Tutorial 9 — Building Controllers in VDM

Overview

This tutorial will help you to:

1. Generate a more complete controller in VDM using Overture
2. Add behaviours to deal with realistic behaviours (noise and ambient light)
3. Add modes for degraded behaviours after sensor failure

Requirements

This tutorial requires the following tools from the INTO-CPS tool chain to be installed:

- INTO-CPS Application
- COE (Co-simulation Orchestration Engine) accessible to the Application
- Overture including FMU plug-in

Tools can be downloaded through the Application (*Window > Show Download Manager*) or may have been provided on the USB drive at your training session. Please ask if you are unsure.

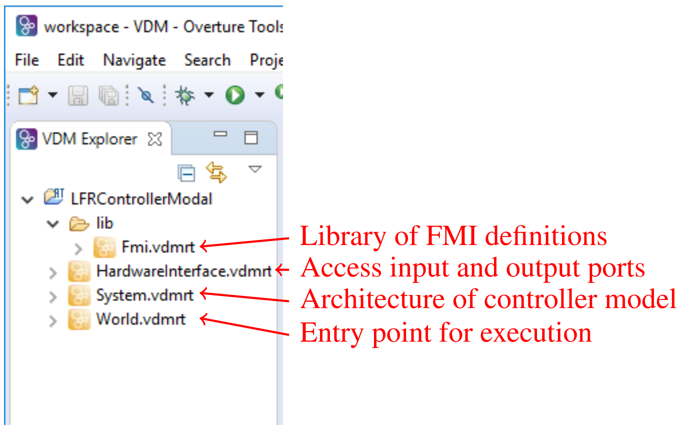
1 Make an Overture Project

We will begin by creating a project in Overture and importing a model description file as in Tutorial 4.

Step 1. Create a project in Overture called *LFRControllerModal*. If you are unsure, follow Steps 1–6 of Tutorial 4. You can accept the default location (your Overture workspace) in Step 4, or make a folder in your *tutorial_9/Models* directory and place the project there.

Step 2. Import *tutorial_9/Models/Controller.modeldescription.xml* using the *Overture FMU > Import Model Description* context menu, as in Steps 7–9 of Tutorial 4.

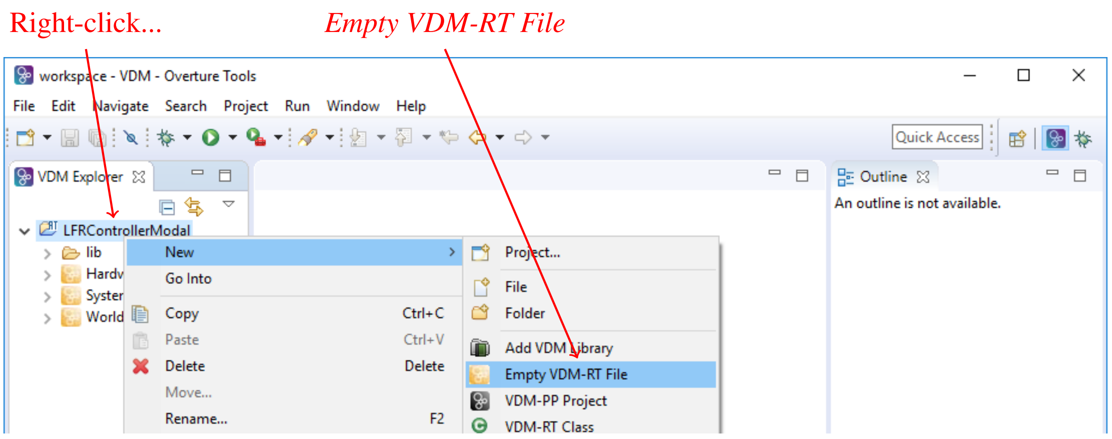
You should see the following structure:


workspace - VDM - Overture Tools
File Edit Navigate Search Proje
VDM Explorer
LFRControllerModal
lib
Fmi.vdmrt
HardwareInterface.vdmrt
System.vdmrt
World.vdmrt
Library of FMI definitions
Access input and output ports
Architecture of controller model
Entry point for execution

2 Adding Structure and a Skeleton Controller

To make a functional controller, we will we will add a *Controller* class to contain the control logic. To provide structure to help with more complicated logic later, we will add classes to represent the *sensors* and *actuators* that manage the interface to the COE (and the physical simulation made in 20-sim). The *Controller* class will use objects of these classes to read the environment and control the robot.

Step 3. Right-click on the *LFRControllerModal* project and select *New* > *Empty VDM-RT File*. Call it *Controller* and click *Finish*.



