Crescendo Technical Report Series
No. TR-002

January 2014

Crescendo Examples Compendium

by

Claire Ingram, Ken Pierce, Carl Gamble,

Sune Wolff, Martin Peter Christensen and Peter Gorm Larsen
Aarhus University, Department of Engineering
Finlandsgade 22, DK-8200 Aarhus N, Denmark

Document history

Month	Year	Version	Version of Crescendo.exe
December	2012	1	1.1.8 (then called DESTECS)
January	2014	2	2.0.0

Contents

Chapter 1

Introduction

This deliverable provides an overview of different public example co-models that stakeholders who are interested in experimenting with the **Crescendo** technology can use as a starting point[1].

This deliverable is structured in different chapters, each of which provides a brief (2 to 3 pages) introduction to one example model. Depending upon your own background and your interest in exploring the **Crescendo** technology and the particular features that interest you, you may wish to start with a different chapter than that presented first. The different chapters and the examples they present illustrate different aspects as explained here:

- Chapter 2 illustrates the *Simple Controller*. This example may be of particular interest to those with a background in CT modelling who want to see how to convert an existing CT model to a **Crescendo** co-model.

- Chapters 3 and 4 present two different models of a water tank which maintains a given water level. Chapter 3 describes a model which in which the controller periodically polls the sensor for the current state, whilst Chapter 4 describes a similar model that employs an event-driven solution to the same problem.

- Chapter 5 presents a small example with a pair of rollers used inside a printer (called "pinches"). This example is of particular relevance for those who are interested in different types of faults and alternative fault tolerance techniques.

- Chapter 6 presents a fuel system for an aircraft. This example has been used in other formalisms as well and different faulty scenarios are also considered.

- Chapter 7 describes a decoder which can translate messages presented in Morse code (as "dots" and "dashes" on a strip of tape) into Latin characters. This example demonstrates the use of shared design parameters to test model performance in different environmental conditions (such as varying levels of background noise).

[1] The corresponding sources of all the examples can be directly imported into the **Crescendo** tool.

- Chapter 8 illustrates a line-following robot. This example demonstrates how the **Crescendo** technology can support design space exploration using an Automated Co-model Analysis (ACA). Here it is used to analyse alternative placements of different light sensors.
- Chapter 9 expands on Chapter 8. It presents five further co-models of a line-following robot, each developed independently by attendees at the 2012 **Crescendo** summer school. Each co-model exhibits different design choices and optimisations for solving the same problems and so these models are interesting to study side-by-side for those interested in exploration of the design space.
- Chapters 11, 12 and 13 present progressively more complex models of a self-balancing scooter, called a Chessway. Chapter 11 describes a very simple model that includes basic functionality only. Chapter 13 describes a more complex model of the same scooter. The latter includes some components linked over a wireless connection, and so it also demonstrates some fault tolerant techniques. The plant model for this particular model is larger than many other examples. Chapter 12 presents an intermediate model which bridges the gap between the simple model and the more complex one. It includes a DE model only, which has been developed in order to answer some key design questions necessary before the model in Chapter 13 can be built.

In order to guide you as to which models to consider inspecting we have created a table illustrating the different characteristics of the various co-models available (see Table 1.1); characteristics which might be of interest include scripted scenarios, fault tolerance, Automated Co-model Analysis (ACA), 2D/3D animation, and patterns which have been implemented in each model. For more on patterns, see **Crescendo** deliverable D2.3 [BFG+12]. Some models here have been implemented entirely as a DE model (e.g., entirely in VDM, with no 20-sim component), or entirely as a CT model (e.g., entirely in 20-sim, with no VDM component). We indicate whether this applies in Table 1.1, with *CT alone enabled* or *DE alone enabled*.

Co-model(s)	Event triggered	CT alone enabled	DE alone enabled	Scenarios included	CT faults included	DE faults included	ACA included	2D/3D animation	Patterns implemented
Simple controller		x							1
Water tank (periodic)				x		x		x	2
Water tank (event-driven)	x							x	2
Paper Pinch		x			x				2, 6, 9, 10, 12
Aircraft fuel system				x	x			x	2, 9, 13
Morse code reader					x				2, 3
Line following robot					x		x	x	2, 3, 4, 5, 8, 9
Summer school Group 1								x	2
Summer school Group 2								x	2, 3
Summer school Group 3								x	2
Summer school Group 4								x	2, 3
Summer school Group 5								x	2
Tractor simple								x	2
ChessWay simple								x	1
ChessWay SL			x			x			
Chessway **Crescendo**		x	x		x	x		x	1, 7, 13

Table 1.1: Overview of characteristics for co-models

Chapter 2

Simple Controller

2.1 Case Description

The Simple Controller model showcases a simple design pattern for connecting an existing CT plant model to a DE model. The overall model consists of three main blocks (see Figure 2.1), `Controller`, `IO`, and `Plant`. The `Controller` block houses the controller that calculates an appropriate output for the motor. The `IO` block resides between the `Controller` and the `Plant`, and handles D/A and A/D conversion, scaling and quantization.

The plant represents a motor which drives a wheel (for example, on a car) to achieve a desired target speed, with a closed feedback loop. The controller computes a motor steering value, using the difference between the set point and the measured value (from the encoder). The motor applies torque to a rotating wheel, and an encoder monitors actual rotations and feeds this information back to the controller.

There are two versions of the Simple Controller model provided, which can be compared side-by-side (it is possible to switch between the two models in 20-sim). One version is entirely implemented as a CT model, with a CT controller block to interact with the motor and an encoder. The second version is a **Crescendo** co-model, which has a controller implemented in VDM. Because there are two versions of the model provided, the Simple Controller may be of interest to those with a background in CT modelling who want to see how to convert an existing CT model to a **Crescendo** co-model. The design pattern visible in the co-model here is reused in many other, more complex **Crescendo** co-models.

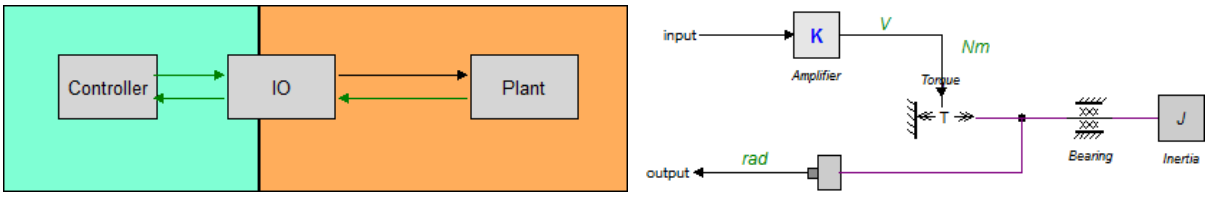


Figure 2.1: The 20-sim top level model (left) and the plant (right)

2.2 Contract

The contract contains two variables of type real. A monitored variable, `counts`, represents data returned from the sensor monitoring the movement of the wheel, in the form of a figure representing the number of radians rotated. A controlled variable, `rev`, produces a output signal for the motor actuator.

