

Exercise 3 — VDM for Embedded Controllers

Ken Pierce, February 2018

A. Aims

The aim of this tutorial is to help you to become familiar with Overture (creating a project, adding standard libraries, executing a simulation) and VDM, including building a controller in VDM-RT and testing it using pre-recorded sensor data, and dealing with realistic and faulty sensor behaviours.

B. Submission

This exercise must be signed-off by a demonstrator at a practical before the Easter break. This means the last practical for sign-offs is Monday 12th March.

C. Instructions

The example in this exercise is a small line-following robot with two infrared sensors. The robot has two motors that allow it to move around. By moving the motors at the same speed, it can drive forwards and backwards, and by moving the motors at different speeds, it can turn left and right. The robot has infrared sensors that can detect if the surface below it is black or white.

In this exercise you will build a controller for this robot using VDM and test with some example data. The controller should allow the robot to follow the line by detecting what each infrared sensor sees and setting the speed of the wheels, in addition to handling realistic and faulty data from the sensors. The structure of the controller is designed in such a way that the test harness could be removed and the same controller used in a co-simulation, with a model of the robot body.

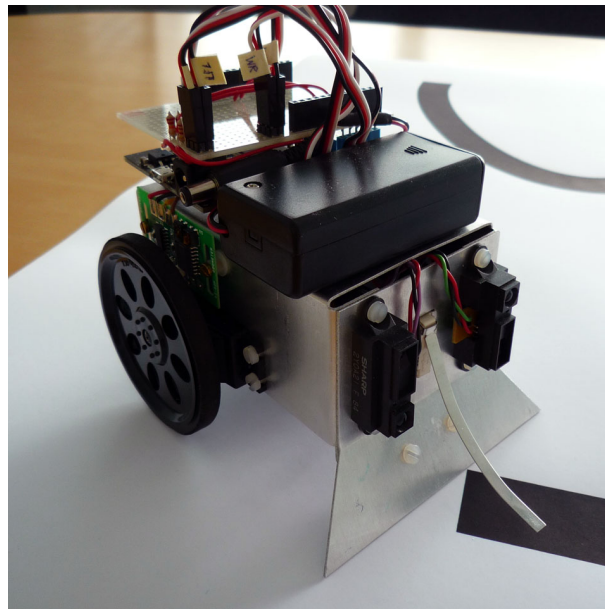
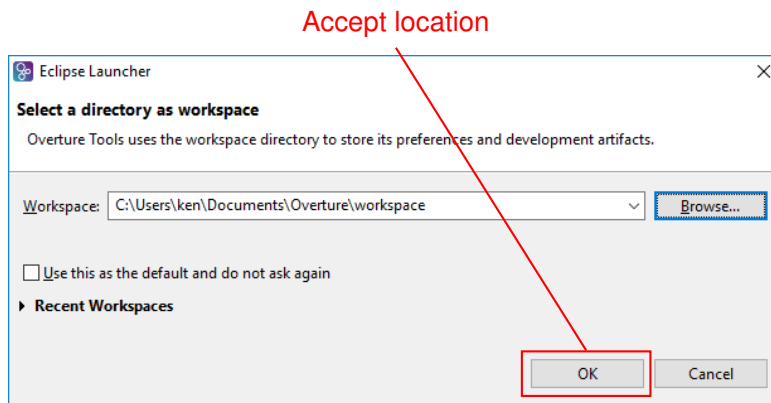