Step 4. Paste in the following listing and click *Save*.

```
class Controller

instance variables

leftSensor: IRSensor;
rightSensor: IRSensor;

leftServo: Servo;
rightServo: Servo;

operations

public Controller: IRSensor * IRSensor * Servo * Servo ==> Controller
Controller(lfl, lfr, ls, rs) == (
  leftSensor := lfl;
  rightSensor := lfr;
  leftServo := ls;
  rightServo := rs
);

Step: () ==> ()
Step() == cycles(20) (
    -- debug information
    IO`printf("Left sensor: %s (%s), right sensor: %s (%s)\n",
      [leftSensor.getReading(),leftSensor.hasFailed(),
       rightSensor.getReading(),rightSensor.hasFailed()]);
);

thread

periodic(10E6, 0, 0, 0)(Step)

end Controller
```

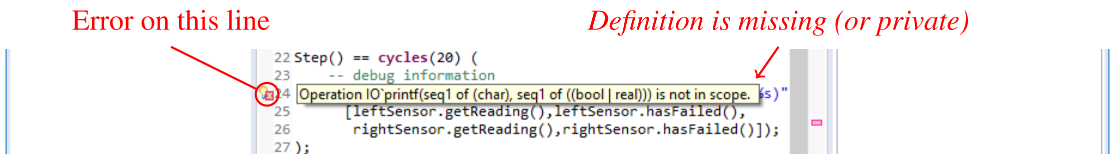
Step 5. Repeat the above step to make a file called *IRSensor* and populate it with the listing below. This class provides read access to two FMI ports, one for the sensor reading (`getReading`) and one to say if the sensor has failed (`hasFailed`).

```
class IRSensor

instance variables

-- access to ports from co-simulation
port : RealPort;
failed : BoolPort

operations

-- constructor for IRSensor
public IRSensor: RealPort * BoolPort ==> IRSensor
IRSensor(p,f) == (
  port := p;
  failed := f
);

public getReading: () ==> real
getReading() == (
  return port.getValue()
);

public hasFailed: () ==> bool
hasFailed() == (
  failed.getValue()
)

end IRSensor
```

Step 6. Next, create a file called *Servo* with the listing below. This class provides write access to a port to move the wheels of the robot (`setSpeed`). The range is -1 to 1 for full forwards or backwards, so a pre-condition is included to protect the operation. Note that since one servo on the robot is flipped over, a reverse flag can be set in the constructor to that setting both servos to 1 makes the robot go forwards at full speed.

```
class Servo

instance variables

-- access to ports from co-simulation
port: RealPort;
reversed: bool

operations

-- constructor for Servo
public Servo: RealPort * bool ==> Servo
Servo(p,r) == (
  port := p;
  reversed := r
);

public setSpeed: real ==> ()
setSpeed(value) == (
  if reversed
  then port.setValue(-value)
  else port.setValue(value)
)
pre -1 <= value and value <= 1

end Servo
```

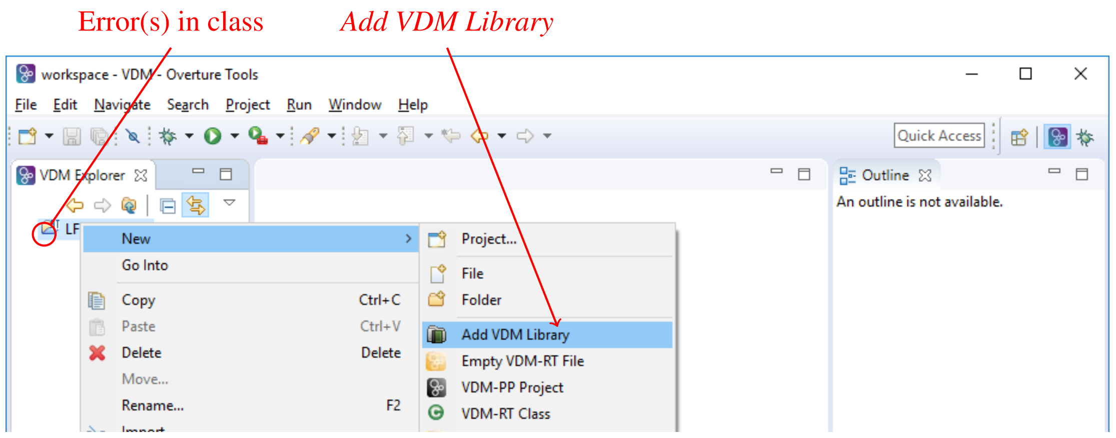
Step 7. You should notice in the *VDM Explorer* that `Controller.vdmrt` has a small red cross next to its icon. This means that there is one or more errors in the definition. In this case, the `IO` standard library is missing, which can be seen by hovering the mouse pointer over the error in the class. Note that "out of scope" either means that the definition is private, or missing entirely.



```
Step() == cycles(20) (
    -- debug information
                                                    %s)"
      [leftSensor.getReading(),leftSensor.hasFailed(),
       rightSensor.getReading(),rightSensor.hasFailed()]);
);
```

Operation IO`printf(seq1 of (char), seq1 of ((bool | real))) is not in scope.

