode_c(StartUp)
emitValveState(Out, Closed)
controller_Mode_c(StartUp)

Alarms

> *“The system shall raise an alarm when the inspiratory flux is below a user-controlled value ($MinV_E$).” [SAV.16]*

Strong interpretation: high priority alarm is unavoidable . . . . does not hold due to self-loops

Weaker version: so long as the high priority alarm has not yet been raised or snoozed, it remains possible

```
[true*]
    (forall v,w:Nat. [getSensorValue_c(FlowIndicator1,v)|get_Min_V_E_c(w)]
        (val(v < w) =>
            [!( alarm_snooze_c(High) || HighAlarm )*]<true*.HighAlarm>true)))
```

Observations Requirement Document

- Unclear how controller and GUI are supposed to work together. Do they synchronize?
  - GUI can crash but ventilation should continue $\Longrightarrow$ no strict synchronization
  - Setting parameters done from the GUI, but where should the data be stored?
  - Decoupling GUI from controller $\Longrightarrow$ race conditions
- When do we consider GUI/Controller to be in a state?
  - When are valves set to safe mode? . . . . at least at power off?

Conclusions & Future Work

Conclusions:

- Natural language requirements ambiguous (even if requirements documents are fairly detailed)
- Faithful modelling of discrete behavior in mCRL2
- Abstraction of continuous behavior using nondeterminism
- Symbolic model checker essential

Conclusions & Future Work

Conclusions:

- Natural language requirements ambiguous (even if requirements documents are fairly detailed)
- Faithful modelling of discrete behavior in mCRL2
- Abstraction of continuous behavior using nondeterminism
- Symbolic model checker essential

Future work:

- Extend model with additional details
- Support verification of continuous behavior
- Counterexamples for symbolic model checking

Questions?

mCRL2

analysing system behaviour

- mCRL2: https://www.mcrl2.org
- Models: https://dx.doi.org/10.5281/zenodo.10978852