Figure 1: The line-following robot

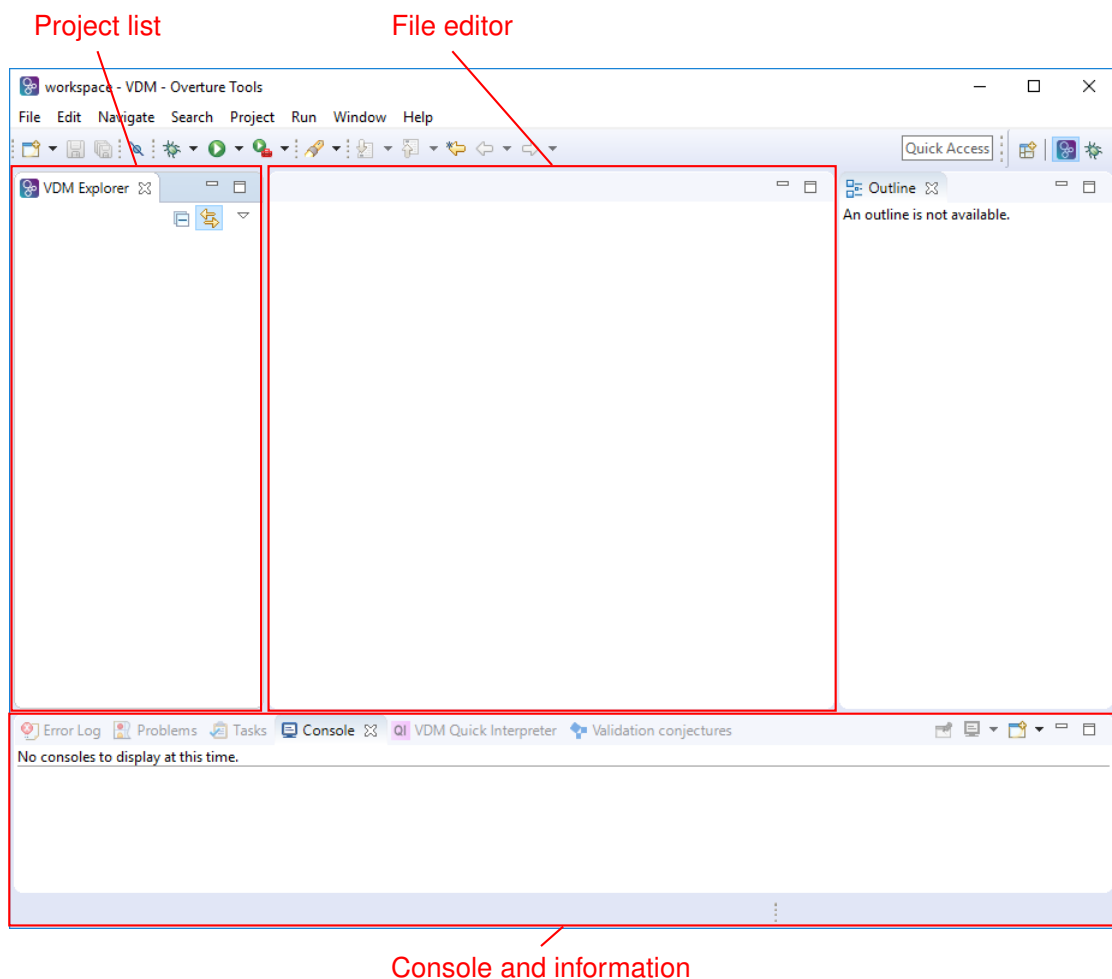
Part 1 — Make an Overture Project

To begin we will create a project in Overture to hold our model.