2.3 Discrete-event

The DE model contains a `Controller` class, which has the main thread of control. The `Controller` instantiates abstract classes to represent the actuator (`IActuatorReal`) and the sensor (`ISensorReal`). At run-time concrete implementations of these classes are provided by the `Actuator` class and the `Sensor` class respectively. Designing a controller that primarily handles abstract classes makes it easier to replace or alter the concrete implementation at a later date if necessary.

The `Controller` is deployed onto a CPU running at 10MHz by the `System` class.

2.4 Continuous-time

There are two versions of the CT model provided in 20-sim, one with all control implemented entirely in a CT (20-sim) model, and one with all control implemented in VDM (i.e., a **Crescendo** co-model). Both models share the same plant, in which a power signal, *K*, is output to a motor, *T*, which produces torque to rotate a wheel, *J*, modeling its inertia, whilst the bearing attached to the wheel models friction.

In the CT-only model the `Controller` block houses the controller that calculates an appropriate output for the motor.

In the second model - a **Crescendo** co-model - the functionality previously implemented by the `Controller` has now been moved into a VDM model in the **Crescendo** tool, and so the second implementation of the `Controller` block, now takes responsibility for importing data to and exporting data from the DE model.

With two implementations available, the Simple Controller model permits a modeller to experiment by locating different functionalities in different parts of the model (for example, moving the controller functionality from the CT model to the DE model) and comparing the results.

2.5 Usage

The Simple Controller is a simple example of a basic model. A good way to see how the model works would be to run and compare the two versions (the CT-only and the co-model). The CT-only model should be launched directly from 20-sim. To do this, right click on the high-level `Controller` block in 20-sim, select `Edit Implementation`, and select `CT`, and launch the

model. Alternatively one can select `Destecs` here and then launch the co-simulation from the **Crescendo** tool.

Chapter 3

Water Tank (Periodic)

3.1 Case Description

This model is an implementation of a water tank that is supplied by a flow of water. There is a valve at the bottom of the tank which can be opened so that water flows out of the tank, or closed to stop the exiting flow of water. Underneath the tank we assume that there is a drain to catch the outflowing water. The tank is equipped with a sensor to measure the current water level.

Figure 3.1 illustrates the tank, the water flowing in, the valve and the drain as components in a CT model, with a DE model providing a controller.

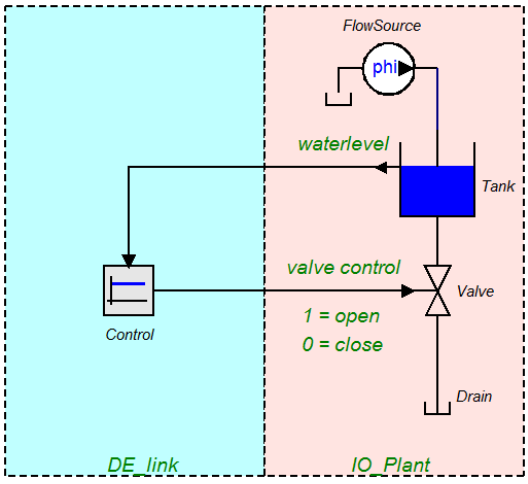


Figure 3.1: Overview of the water tank in the case study

We should like the water in the tank to remain between some maximum and some minimum level. If the water level falls below our minimum desired level then the valve must be closed to ensure water stops flowing out of the tank. If the water rises above our maximum desired level, then the valve must be opened to enable water to flow out. This particular water tank model accomplishes this by polling the water level sensor periodically, at intervals of n milliseconds, to

monitor the current water level, checking that it falls within acceptable boundaries, and taking some action if it does not.

3.2 Contract

The contract for the periodic water tank model includes one *monitored* variable of type real (called `level`) which represents the current level of water measured by the sensor. And there is one *controlled* variable of type bool (called `valve`) which is a signal to open or shut the valve.

In addition, there are two shared design parameters of type real: `maxlevel` and `minlevel`, which dictate the desired maximum and minimum level of the water respectively. The shared design parameters can be configured at runtime when executing a simulation of the model, by opening the debug configuration options in the **Crescendo** tool before the simulation is started.

3.3 Discrete-event

The DE model contains a `Controller` class, which has the main thread of control. An instance of `LevelSensor` is created to represent the sensor that measures the current water level. An instance of `ValveActuator` is created to represent the valve at the bottom of the tank. `LevelSensor` simply returns a value to represent the current level of water, whilst `ValveActuator` accepts a boolean value that sets the valve to be open or closed.

The `Controller` starts a single thread that loops repeatedly, waiting for some milliseconds *n* and then collecting a reading from the water level sensor. If the level is calculated to be above the desired maximum level then `Controller` sets the valve to be open, calling the appropriate method in the `ValveActuator` class. Similarly, if the water level is calculated to be below the desired minimum level then `Controller` sets the valve to be closed. The same thread continues to loop.

The `Controller` is deployed onto a single CPU by the `System` class.

3.4 Continuous-time

The CT model for the periodic water tank consists of two main parts, one to represent the link to the DE side of the model (`DE_link`) and one to represent the CT side (`I_O_Plant`).

The `DE_link` block handles interaction with the DE model itself. It contains a block, `Control`, which handles the input from the DE model (`valve`, to represent a signal for the valve actuator) and also the output from the CT model (`level`, which represents the current water level as read by the sensor). These are the shared variables declared in the contract.

The `I_O_Plant` block incorporates other blocks to represent the plant. There is a `FlowSource` to represent the source of incoming water, a `tank` to contain the water and a `Valve` at the bottom of the tank to control outward flow. Underneath the tank is a `Drain`.

3.5 Usage

Running a simulation of the periodic water tank illustrates how the water level in the tank rises until the maximum level is reached, at which point the valve is opened and the level falls until the minimum is reached. The valve is then closed again and the process repeats itself. One way to experiment with the model is to change the shared design parameters which dictate the values for the maximum and minimum water levels. These can be set in the Debug configuration window in the **Crescendo** tool. We recommend changing the values and running the co-simulation with these varying values to view the effect on the behaviour of the system.

Chapter 4

Water Tank (Event-Driven)

4.1 Case Description

The event driven water tank model is very similar to the periodic water tank model described in Chapter 3. It models a water tank that is supplied by a flow of water inwards. There is a valve at the bottom of the tank which can be opened so that water flows out of the tank, or closed to stop the outflowing water. Underneath the tank we assume that there is a drain to catch any water flowing out. The tank is equipped with a sensor to measure the current water level.

We should like the water in the tank to remain between some maximum and some minimum level. If the water level falls below our minimum desired level then the valve should be closed to stop the outflow. On the other hand, if the water rises above our maximum desired level, then the valve should be opened to enable water to flow out.

The event-driven model accomplishes this using events, which are triggered on the CT side of the co-model. When the water level sensor detects that the water is too high, it triggers a appropriate event and signals the DE model. The DE model then triggers the valve to open by signalling the valve actuator. Similarly, if the sensor detects a water level that is too low, the CT model triggers a different event and signals the DE model; the DE model then closes the valve.