Add the library by right-clicking the project and selecting *New* > *Add VDM Library*. Then check the *IO* box and click *Finish*. This will add `IO.vdmrt` to the `/lib` folder and the error should go away.



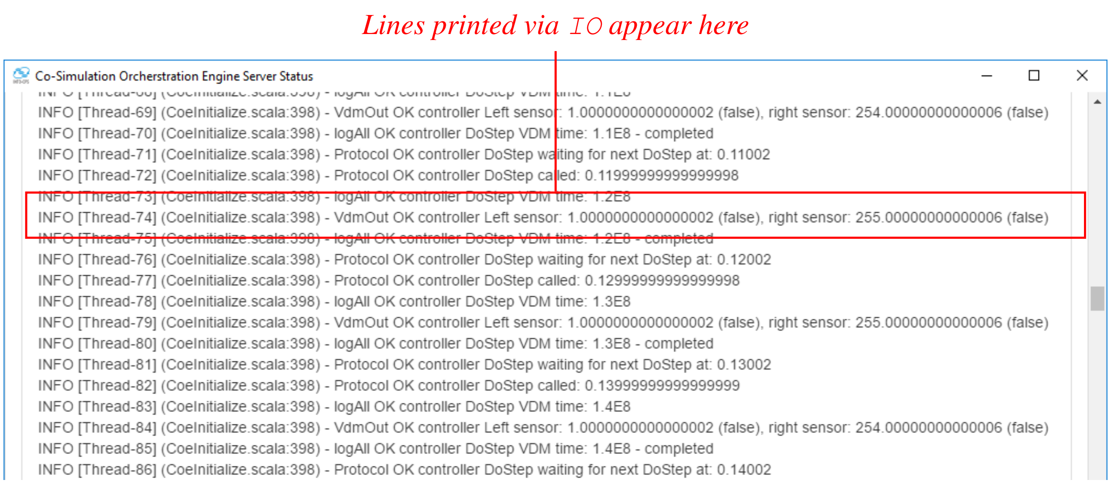
Step 8. In order to complete this basic controller and make an FMU, we must update the `System` class to instantiate sensor and actuator objects, then instantiate a controller object with these. Add the following lines to `System.vdmrt`:

```
system System

instance variables

-- Hardware interface variable required by FMU Import/Export
public static hwi: HardwareInterface := new HardwareInterface();

public static controller: [Controller] := nil;

private leftSensor: IRSensor;
private rightSensor: IRSensor;

private leftServo: Servo;
private rightServo: Servo;

private cpu : CPU := new CPU(<FP>, 1E6);

operations

public System : () ==> System
System () ==
(
  -- create sensor and actuator objects
  leftSensor := new IRSensor(hwi.lfLeftVal, hwi.lfLeftFailFlag);
  rightSensor := new IRSensor(hwi.lfRightVal, hwi.lfRightFailFlag);
  leftServo := new Servo(hwi.servoLeftVal, false);
  rightServo := new Servo(hwi.servoRightVal, true);
  -- create controller object
  controller := new Controller(leftSensor, rightSensor, leftServo, rightServo);

  -- deploy objects
  cpu.deploy(controller,  "Controller");
  cpu.deploy(leftSensor,  "Left sensor");
  cpu.deploy(rightSensor, "Right sensor");
  cpu.deploy(leftServo,   "Left servo");
  cpu.deploy(rightServo,  "Right servo");
);

end System
```

Step 9. Finally, uncomment line 8 of `World.vdmrt` to ensure that the controller logic will be started at the beginning of co-simulation:

```
class World

operations

public run : () ==> ()
run() ==
 (
  start(System`controller);
  block();
 );

private block : () ==>()
block() ==
  skip;

sync

  per block => false;

end World
```

Step 10. You can now test the project in INTO-CPS by exporting an FMU and pasting it into the *tutorial_9/FMUs* (follow Steps 16–19 of Tutorial 4). Then you can then open *tutorial_9* in the *INTO-CPS Application* and co-simulate with the *lfr-3d* or *lfr-non3d* multi-models. You should see the output from the VDM controller in the COE output:


Lines printed via IO appear here
Co-Simulation Orcherstration Engine Server Status

```
INFO [Thread-69] (CoeInitialize.scala:398) - VdmOut OK controller Left sensor: 1.0000000000000002 (false), right sensor: 254.00000000000006 (false)
INFO [Thread-70] (CoeInitialize.scala:398) - logAll OK controller DoStep VDM time: 1.1E8 - completed
INFO [Thread-71] (CoeInitialize.scala:398) - Protocol OK controller DoStep waiting for next DoStep at: 0.11002
INFO [Thread-72] (CoeInitialize.scala:398) - Protocol OK controller DoStep called: 0.11999999999999998
INFO [Thread-73] (CoeInitialize.scala:398) - logAll OK controller DoStep VDM time: 1.2E8
INFO [Thread-74] (CoeInitialize.scala:398) - VdmOut OK controller Left sensor: 1.0000000000000002 (false), right sensor: 255.00000000000006 (false)
INFO [Thread-75] (CoeInitialize.scala:398) - logAll OK controller DoStep VDM time: 1.2E8 - completed
INFO [Thread-76] (CoeInitialize.scala:398) - Protocol OK controller DoStep waiting for next DoStep at: 0.12002
INFO [Thread-77] (CoeInitialize.scala:398) - Protocol OK controller DoStep called: 0.12999999999999998
INFO [Thread-78] (CoeInitialize.scala:398) - logAll OK controller DoStep VDM time: 1.3E8
INFO [Thread-79] (CoeInitialize.scala:398) - VdmOut OK controller Left sensor: 1.0000000000000002 (false), right sensor: 255.00000000000006 (false)
INFO [Thread-80] (CoeInitialize.scala:398) - logAll OK controller DoStep VDM time: 1.3E8 - completed
INFO [Thread-81] (CoeInitialize.scala:398) - Protocol OK controller DoStep waiting for next DoStep at: 0.13002
INFO [Thread-82] (CoeInitialize.scala:398) - Protocol OK controller DoStep called: 0.13999999999999999
INFO [Thread-83] (CoeInitialize.scala:398) - logAll OK controller DoStep VDM time: 1.4E8
INFO [Thread-84] (CoeInitialize.scala:398) - VdmOut OK controller Left sensor: 1.0000000000000002 (false), right sensor: 254.00000000000006 (false)
INFO [Thread-85] (CoeInitialize.scala:398) - logAll OK controller DoStep VDM time: 1.4E8 - completed
INFO [Thread-86] (CoeInitialize.scala:398) - Protocol OK controller DoStep waiting for next DoStep at: 0.14002
```

As you can see from the live graph (and 3D visualisation), the left sensor is over black (low) and the right sensor is over white (high). The range is (0,255).

3 A Basic Controller

We will now add some basic line following logic. A so-called "bang-bang" controller turns left if the line is to the left, and right if the line is to the right. This creates a characteristic zig-zag motion.

Step 11. The control logic in the `Controller` class is in the `Step` operation. This is called periodically. Add an `if` statement to the `Step` operation to turn the robot to the left if the left sensor is over black and the right sensor is over white. You can assume that a sensor reading over 150 (halfway) is white and below 150 is black. You can drive the robot left and forward using the following calls:

```
    leftServo.setSpeed(0);
    rightServo.setSpeed(0.8)
```

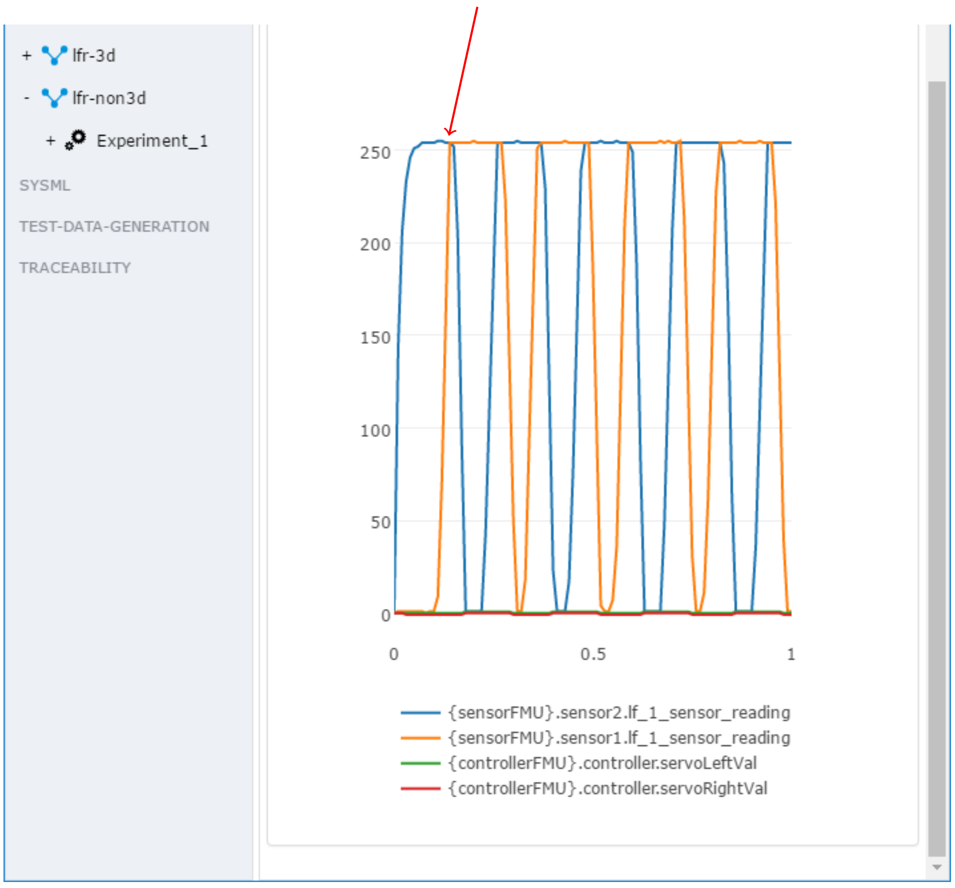
Changing the values will make the robot turn more or less. If both values are the same, the robot will move forwards or backwards in a straight line. If both values are exactly opposite (e.g. -1 and 1), the robot will turn on the spot.