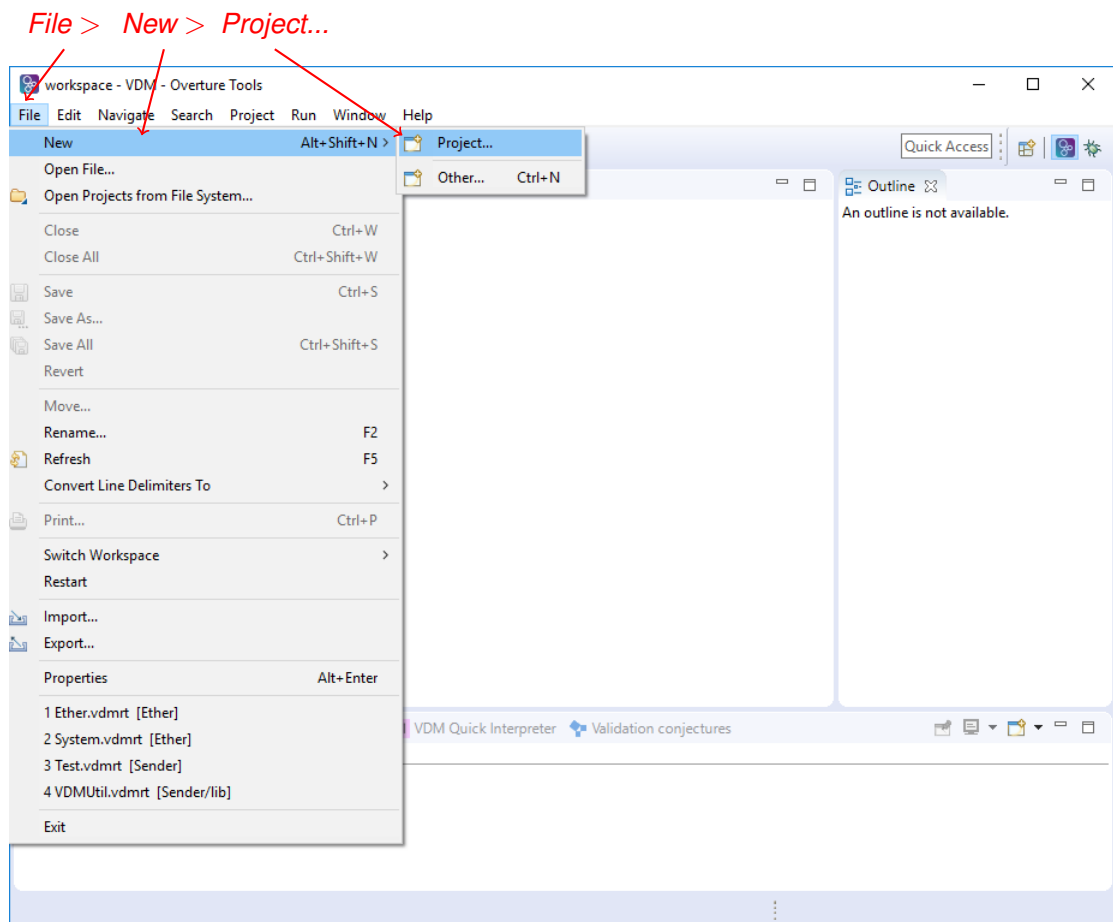
Step 1. Open *Overture*. It will prompt you to select a location for its workspace. You may accept the default location by pressing *OK*, or press *Browse...* to select a different location. If you do not want to be prompted in future, check *Use this as the default and do not ask again*.



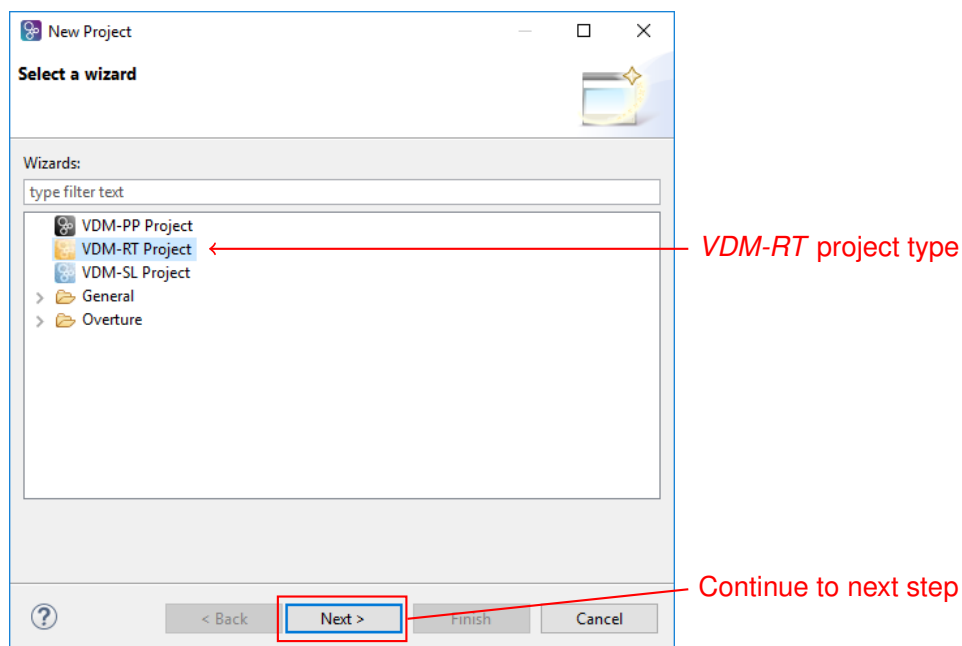
This is the *Overture* window, which includes a project list, file editor and a console.