4.2 Contract

The contract for the event-driven water tank model is very similar to that for the periodic water tank model. It includes one *monitored* variable of type real (called `level`) which is the signal representing the current level of water in the tank as measured by the sensor. And there is one *controlled* variable of type bool (called `valve`) which is a signal produced by the DE model to open or shut the valve.

In addition, there are two shared design parameters of type real: `maxlevel` and `minlevel`. The shared design parameters can be configured at runtime when executing a simulation of the model, by opening the debug configuration options in the **Crescendo** tool before the simulation is started.

Finally, the event-driven model also includes two declared events: `high`, which is an event triggered when the water level passes some maximum desired level; and `minLevelReached`, which is triggered when the water level falls below some desirable minimum level.

4.3 Discrete-event

The DE model contains an `Controller` class, which has the main thread of control. An instance of `LevelSensor` is created to represent the sensor that measures the current water level, and also an instance of `ValveActuator` is created to represent the valve at the bottom of the tank. An instance of `WatertankEventHandler` is created to receive the events that will be triggered by the CT model.

`LevelSensor` returns a value to represent the current level of water, whilst `ValveActuator` accepts a boolean value that sets the valve to be open or closed. We assume that the level sensor is able to self-detect and report some errors, and the `LevelSensor` class checks for these. The class `WatertankEventHandler` exists to catch the events triggered by the CT model and react to them. It reacts to the `high` event by opening the valve on the tank, and it reacts to the `minLevelReached` by closing the valve.

4.4 Continuous-time

The CT model for the periodic water tank consists of two main parts, one to represent the link to the DE side of the model (`DE_link`) and one to represent the continuous-time side (`I_O_Plant`).

The `discrete-event` block handles interaction with the DE model itself. It contains a block, `Control`, which handles the input parameter produced by the DE model and also the parameter for data output by the CT model. It also houses some logic for triggering the relevant events. Events in this case are triggered by a zero value being reached. The event `minLevelReached` is used to communicate to the DE model that the water level has fallen below some minimum desirable level of water, and is calculated by subtracting the minimum desired level from the actual current level; a result at zero results in the triggering of the event `minLevelReached`. Similarly, the maximum desired level is also subtracted from the current level; when the result is zero this triggers the event `high`.

The `continuous-time` block incorporates other blocks to represent the plant. There is a `FlowSource` to represent the source of incoming water, a `tank` to contain the water and a `Valve` at the bottom of the tank to control outward flow. Underneath the tank is a `Drain`.

4.5 Usage

As with the periodic water tank model, running a simulation of the event-driven water tank will illustrate how the water level in the tank rises and falls between the maximum and minimum levels. One way to experiment with the model is to change the shared design parameters which dictate

these levels and experimenting to observe the effect on the model's behaviour. These can be set in the Debug configuration window in the **Crescendo** tool.

Chapter 5

Paper Pinch

5.1 Case Description

The paper pinch co-model illustrates a device for manipulating paper (e.g., in a printer) and includes several different types of fault modelling. A pair of rollers sit one above the other and a motor is capable of rotating the bottom-mounted one. The top roller is spring loaded to produce a light downward pressure at all times. Rotating the bottom roller allows the pair to 'pinch' a sheet of paper and pass it between the two rollers, thus moving it through the printer. An overview of the rollers and motor and how they are connected can be seen in Fig 5.1.

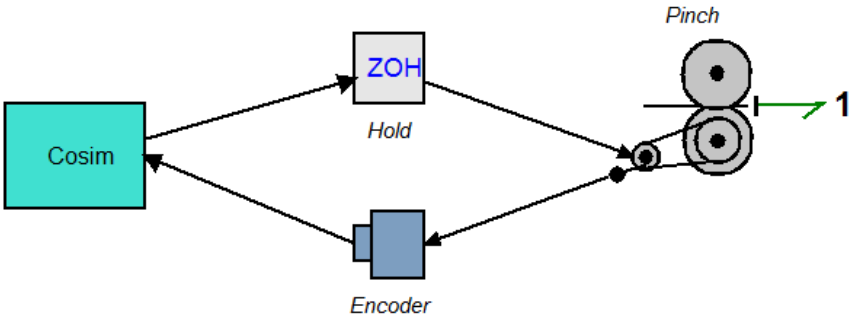


Figure 5.1: Diagram of a single pair of rollers in a printer ("Pinch"), the cosim Controller, an output line between controller and pinch (with a zero-order hold) and an encoder to feed data back to the Controller.

A real printer might typically have many sets of 'pinches' to manipulate the paper from an input tray through the printing process to the output tray. This often involves moving the paper in S-curves as well as in a simple linear direction. There is a requirement that the pinches in the system move paper at a synchronised speed. If one pinch feeds the paper too quickly or too slowly to its neighbours then the paper will fold and bunch up between the sets of pinches, or will be torn, and result in a paper jam. Alternatively the paper may become skewed, resulting in suboptimal print quality.

The main purpose of this co-model is to simulate some different types of fault handling. The co-model allows the modelling of two types of faults and two fault tolerant techniques.

5.2 External Links

The model was described in [PFG11]. This paper demonstrated the modelling and simulation of errors and fault tolerance mechanisms for embedded systems.

5.3 Contract

The contract contains four shared state variables of type real. One of these is responsible for signalling power to the actuator `pwm`. This increases or decreases the speed of the roller to control movement of the paper through the pinch. Three more shared state variables are used to read the current speed of the rollers, delivered by three separate encoders: `enc1`, `enc2` and `enc3`.

5.4 Discrete-event

The DE model implements a basic Decorator pattern (as described by Gamma *et al* [EJ95a]), consisting of the `Controller` which has the main thread of control, as well as an `ISensorInt` abstract class to monitor the readings from the pinch and an `IActuatorPWM` abstract class to produce a power signal to the actuators. In the model `Encoder` is provided as a possible implementation of the abstract `ISensorInt`, and `PWM` is provided as a possible implementation of the abstract `IActuatorPWM`.

The paper pinch model simulates several types of faults, and also incorporates several methods for fault tolerance (see Section 5.5 for details of faults which are injected). Firstly, the `Controller` is capable of implementing a voting system, which helps to detect cases where a single encoder has produced an incorrect value. This functionality is implemented in the `Voter` class, which is an alternative implementation of `ISensorInt`. `Voter` considers inputs from multiple sensors and translates this into a single value.

A further aspect of fault management is implemented in the `SafetyKernel` class, which is an alternative implementation of `IActuatorPWM`. `SafetyKernel` intercepts the controller's output to the power line and caps the signal to some absolute value. This prevents sudden large increases or decreases in power delivered to the actuator; if a mistake has been made and the `Controller` produces an inappropriate power output to the actuator, then this behaviour will curb the severity of the fault.

5.5 Continuous-time

The CT model includes a controller block (`Cosim`) which handles the interface with the DE model. The controller produces a power signal to an actuator, which then rotates the bottom roller in the

pinch device. Sensors detect the actual rotational speed achieved by the pinch and several encoders feed this information back to the controller. The input to the encoder is the speed of some shaft in revolutions per second and the output is a count indicating the number of rotations.