Step 12. Add an `else if` clause to this statement to turn right if the left sensor is over white and the right sensor is over black.

Step 13. Add an `else if` clause to go forwards if both sensors are over black.

Step 14. Re-generate your FMU and check that the robot follows the line. If not, you can add `print` statements to your if-statement to check what conditions are being triggered.

Zig-zag behaviour demonstrating line following

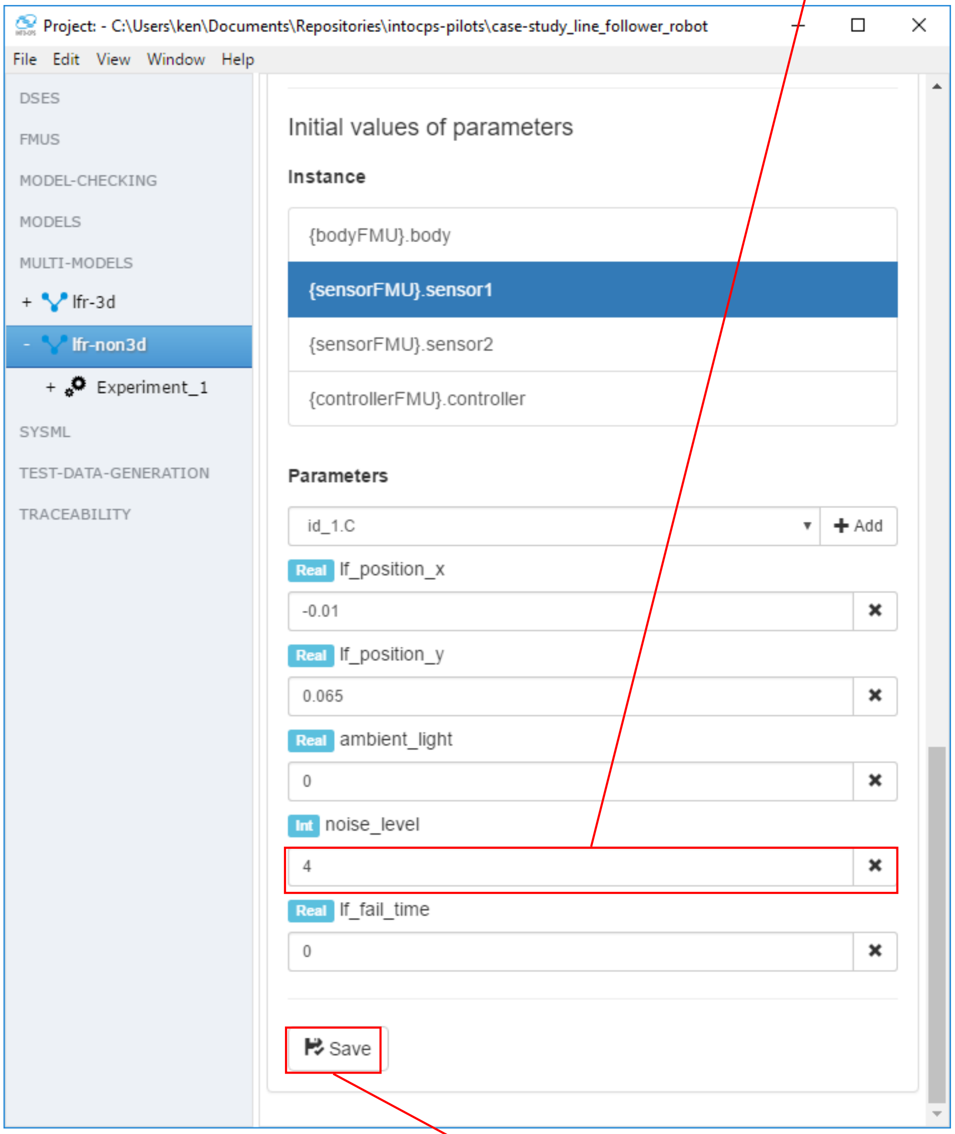

+ lfr-3d
- lfr-non3d
+ Experiment_1
SYSML
TEST-DATA-GENERATION
TRACEABILITY
250
200
150
100
50
0
0
0.5
1
{sensorFMU}.sensor2.lf_1_sensor_reading
{sensorFMU}.sensor1.lf_1_sensor_reading
{controllerFMU}.controller.servoLeftVal
{controllerFMU}.controller.servoRightVal

4 Dealing with Noisy Data

The sensor model contains some *realistic* and *faulty* behaviours, which can be turned on or off from the INTO-CPS Application in the multi-model configuration. The first realistic behaviour is sensor noise. This occurs when converting analogue readings to a digital values, and results in readings that bounce up and down.