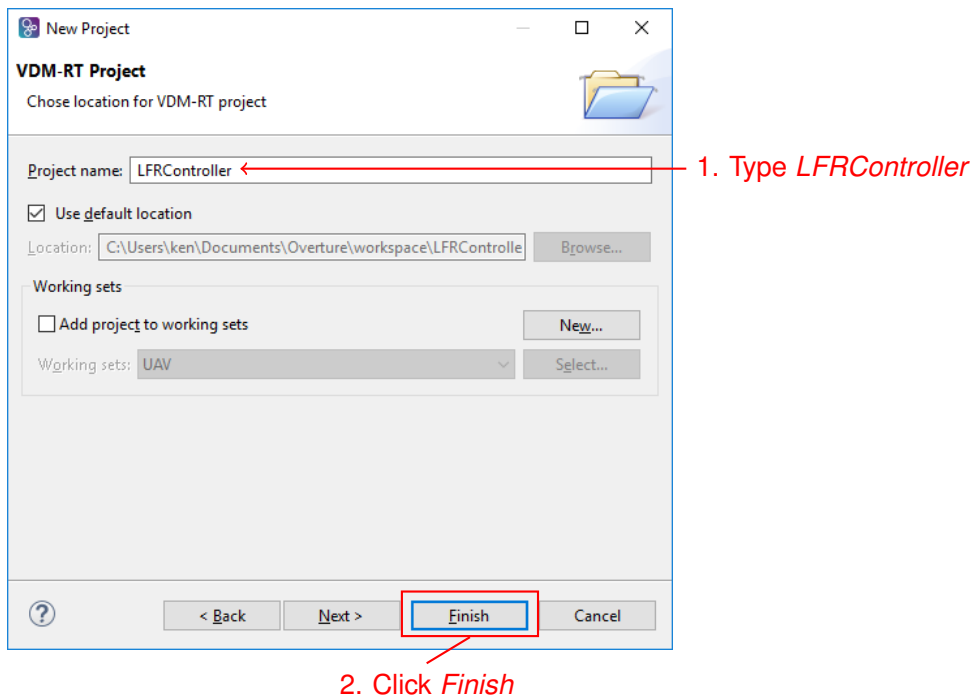
Step 2. First create a project that will hold the controller model. Select *File > New > Project...*



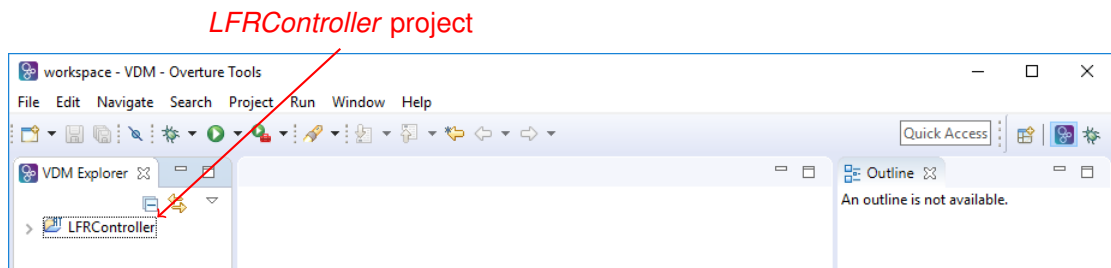
Step 3. In the *New Project* window, select *VDM-RT Project* and click *Next >* to go to the next step.