The purpose of this co-model is to simulate fault handling and so the co-model includes three encoders, to allow the model to simulate different types of faults. Each encoder includes a fault block that can intercept the input to the encoder and inject a possible fault. One of the three encoders is fault-free, one produces occasional bit-flips (sudden, large-value errors in the reading) and the final encoder produces a 'drift' error (low-value, cumulative, gradual errors in the reading). The fault blocks can be individually triggered to inject a fault.

5.6 Usage of fault tolerant features

The primary purpose of this model is to simulate different types of fault and fault tolerance, and so one obvious scenario is to test the fault handling. As previously described in Section 5.5, there are three encoders, each with a fault block. By default, one fault block in this model is set to produce a bit-flip error (characterised by a sudden large error in the value read in); one is set to produce a 'drift' type of error (characterised by small, gradual errors); and one fault block is configured to produce no error at all. Running a simulation with the default configuration will show output from all three encoders alongside their respective fault blocks; a change in the signal for a fault block indicates that it has activated and injected a fault into the model. There will be subsequent changes in other readings as the model detects and copes with the change. For example, when a bit-flip fault is triggered, the output from `Encoder1` can be seen to change value suddenly. If the model is employing the fault tolerant `Voter` and `SafetyKernel` implementations of the `ISensorInt` and `IActuatorPWM` classes, the `Voter` class can cope with the bit flip by considering the values of all three encoders and deciding that because the reading from `Encoder1` does not correspond with the others it is a fault and should be ignored; as a result the output value from the `Controller` does not change.

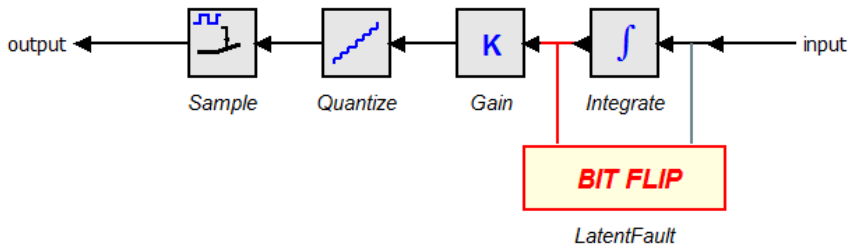


Figure 5.2: Diagram of the blocks inside an encoder for the paper pinch co-model, showing a fault block intercepting the input value.

A good next step would be to configure the fault blocks. Any of the fault blocks can be set to produce a bit-flip fault, a bit-drift fault, or no fault at all. To to this, open the CT model in 20-sim

and double click on an encoder block to open it up. Figure 5.2 shows an encoder with a fault block attached; the fault block, when triggered, tampers with the input signal and produces an erroneous output. To configure the fault block, right-click on it and select ”Edit Implementation”. This produces a menu with ”bitflip”, ”drift” and ”nofault” as options. Using this method, it’s possible to change the type of error that is produced for different encoders. It’s also possible to configure the time in the simulation at which the fault block is triggered to produce a fault. To do this, right-click on the fault block and select ”Parameters” (click ”yes” if 20-sim asks if it should check the model first). This produces a dialog box where parameters for the fault block are set; changing the value for the ”fault_time” parameter will change the time when the fault-block activates and injects a fault during the simulation. We recommend experimenting with changing the types of error on each encoder, and changing the times when faults are triggered, and then running the simulation to examine how the model copes with different types of fault: i.e., how quickly it can detect them and take some corrective action.

Chapter 6

Aircraft Fuel System

6.1 Case Description

The fuel system on-board aircraft is a complex system where fuel must be transferred between multiple tanks to ensure the engines always have a sufficient supply of fuel and that the aircraft is in balance. An overview of the fuel tanks and how they are connected can be seen in Fig 6.1.

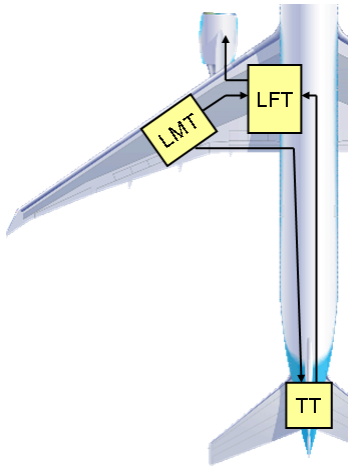


Figure 6.1: Overview of the fuel tanks in the case study

In the real system, tanks are also placed symmetrically in the right-hand side of the aircraft — these have been abstracted away in the model.

At takeoff, the Trimmer Tank (TT) placed in the rear end of the aircraft must be empty for safety reasons, but once cruise altitude has been reached the trimmer tank must be filled to distribute weight and therefore obtain better balance of the aircraft. The Left Feeder Tank (LFT) must always have sufficient supply of fuel to the engine which continuously consumes fuel (more during takeoff than at cruise altitude). If the level of fuel in the feeder tank reaches a minimum threshold fuel must be pumped from the Left Middle Tank (LMT) to the feeder tank. If the middle tank becomes

empty, fuel must be transferred from the trimmer tank to ensure a sufficient level of fuel for the engine. Finally, before landing the aircraft the trimmer tank must be empties for safety reasons.

The purpose of the model was to model different faulty scenarios (e.g. failing sensor) and examine if the aircraft had sufficient fault-resilience mechanisms to ensure that the aircraft can land safely.

6.2 External Links

The model was used in a comparative study [WPD12], analysing the collaborative modelling capabilities of the **Crescendo** tool as well as the Ptolemy tool [BHLM94, DGG+99, EJL+03].

The aircraft fuel system case is inspired by the work of Jiminez et al. [JGSIS07] — the **Crescendo** model only includes the left-hand side of the aircraft fuel system and the outermost tank is removed. A similar case study modelled in Ptolemy is published in [DLSV11].

6.3 Contract

The contract contain three monitored variables of type real used to read the current fuel level of the three tanks: `LFT_lvl`, `LMT_lvl` and `TT_lvl`. To control the flow of fuel between the three tanks the DE controller uses the following three controlled signals of type bool to pumps in the system: `LMT2LFT_valve`, `LMT2TT_valve` and `TT2LFT_valve`. Finally, the contract contain two events used to trigger state change in the controller: `enter_cruise` is triggered when the aircraft reaches cruise altitude, and `enter_pre_landing` is triggered when the aircraft is close to landing and the trimmer tank needs to be emptied.

6.4 Discrete-event

The DE model consist of the `Controller` which has the main tread of control, as well as a `TTWatchdog` monitoring the readings from the TT fuel level sensor (used for detecting faulty readings). In addition, an abstract boolean actuator and the concrete implementation `PumpSwitch_CT` is also part of the model. There are two abstract sensors (one for normative and faulty behaviour) and a concrete implementation of both of these sensors.

The `Controller` and `TTWatchdog` are deployed on two separate CPUs running at 10MHz which are connected by a single BUS.

6.5 Continuous-time

On the top level of the hierarchy, the model consists of three main parts: I_O, the plant and a block used for the 3D animation. The I_O handles the passing of variables to/from the DE controller model. The animation block scales the level values from the three tank between 0 and 1 to be used

in the 3D animation. The plant block contains the three tanks as well as the switches controlling the flow between the tanks.