Step 15. Edit the *Initial values of parameters* for {*sensorFMU*}.*sensor1* and {*sensorFMU*}.*sensor2* and set the *noise_level* to *4*, where the range is (0,8).

Set noise_level



Initial values of parameters

Instance

{bodyFMU}.body
{sensorFMU}.sensor1
{sensorFMU}.sensor2
{controllerFMU}.controller

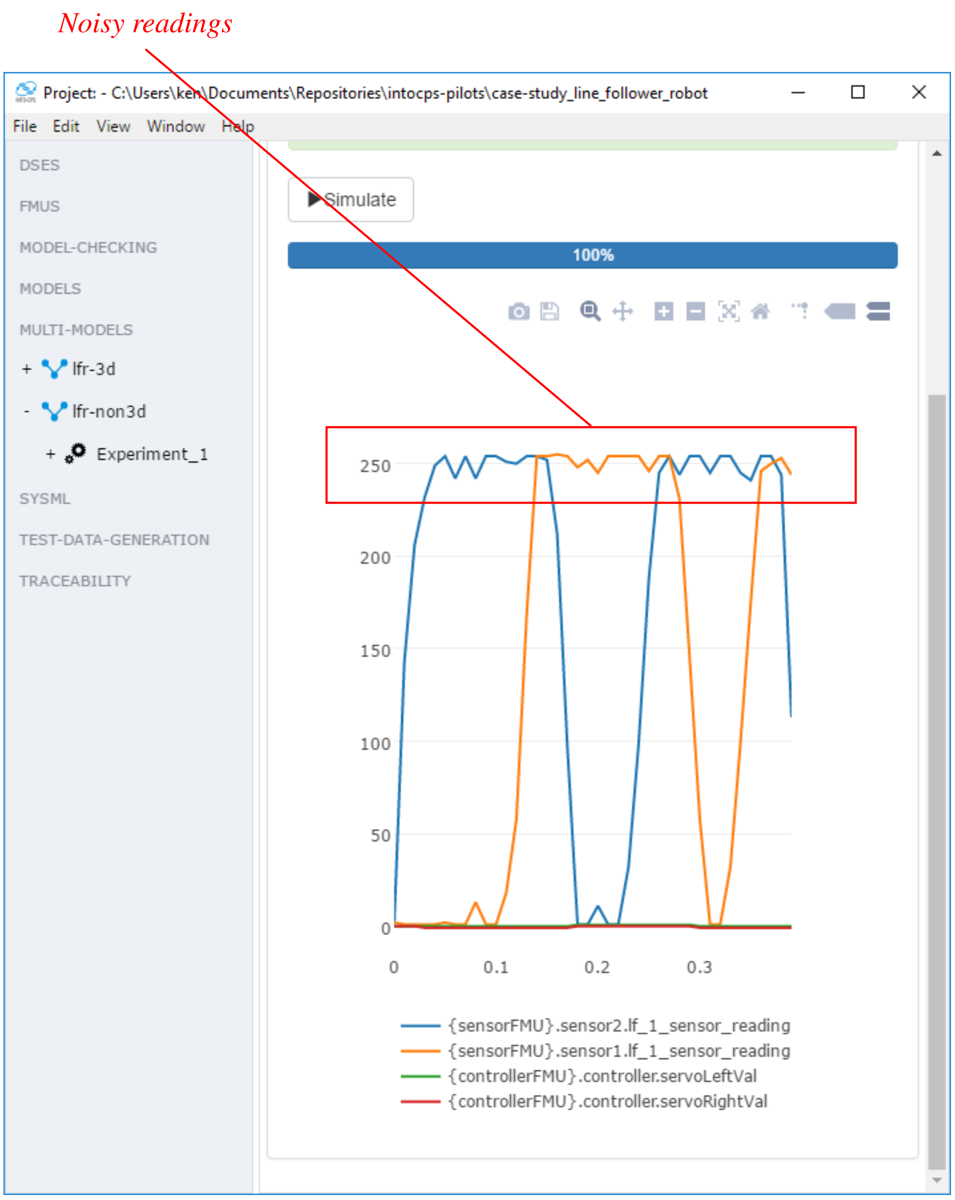
Parameters

id_1.C

Real lf_position_x: -0.01
Real lf_position_y: 0.065
Real ambient_light: 0
Int noise_level: 4
Real lf_fail_time: 0

Don't forget to *Save* configuration

Step 16. Run the co-simulation and observe the curve of the sensor readings now shows a noisy signal.