Step 4. The next screen asks for a name for the project. Call it *LFRController* and click *Finish*.



You should see the new project in the project list.



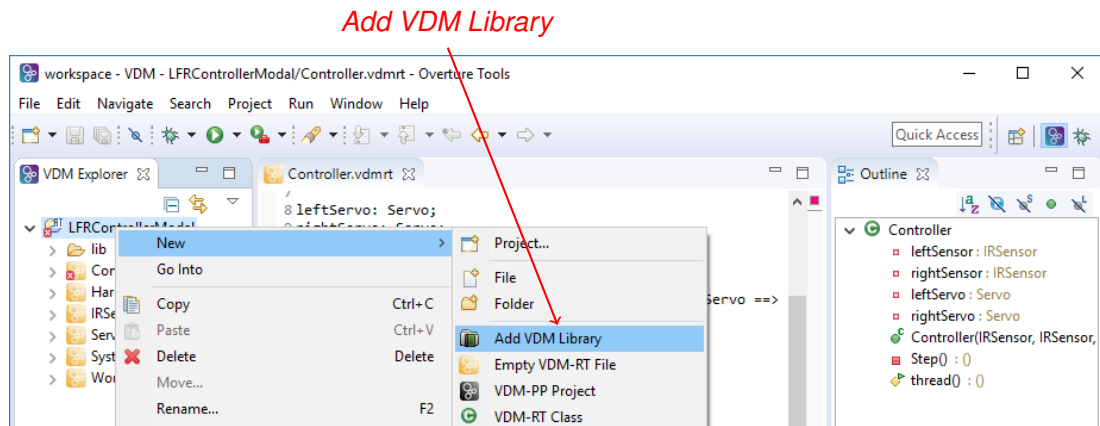
Part 2 — Basic Structure and Skeleton Controller

The basic structure of a VDM-RT controller is to have: a `Controller` class that contains control logic; one or more `Sensor` and `Actuator` classes that allow the controller to interact with its environment; a `System` class that instantiates these objects and describes the composition of the model; and a `World` class that provides an entry point for simulation and starts any threads.

In a co-simulation context, the sensor and actuator classes will interact with other models in the co-simulation. However we can also take a “DE-first approach” where test data comes from part of the VDM model, in order to test ideas before other models are ready, for example. This is often provided through an `Environment` class, which the `Sensor` and `Actuator` classes interact with. This can be removed later when co-simulation is ready by providing alternative implementations of these classes that interact with the COE.

Such an `Environment` class is included with this tutorial. We suggest that you complete at least this section before examining it closely — you can return to it to understand how it works later.

Step 5. We require the standard IO library to print to the Overture console testing. 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 a folder called */lib* in your project.



Step 6. Copy *Environment.vdmrt* and paste it into your project. You can do this by right-clicking on the project and selecting *Paste*.

Step 7. Right-click on the *LFRControllerModal* project and select *New > Empty VDM-RT File*. Call it *Controller* and click *Finish*.