The three tanks are modelled identically, and only the max capacity and the starting fuel level are different. The tanks themselves ensures that the fuel level is always between the empty and full thresholds, that the fuel can only enter the tank if it is not full, and that fuel only can be pumped from the tank if it is not empty.

6.6 Usage

The fuel system model comes with a single scenario script used for triggering the faulty TT level sensor:

```
when time >= 3.9 do ( ct boolean TTLEVELERROR := true; );
```

Once the error is triggered in the CT model, the TT level sensor will start sending a *faulty value* of value 9e18 from the `TT_level_Fault` block of the Plant in the CT model. A `TTWatchdog` class monitors the readings from the trimmer tank, and if two consecutive faulty values are read the asynchronous operation `TTErrorDetected` in the `Controller` class of the DE model is invoked and the fault is handled.

Chapter 7

Morse Code Reader

7.1 Case Description

This model represents a device for decoding a message presented in Morse code. Morse code is a system which uses one or more “dots” and “dashes” to represent standard Latin characters. The dots and dashes can be represented as long or short flashes of light, as long and short audible tones, or as long and short marks on paper. In this case the message to be decoded is delivered as long and short marks on a strip of light-coloured tape. The marks represent dots if they are narrow and dashes if they are wide. The decoder has a sensor able to detect light and dark on the tape and a motor which is capable of moving the tape so that it passes in front of the sensor. When it is ready to read a message, the decoder produces a signal to the motor to move the tape continuously past the sensor, and periodically collects a reading from the sensor which indicates whether a light or a dark colour is currently visible on the tape. These readings are collected and used to determined whether the tape is currently displaying a dot, a dash or a space. Information on dots and dashes and the spaces between them is collected in turn, and then translated into standard Latin characters using a lookup table. Figure 7.1 presents a diagram of the Morse decoder device, showing the tape with the message, the motor that moves the tape, the sensor to read the tape and the controller which is linked to both sensor and motor.

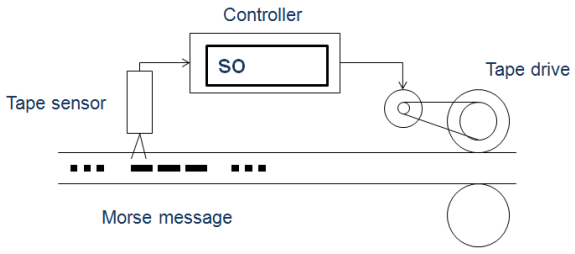


Figure 7.1: Overview of the Morse decoder

It’s assumed that the message which is encoded is preceded by a standard Morse “attention” sequence and is followed after its completion by a standard “end of work” sequence. The following

assumptions are also made:

- a dash is three times the length of a dot
- there is a space (no signal) the same length as a dot between dots and/or dashes
- there is a space the length of three dots between two separate ASCII characters
- that there is space the length of seven dots between two ASCII words

The exact length of the mark used to represent a dot and the mark used to represent a dash are unknown and must be calculated by the decoder before message can be decrypted. Therefore the Morse code reader firstly detects the "attention" sequence that precedes the message, and from this it performs some calibration to determine the width of a dot and the width of a dash. The width is determined in terms of the number of samples taken which were not background colour.

7.2 Contract

The contract for the Morse decoder model contains two variables of type real. A *monitored* variable, `morseSensorShared`, represents input from the sensor that detects light and dark. A *controlled* variable, `motorVoltageShared`, is used to produce a signal to the motor which moves the tape past the sensor.

The contract also includes two shared design parameters of type real: `AToDResolutionBits` and `AToDNoiseBits`. These are used for configuring the noise and resolution of the input data so that fault tolerance mechanisms may be exercised.

7.3 Discrete-event

The DE model contains an `AbstractMorseReader` class, which assumes the role of a controller and has the main thread of control. At runtime a concrete implementation of the `AbstractMorseReader` is provided by the `ModalController`, which in turn instantiates abstract classes to represent the actuator for the motor (`AbstractActuatorReal`) and the sensor (`AbstractFilteredSensorReal`). At run-time concrete implementations of these classes are provided by the `ActuatorReal` class and the `FilteredSensorReal` class respectively. Designing a controller that primarily handles abstract classes makes it easier to replace or alter the concrete implementation at a later date if necessary.

`ActuatorReal` allows the DE model to set the voltage for the motor to move tape past the sensor. `FilteredSensorReal` allows the sample size to be set (how many samples are used to determine if something is black or white, determined during calibration), as well as reading in a value from the sensor.

Some other classes are also provided. The `MorseLookup` class provides a method to translate a sequence of dots and dashes into ASCII characters. The `DotDashOrSpace` provides methods to check for the existence of a dot, a dash or a space given some input from the sensor.

The process of decoding the Morse message involves a number of different stages, or modes, of operation. The `ModalController` class stores a variable, `controllerMode`, which is of type `AbstractControllerMode` and is responsible for providing the methods to be employed during the current mode. During initial operation `ControllerModeMeasureSignalNoise` provides the concrete implementation of `AbstractControllerMode`; this class examines the input whilst the tape remains stationary, in order to determine the signal to noise ratio. A higher level of noise results in a lower overall speed being selected for reading the rest of the tape. The motor is off during this calibration phase but on for all other phases. Next the concrete implementation of `AbstractControllerMode` is replaced with the `ControllerModeMeasureDotDashLength` class, which provides methods to handle the calibration process of identifying the attention sequence and measuring dot length. Then the mode changes again and the functionality of the `AbstractControllerMode` is provided by the concrete implementation `ControllerModeReadMessage`. This mode class provides methods to collect sensor readings at appropriate sampling intervals and to translate collections of dots and dashes. Finally the concrete implementation is provided by `ControllerModeIdle`, which simply ensures that the motor voltage is set to zero.

The `AbstractMorseReader` (the controller class) is deployed onto a single CPU by the `System` class.

7.4 Continuous-time

The CT model consists of three main blocks, `controller`, `Motor` and `morseSensor`, in addition to a single block which represents the plant. The `Controller` block handles the interface between the CT model and the DE model. The `Motor` block resides between the `Controller` and the plant, and accepts an input which is the voltage to be applied to the motor itself.

The `morseSensor` block represents the sensor and accepts some input from the plant, handles A/D conversion and produces an output for the `Controller`. The output indicates how dark was the currently visible section of tape.

7.5 Usage

The model uses some data files to represent tapes with Morse code messages, and some examples are provided with the model. Each message is represented in three files: an image is provided purely for the user to visualise what the sensor should be able to see; a file *message.txt* represents the same data, but is actually used by the sensor to generate data; and a further file *message-length.txt* is used to ensure that the image is displayed to scale. One way to get started with the model is to try decoding different messages. This simply involves running the model with the included sample messages. A further step could be to generate your own message files and test out the decoding process. A Java *.jar* file, `TextToMorseTape.jar`, is included alongside a readme file alongside the CT model. This jar file can be used to generate the three message files required, given some text.