Noisy readings
Project: - C:\Users\ken\Documents\Repositories\intocps-pilots\case-study_line_follower_robot
File Edit View Window Help
DSES
FMUS
MODEL-CHECKING
MODELS
MULTI-MODELS
lfr-3d
lfr-non3d
Experiment_1
SYSML
TEST-DATA-GENERATION
TRACEABILITY
Simulate
100%
250
200
150
100
50
0
0
0.1
0.2
0.3
{sensorFMU}.sensor2.lf_1_sensor_reading
{sensorFMU}.sensor1.lf_1_sensor_reading
{controllerFMU}.controller.servoLeftVal
{controllerFMU}.controller.servoRightVal

Step 17. To cope with this noise we will add a filter that provides a floating average of the last five readings. Create a file called `FilteredIRSensor.vdmrt` and populate it from the listing below. This class is defined as a *subclass* of `IRSensor` so it can be passed seamlessly to the `Controller` class. It encapsulates an `IRSensor` object, so it can intercept the readings and provide a filtered value:

```
class FilteredIRSensor is subclass of IRSensor

instance variables

-- sensor to be filtered
private sensor: IRSensor;

-- sequence of previous readings
private samples: seq of real
```

```
operations

-- constructor for FilteredIRSensor
public FilteredIRSensor: IRSensor ==> FilteredIRSensor
FilteredIRSensor(s) == (
  sensor := s;
  samples := []
);

public getReading: () ==> real
getReading() == (
  dcl reading: real := sensor.getReading();
  dcl average: real := 0;

  IO`printf("Average: %s of %s\n", [average, samples]);
  return average
);

public hasFailed: () ==> bool
hasFailed() ==
  return sensor.hasFailed();

end FilteredIRSensor
```

Step 18. As defined above, the `getReading` operation simply passes on a value of 0. Extend this operation (at the highlighted line) to store `reading` in the `samples` sequence and to calculate the *average* value of the sequence. The samples should store only the 5 newest values. *Hint: the ^ operator concatenates lists,* ***hd*** *yields the first item in a list, and* ***tl*** *yields the remainder of a list once the head is removed.*

Step 19. We have to modify the `System` class create `FilteredIRSensor` objects and pass them to the controller. Modify `System` as follows, then run your co-simulation again. You can check your filtered value with the information printed in the COE status window.

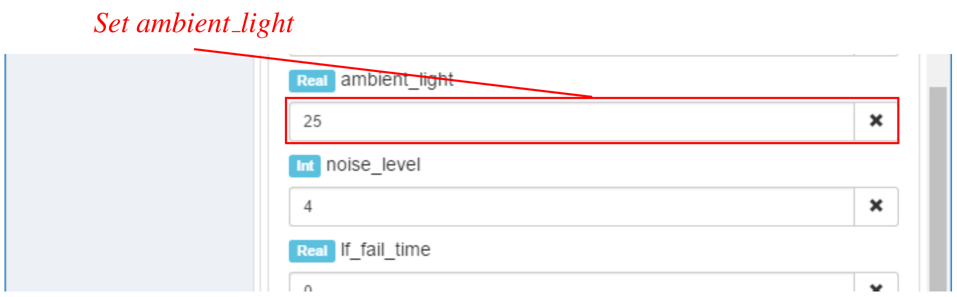
```
private leftSensor: IRSensor;
private rightSensor: IRSensor;

private leftFilter: FilteredIRSensor;
private rightFilter: FilteredIRSensor;

private leftServo: Servo;
private rightServo: Servo;
```

```
public System : () ==> System
System () ==
(
  -- create sensor and actuator objects
  leftSensor := new IRSensor(hwi.lfLeftVal, hwi.lfLeftFailFlag);
  rightSensor := new IRSensor(hwi.lfRightVal, hwi.lfRightFailFlag);
  leftFilter := new FilteredIRSensor(leftSensor);
  rightFilter := new FilteredIRSensor(rightSensor);
  leftServo := new Servo(hwi.servoLeftVal, false);
  rightServo := new Servo(hwi.servoRightVal, true);

  -- create controller object
  controller := new Controller(leftFilter, rightFilter, leftServo, rightServo);

  -- deploy objects
  cpu.deploy(controller,  "Controller");
  cpu.deploy(leftFilter,  "Left sensor");
  cpu.deploy(rightFilter, "Right sensor");
  cpu.deploy(leftServo,   "Left servo");
  cpu.deploy(rightServo,  "Right servo");
);
```