Step 8. 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 9. Repeat the above step to make a file called *IRSensor* and populate it with the listing below. This class provides operations to read infrared sensor data (*getReading*) and one to say if the sensor has failed (*hasFailed*).

```
class IRSensor

instance variables

-- access to test data
env: Environment;
side: Environment`Side

operations

-- constructor for IRSensor
public IRSensor: Environment * Environment`Side ==> IRSensor
IRSensor(e,s) == (
  env := e;
  side := s
);

public getReading: () ==> real
getReading() ==
  return env.getSensorValue(side);

public hasFailed: () ==> bool
hasFailed() ==
  return getReading() = 0

end IRSensor
```

Step 10. Next, create a file called *Servo* with the listing below. This class provides an operation to move a wheel of the robot (*setSpeed*). The range is 1 to -1 for full forwards or backwards respectively, so a pre-condition is included to protect the operation.

```
class Servo

instance variables

-- access to test data
env: Environment;
side: Environment`Side

operations

-- constructor for Servo
public Servo: Environment * Environment`Side ==> Servo
Servo(e,s) == (
  env := e;
  side := s
);

public setSpeed: real ==> ()
setSpeed(value) ==
  env.setServoSpeed(side,value)
pre -1 <= value and value <= 1

end Servo
```

Step 11. We must create a `System` class to instantiate the environment, sensor and actuator objects, then instantiate a controller object with these as parameters. Create the following listing as in a file called `System.vdmrt`:

```
system System

instance variables

-- test data object
public static env: Environment := new Environment(<BASIC>);

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(env, <LEFT>);
  rightSensor := new IRSensor(env, <RIGHT>);
  leftServo := new Servo(env, <LEFT>);
  rightServo := new Servo(env, <RIGHT>);

  -- 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 12. Finally create the following as `World.vdmrt`:

```
class World

instance variables

public static done: bool := false

operations

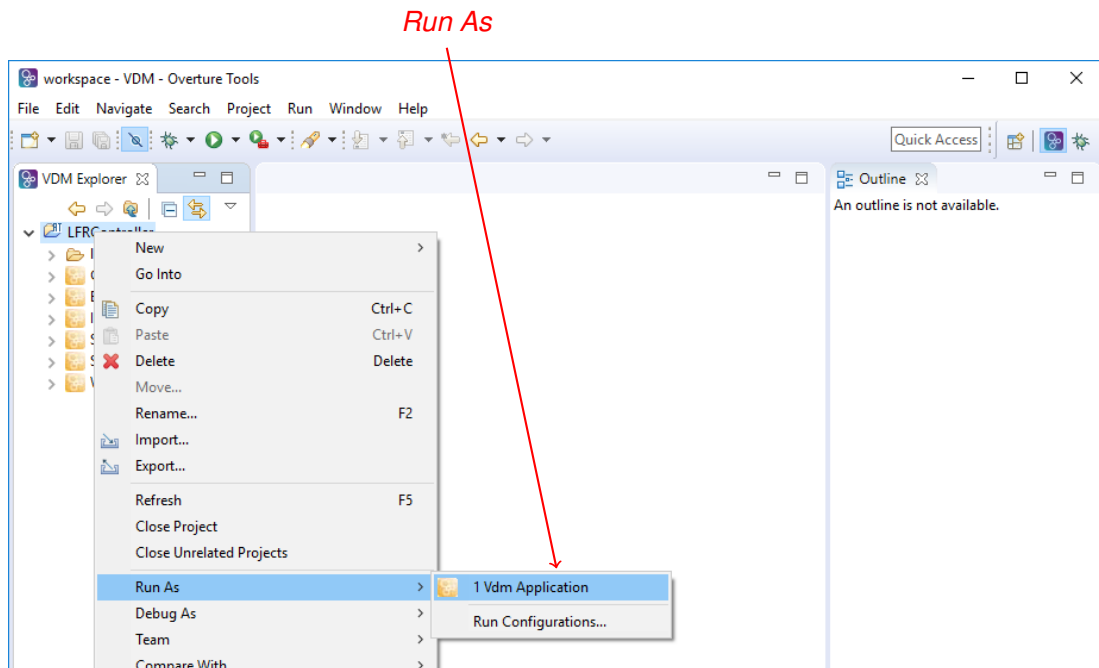
public run : () ==> ()
run() == (
  start(System`controller);
  start(System`env);
  block();
);