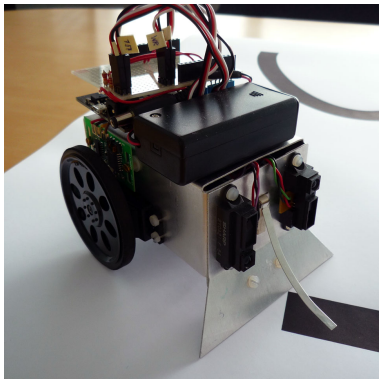
The model needs to deal with the difficulty of reading a signal which is delivered against some background noise. The model copes with this by analysing the signal-to-noise ratio before it begins and setting a lower speed for reading more "noisy" data. This allows the decoder to take more samples for each of the dot or dash characters, increasing confidence that data is accurately interpreted. An interesting experiment for the Morse decoder model involves varying the amount of background noise that it is expected to cope with. This is set as a shared design parameter in the contract, and can be set at runtime by opening the Debug configuration for the **Crescendo** model. Running the model with the same input files and higher levels of background noise should see a slower decryption process because the tape moves more slowly and more samples are collected, whilst decreasing the background noise should see the decoder produce the decrypted message more quickly.

Chapter 8

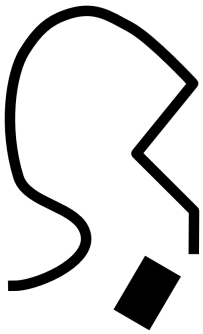
Line Following Robot

8.1 Case Description

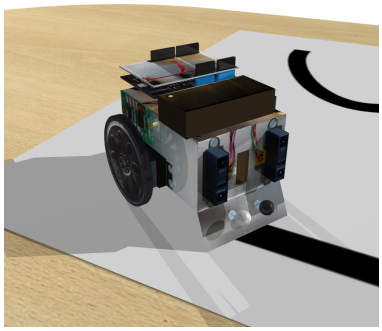
The line following robot is a co-model that demonstrates how design space exploration is supported in **Crescendo**. The co-model simulates a robot that can follow a line painted on the ground. It is assumed that the colour of the line is known and that it is differentiated from the background colour (which is not known), but the line may exhibit gentle or sharp corners or angled corners in any direction. The robot uses a number of sensors to detect light and dark areas on the ground and has two wheels, each powered individually with motors to enable the robot to make fine-graded changes in direction.



(a) An R2-G2P robot



(b) A line-follow path



(c) 3D representation of the R2-G2P

Figure 8.1: The line-following robot

There are a number of design decisions to make when designing the robot. For example, possible variations might include changing the number of sensors or changing their positions (sensor position can be adjusted laterally or longitudinally). Any given design can be assessed by comparing distance travelled, which will be shorter for more accurate robots, and average speed, which

will ideally be as high as possible. **Crescendo** facilitates exploration of the design space by automating simulations with different design parameters. For example, **Crescendo** can automatically simulate a model with two sensors set initially close together, then increasing the distance between them by some distance `n` until a maximum distance is reached. The results of each simulation can then be compared to determine the ideal robot design.

The co-model supplied here uses two sensors to detect light and dark, and two motors (one for each wheel). Two encoders (also one for each wheel) close the feedback loop so that actual wheel rotations can be monitored and the output to the motors adjusted accordingly.

The robot moves through a number of phases as it follows a line. At the start of each line is a specific pattern that will be known in advance. Once a genuine line is detected on the ground, the robot follows it until it detects that the end of the line has been reached, when it should go to an idle state. For modelling purposes the co-model accepts a textual representation of a bitmap as the surface with a line to be followed.

8.2 Contract

The contract contains eight shared variables. Two *controlled* variables of type real produce signals for the actuators that power the motors for the left (`servo_left`) and right (`servo_right`) wheels. And two *monitored* variables of type real are used to feed back information about the rotations of the left (`encoder_left`) and right (`encoder_right`) wheels.

Two *monitored* variables of type real are used to represent inputs from line-following sensors that can detect areas of light and dark on the ground. In this model there are two sensors, `lf_left` and `lf_right`, but other models may carry more or fewer line-following sensors.

Two *monitored* variables of type bool are used to indicate whether a fault has been detected on the left wheel (`lf_left_fail_flag`) or right wheel (`lf_right_fail_flag`). It is assumed that the CT model can self-detect and report some faults to the DE model.

The contract also includes seven *shared design parameters*: wheel radius, in metres (`wheel_radius`)); encoder resolution, in counts per revolution (`encoder_resolution`); the separation of the line-following sensors from the centre line, in metres (`line_follow_x`); distance forward of the line-following sensors from the centre of the robot, in metres (`line_follow_y`); an array giving initial position `x`, `y`, `theta` (`initial_position[3]`); and two shared variables representing controller aggression in terms of maximum wheel speed (`fast_wheel_speed`) and turn ratio (`slow_wheel_ration`), both in the range [0,1].

8.3 Discrete-event

The DE model makes heavy use of inheritance, providing abstract classes for controller, sensors and actuators. The servos are accessed through objects of the `SpeedServo` class, which inherits from the `IActuatorRealPercent`; and the encoders are read through objects of the `Encoder` class, which inherits from the `ISensorReal`.

The line-following sensors make greater use of the benefits of inheritance. The raw sensor readings are accessed through objects of the `IRSensor` class, which inherits from the `ISensorInt8` class. Because the CT model of the line-following sensors includes sensor noise, the `Filtered-IRSensor` is used. This class contains an `IRSensor` and maintains a floating average using the last five sensor readings. This class also inherits from the `ISensorInt8`, which means that an `IRSensor` object can be swapped for a filtered object seamlessly. This is an example of the standard *decorator* pattern [EJ95b].

As mentioned above, the sensors are calibrated to set when it is considered to be reading black. This can change if the ambient light is increased in the model. The calibration is set in a `BlackWhiteSensor` class, which holds an `IRSensor` object and has Boolean operations to determine if the sensor is seeing black or white.

The controller is modal, meaning that with each 'phase' (initialisation, sensor calibration, line-following, etc.) is realised as a mode, represented by the `IMode` interface. This interface defines four operations: `Enter`, called when the controller changes to this mode; `Step()`, called each control cycle; `Exit()`, called when the controller finishes this mode; and `Done()`, which allows a mode to indicate it has finished its task. The two abstract classes `AbstractMode` and `AbstractModeTimed` provide empty implementations of these operations. The timed mode keeps track of how long it has been running, and `Done()` will yield **`true`** after a specified time.

The top-level controller class is the `AbstractModalController`. This class holds a reference to each sensor object and provides getter methods to access these objects. The abstract mode classes hold a reference to the `AbstractModalController`, so that each mode can access the sensors and actuators. The `LineFollower` class is a subclass of this class, which instantiates the modes it requires to follow the line, which are held in a map. The identifier of the current mode is recorded in the `mode` variable and modes are changed with the `ChangeMode` operation. The `CheckModeChange` operation contains the logic for changing mode.