5 Dealing with Ambient Light

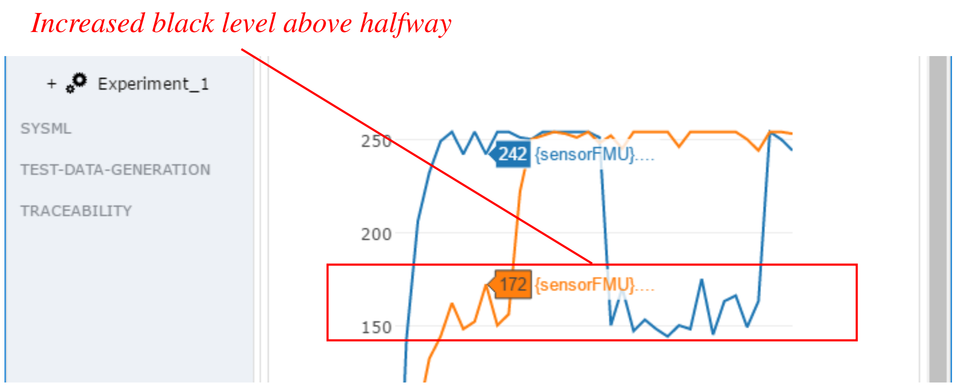
The second realistic behaviour is ambient light. The infrared sensor works by shining a beam of infrared light out and looking for a reflection, however the environment can contain a lot of infrared light, e.g. if it's a sunny day. This can make it difficult for the sensor to see black.

Step 20. Edit the *Initial values of parameters* for {*sensorFMU*}.*sensor1* and {*sensorFMU*}.*sensor2* and set the *ambient_light* to *25* (W), where the range is (0, 40).



Real ambient_light
25
Int noise_level
4
Real lf_fail_time
0

Step 21. Run the co-simulation and observe the increased black level.



Step 22. This can be overcome by adding modal behaviour to the control. Since we know that the left sensor begins over black, we can add a *calibration* mode that takes some readings to determine what value black is and uses this to determine the threshold. The controller can then switch to the existing logic in a *following*. Because the filtering delays the response of the sensor, we should also *wait* briefly before taking the calibration readings.

Step 23. Add the following type to the `Controller` class:

```
types

Mode = <WAIT> | <CALIBRATE> | <FOLLOW>
```

Step 24. Add the following instance variables

```
mode: Mode := <WAIT>;
samples: seq of real := [];
THRESHOLD: real := 150
```

Step 25. Modify the `Step` operation to include the modal behaviour described above. A simple way is to add a top-level `if` statement such as:

```
if mode = <WAIT> then ...
elseif mode = <CALIBRATE> then ...
elseif mode = <FOLLOW> then ...
```

The `<WAIT>` should do nothing until the simulation time is at 0.5 seconds (the current simulation in seconds time is given `time/1e9`), then change `mode` to `<CALIBRATE>`. Calibrate mode should add five readings from the `leftSensor` to the `samples` list, compute `threshold` as the average, then change mode to `<FOLLOW>`. The follow mode should contain your existing logic, but use `threshold` to determine if a sensor is seeing black and white.

Step 26. Add some `IO`printf` statements to your controller to indicate when it changes mode, then run the co-simulation and convince yourself the controller is working.

6 Dealing with Sensor Failure

The faulty behaviour in the sensor model is a complete failure, which will always produce a value of zero. It is possible to follow the line using a single sensor if this occurs. The parameter sets the time, in seconds, when the failure will occur (0 means never).

Step 27. Edit the *Initial values of parameters* for *{sensorFMU}.sensor1* and set the *lf_fail_time* to *2* (s). You can set it later but you have to simulate longer until it triggers. Run the co-simulation to see how the robot behaves after the failure.

Step 28. Extend your controller to add a new mode called `<SINGLE_FOLLOW>`. Your controller should switch to this mode if one of the sensors fails, then continue following the line using the remaining working sensor. If both sensors fail the robot should stop. The robot will need to move slower, which is a *degraded behaviour*: where a service is still offered but with lower performance. *Hint: you can follow the line with a single sensor by detecting the edge of the line – a change from black to white, or vice versa. Also slow means 0.3 power maximum.*

Step 29. Run the co-simulation again to check that the controller switches mode at the right time, and can now follow the line despite the failed sensor.

7 Additional Exercises

The suggested layout for the controller logic is not necessarily easily maintainable for larger controllers. Try refactoring the controller to make it more maintainable. Suggestions include:

- Moving the logic for each mode, and for mode changes, to auxiliary functions.
- Make an object-oriented version using the *State pattern*. Each mode is represented by an object that contains the logic. You can create an abstract mode class that provides access to the sensor and actuators, and empty `Enter`, `Step`, and `Exit` operations. Each mode is then defined as a subclass of this mode, overriding the operations as required. You can permit internal mode changes by allowing modes to return a new mode from their `Step` operation.