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

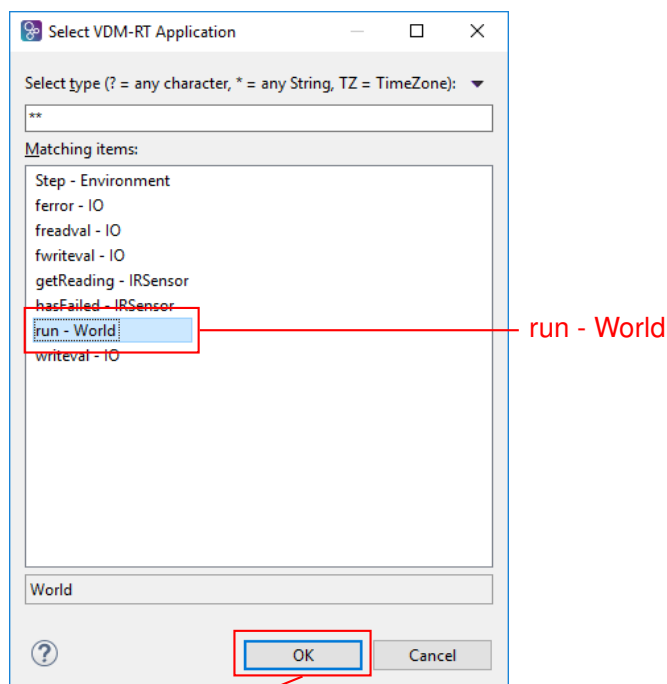
sync per block => done;

end World
```

Step 13. You can now run a simulation to test the controller by right-clicking on the project and selecting *Run As > 1 Vdm Application*.

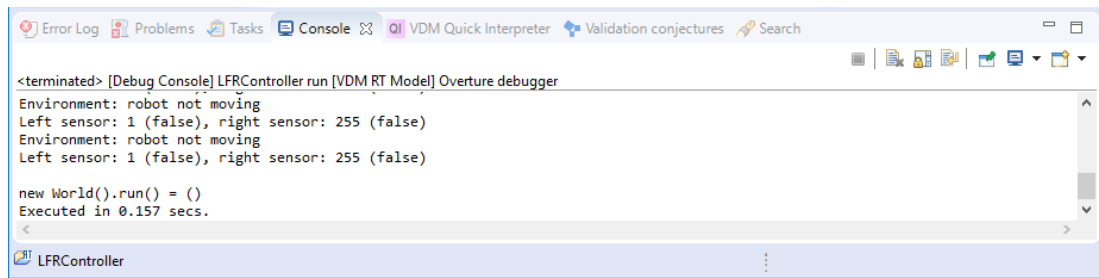


In the dialogue box select *run - World* and press *Okay*.



2. Click *Ok*

You should see the following output in the *Console* pane. This is produced by the call to `IO `printf` in `Controller` (and in `Controller`).



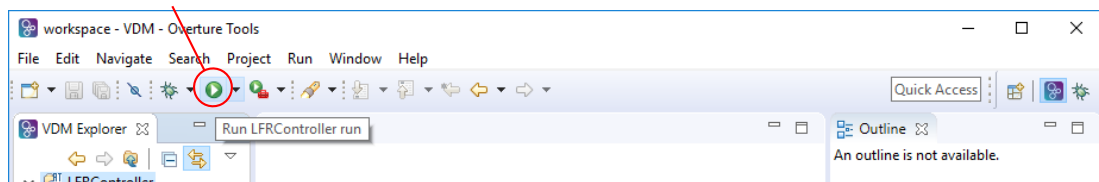
```
<terminated> [Debug Console] LFRController run [VDM RT Model] Overture debugger
Environment: robot not moving
Left sensor: 1 (false), right sensor: 255 (false)
Environment: robot not moving
Left sensor: 1 (false), right sensor: 255 (false)