The following modes are followed in order: `WaitMode`, which gives the sensors time to start up; `CalibrateMode`, which calibrates the sensors using the predetermined starting pattern; `FindLineMode`, which drives forward until the line is found; and finally `FollowMode`, which follows the line.

If one of the sensors fails, the controller switched to `SingleFollowMode`, which uses the one remaining working sensor to follow the line. Modes can also cause a change of mode by returning the identifier of a mode from the `Step()` operation. This functionality is used by the `SingleFollowMode` class to switch to `RefindLineMode` when it loses the line. The `RefindLineMode` sweeps back and forth until it sees black, then returns to `SingleFollow-Mode`.

The `System` class instantiates the `LineFollower` object and deploys it to a single CPU. All sensor and actuator objects are created in the `IOFactory` class, which provides so-called "factory" methods that return the requested sensors. The `IOFactory` object is created in the `System` class and passed to the `LineFollower` object. The periodic thread is defined in the `Thread` class. This class holds a reference the controller and IO factory. This means that all the sensors and actuators can be synchronised at once, then the controller is called. This improves simulation speed.

8.4 Continuous-time

The CT model (pictured in Figure 8.1) is broken down into blocks that broadly represent physical components. The central *body* block is a bond graph representation of the physics, with bond graph models of the wheels (*wheel_left, wheel_right*) and servos (*servo_left, servo_right*) connected on either side. The encoder blocks (*encoder_left, encoder_right*) produce 44 counts per revolution of the wheel. Two line-following sensor blocks (*lf_left*, *lf_right*) are also connected to the body, as well a *map* block. The encoder, line-following sensor and servo blocks are all connected to a *controller* block that handles exchange of data with the DE model.

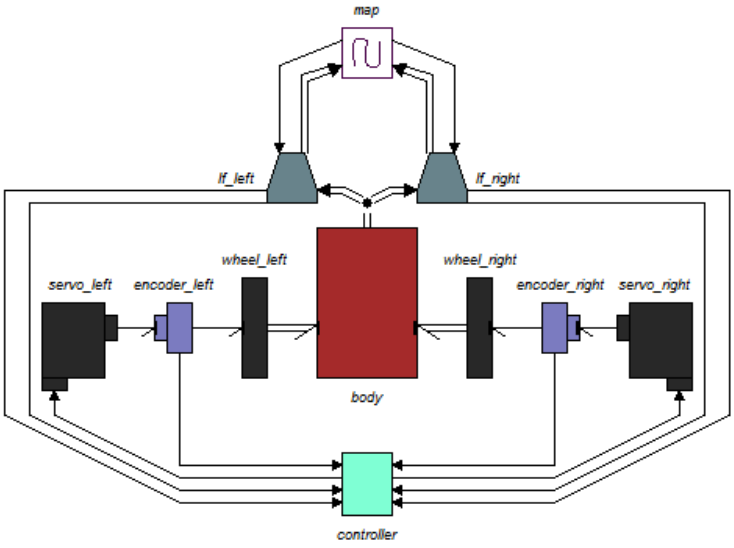


Figure 8.2: Continuous-time model of the R2-G2P

The line-following sensor (seen at the top of Figure 8.2) take the position and orientation from the *body* block and calculate their position in the world using the `line_follow_x` and `line_follow_y` shared design parameters. This position information is passed to the map block, which takes a sample of values and passes a raw reading back to the sensors. The sensors then convert this to an 8-bit value, taking into account realistic behaviours: ambient light levels, a delayed response to changes, and A/D conversion noise.

The line-following sensor blocks also include a block to model a "stuck" failure (*stuck_fault*). This block has two implementations: the default (*no_fault*) does nothing to the signal and has an empty icon. The alternative is called (*fail_fixed_value*) and has a thick, red outline and causes the sensor to always yield a fixed value (the stuck values are defined in the *globals* block).

If the *fail_fixed_value* implementation is selected in the left sensor, then it will fail after 12 seconds, always yielding 0 (black). If the *fail_fixed_value* implementation is selected in the right sensor, it will fail when the value `lf_right_stuck_value` is changed from -1 to a value between 0 and 255 by a script. The sensor will then yield whatever value is set, so this sensor can be made to get stuck on white, for example.

8.5 Usage

The line-following robot model is a good opportunity to see how **Crescendo** can support design space exploration. A good starting point is to run the co-model (using the `LineFollowingRobot` launch configuration), examining the speed and accuracy (distance) with which it can complete various lines. You can then run the `LineFollowingRobot_Outside` launch, in which the shared design parameter `line_follow_x` has been increased so that the sensors are "outside" the line on either side. Note how the simple line-following algorithm still works.

Next you can try your own combinations by altering `line_follow_x` and `line_follow_y` and observing the results. The robot generally needs to be able to distinguish when it has encountered the end of the line, and when it has encountered a sharp angle that may take the line behind its current field of view. Changing the forward/back position of the sensors may make this task easier or more difficult, whilst changing the width between the two sensors also has an impact, by altering the resolution of the information which can be gathered about the line.

To try various designs automatically, you can define an "ACA Launch" configuration. Select `LineFollowingRobot` as the "Base Configuration", then on the "SDPs Sweep" tab, then try the following values: `line_follow_x` from 0.01 to 0.05 by 0.02, and `line_follow_y` from 0.01 to 0.13 by 0.06. The **Crescendo** tool will then run 9 co-simulations generated from the above numbers.

Beyond this step, you could try to improve the robot's line-following. First, you could change `fast_wheel_speed` and `slow_wheel_ratio` (between 0.0 and 1.0) to alter the controller aggression. The robot could be made to drive more smoothly if a *proportional* control mode was used, which applies more or less turn depending on how far from the line it is. You could change the `FollowMode` class, or create you own class and instantiate it in the constructor of `LineFollower` class.

Increasing the number of sensors would also allow for performance improvements. You can duplicate a sensor in the CT model and connect it up. You would need to alter the *map* and *controller* blocks to handle the new signals; instantiate more sensor objects in the DE model and make them accessible to the modes; and add a new entry to the contract / `vdm.link` file. Chapter 9 describes five different models of a similar line-following robot, each exhibiting different design choices in terms of number and position of sensors, as well as different methods for identifying the line against a undetermined background.

Chapter 9

DESTECS Summer School models

9.1 Introduction

In July 2012 the DESTECS project hosted a summer school for postgraduate level students and practitioners, which included multi-day projects in groups. Five groups in total were set the assignment of developing their own line-following robots, including setting appropriate design priorities and testing out different possible designs to achieve the optimum result. We include these models here, as they provide a good demonstration of how DESTECS can facilitate the exploration of the design space. These models are provided as-is, showing the real work of first-time DESTECS users working on a time-limited project, and should be viewed as such. Photos of each group, along with their designs, are shown in Figure 9.1.

Each group had the same brief: to develop a line-following robot model as described in Chapter 8. Groups aimed to create the 'best' performing robot, where performance was measured by total time taken to follow a given set of lines (not known in advance) and how accurately the robot measured these lines. Models were required to measure the time taken and the distance of the line for this reason. It is assumed that the robot has a fixed starting position which begins with a predictable pattern on the ground before the line begins.