new World().run() = ()
Executed in 0.157 secs.
```

As you can see, the left sensor is over black (low) and the right sensor is over white (high), corresponding to the range (1,255), with 0 reserved for distinguishing a failed sensor.

Step 14. After you have simulated the model once, a *Run Configuration* is created called *LFRController run*. You can use the green play button to simulate the project from now on.

Run simulation



Part 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. The `Environment` class does not simulate robot movement, but it does say which way the robot would move, so you can check your implementation by reading the console output.

Step 15. 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 16. 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 17. Add an `else if` clause to go forwards if both sensors are over black.

Step 18. Re-run your simulation and check that the `Environment` outputs the direction you expect.

Part 4 — Dealing with Noisy Data

The `Environment` contains some test data for *realistic* and *faulty* behaviours, which can be selected by changing the parameter passed to the constructor (called in `System`). 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 19. Edit the `System` class to change the parameter passed to the constructor of `Environment` from `<BASIC>` to `<NOISY>`.

```
system System

instance variables

-- test data object
public static env: Environment := new Environment(<BASIC>);

public static controller: [Controller] := nil;
```

Step 20. Run the simulation and observe the values of left and right are no longer stable at 1 and 255, but bounce around.

Step 21. 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;

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

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

end FilteredIRSensor
```

Step 22. 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 (e.g. `list ^ [new_item]`), `hd` yields the first item in a list, and `tl` yields the remainder of a list once the head is removed.*

Step 23. We have to modify the `System` class create `FilteredIRSensor` objects and pass them to the controller. Modify `System` as follows.

```
private leftSensor: IRSensor;  
private rightSensor: IRSensor;  
  
private leftSensorFiltered: FilteredIRSensor;  
private rightSensorFiltered: FilteredIRSensor;  
  
private leftServo: Servo;  
private rightServo: Servo;
```

```
public System : () ==> System  
System () ==  
(  
  -- create sensor and actuator objects  
  leftSensor := new IRSensor(env, <LEFT>);  
  rightSensor := new IRSensor(env, <RIGHT>);  
  leftFilter := new FilteredIRSensor(leftSensor);  
  rightFilter := new FilteredIRSensor(rightSensor);  
  leftServo := new Servo(env, <LEFT>);  
  rightServo := new Servo(env, <RIGHT>);  
  
  -- create controller object  
  controller := new Controller(leftFilter, rightFilter, leftServo, rightServo);  
  
  -- deploy objects  
  cpu.deploy(controller, "Controller");  
  cpu.deploy(leftSensor, "Left sensor");  
  cpu.deploy(rightSensor, "Right sensor");  
  cpu.deploy(leftFilter, "Left sensor filtered");  
  cpu.deploy(rightFilter, "Right sensor filtered");  
  cpu.deploy(leftServo, "Left servo");  
  cpu.deploy(rightServo, "Right servo");  
);
```

Step 24. Run the simulation and observe the data produced by the filtered sensor (floating average and sequence of readings).

Part 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 25. Edit the `System` class to change the parameter passed to the constructor of `Environment` from `<NOISY>` to `<AMBIENT>`.

Step 26. Run the simulation and observe the values of left sensor are much higher, past halfway to white. This means the controller may mistake black for white.

Step 27. This can be overcome by adding modal behaviour to the controller. 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 28. Add the following type to the `Controller` class:

```
types
```

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

Step 29. Add the following instance variables

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

Step 30. 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 0.05 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 31. Add some `IO 'printf` statements to your controller to indicate when it changes mode, then run the simulation to convince yourself the controller is working.

Part 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 32. Edit the `System` class to change the parameter passed to the constructor of `Environment` from `<AMBIENT>` to `<FAILURE>`.

Step 33. Run the simulation and observe that one of the sensors fails after some readings, yielding a value of 0 (`true`).

Step 34. 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. *Hint: the robot should change direction when it sees the edge of the line – a change from black to white, or vice versa.*

Step 35. Run the simulation again to check that the controller switches mode at the right time, and that the robot would change direction as it detects a change from black to white or white to black.

Step 36. Show your final simulation to a demonstrator.

[2 marks]

Remember to have this exercise signed off by demonstrators by Monday 12th March.