As described briefly in Chapter 8, there are several design aspects to consider:

- There was no limit on the number of sensors that could be added, and there was considerable possibility for groups to experiment with this. Whilst it is possible to design a robot with only one sensor that would fulfil the requirements of the exercise, adding more sensors may allow the robot to plot a more accurate course.

- Groups were free to change the position of the sensors as they saw fit. A particular problem is posed by the difficulty of detecting sharp, acute angles, resulting in the line doubling sharply back on itself. Some designs of robot will find these scenarios difficult to detect and will lose track of the line's location. This can be made easier or more difficult by placing sensors further forward, further back, wider apart or closer together, and several groups did opt to experiment with alternative layouts to resolve the difficulty.

- The motors that power the wheels for this particular robot can accept different voltages, which result in varying speeds being delivered to the wheels. Groups can decide whether to implement higher speeds if they are confident of the line's course. They can also decide how to effect a turn. It is possible to turn with both wheels still moving forward (at different speeds); this approach prioritises speed. Alternatively, it's also possible to halt one wheel or even reverse it in order to effect a turn; this approach minimises the distance travelled as the robot corners.

We present here the models that were created as they exhibit quite different design choices. Note that some library class files were provided to ensure that all groups had the same basic environment in which to operate and to produce a 3D simulation of the finished model; these take the form of some classes in the DE model and some blocks in the CT model. In particular, all the CT models include a block `RobotPhysics`, which incorporates blocks `LeftWheel` and `RightWheel`, powered by blocks `LeftMotor` and `RightMotor`. Each wheel/motor combination also links to an encoder (`encoderLeft` and `encoderRight`). Outputs from this block include data on the robot's current position and angle, and the position of each wheel. Inputs arrive from outside and are delivered to the separate motors.

All groups were required to monitor rotations of the wheels for their robot. This is accomplished using a sensor that detects light and dark which is mounted facing each wheel; it's assumed that the wheels of the robot are marked with light and dark stripes that can be counted as they move past the sensor.

Finally, the groups were given an initial set of shared design parameters to use in their contract, including: two real parameters for controlling the position of the sensors longitudinally and laterally; one real for setting the encoder resolution; one real for setting the initial angle or orientation of the robot; and finally one array which contains the initial starting co-ordinates (in the format x,y). Groups could decide whether to employ these or not; some groups opted not to and removed them from the contracts, whilst others did not use all of the parameters but left them in the contract for future development purposes.

9.1.1 Usage

We do not provide individual suggestions for each of these models as their outward design parameters are very similar. One good starting point is to run a simulation for each robot, try out different line patterns, and identify an optimal design out of the designs presented. Each robot will handle different types of corners or curves differently, and some have optimisations that allow them to move faster over straight sections, or which allow them to turn more quickly on a sharper corner. Other robots prioritise an accurate route over a speedy one.

Different groups chose to employ different numbers of sensors, and to vary the positioning. We recommend executing simulations with the robots to see how different sensors may be employed, and effect that it has on overall accuracy and speed. The groups used the following designs:

- **Group 1** used two sensors, placed symmetrically on the left and right. Their model prioritises speed when turning.

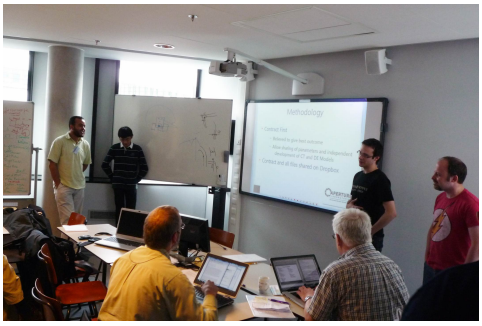
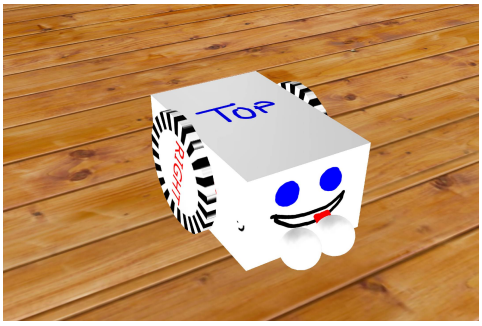
- **Group 2** used four sensors, two placed symmetrically (left and right) at the front of the robot, and two more also placed symmetrically towards the rear of the robot. Group 2 also decided to vary the speed of the robot, so that on straight sections of track the robot moves more quickly, and also executes tight corners as fast as possible (the four-sensor design helps with this).

- **Group 3** used three line sensors placed in a row, with the central sensor placed slightly further back than the left and right sensors. Their model minimises distance taken to execute a turn by reversing one wheel.

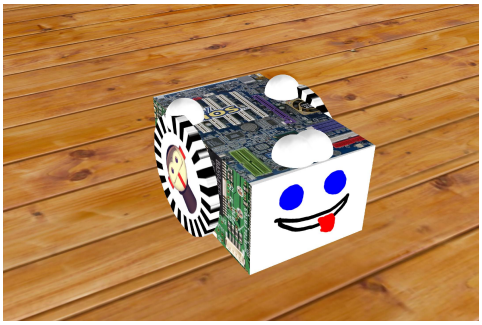
- **Group 4** also used three sensors, placed in a row. They aimed to keep the central sensor directly over the line, so the extra middle sensor helps to improve course accuracy.

- **Group 5** used two sensors, placed symmetrically on the left and right. Their model minimises distance taken to execute turns.

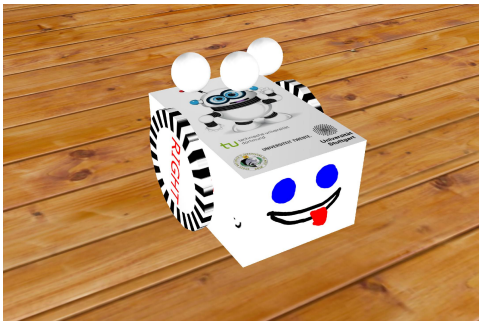
The models generally use shared design parameters to control the longitudinal and lateral positioning of the two sensors. Another worthwhile experiment therefore is to change the values of these parameters to change the sensor position, and see how this affects performance of the robot (distance travelled and time taken are recorded by each model, so performance with different settings can be compared). Drastic changes to the position of the sensors may even cause a robot's line-following algorithm to break, as assumptions made by the algorithm about sensor location no longer hold true. The initial values of the parameters (for groups that use them) can be set when running the model by opening the Debug window of the DESTECS tool before running the co-simulation.

(a) Group 1



(b) Group 2



(c) Group 3


(d) Group 4


9.2 Group 1

9.2.1 Case Description

Group 1 opted for a design with two sensors (left and right), placed symmetrically. The layout chosen is not completely finished and may not handle all maps equally well - it may cope with some angles and corners better than others. Group 1's model is an example of a model that prioritises speed when executing turns, as both wheels continue to turn as fast as is practical whilst turning. Some other models made different decisions and either halted one wheel entirely or reversed one wheel to execute a turn.

9.2.2 Contract

The contract contains six variables of type real. Two *monitored* variables, `leftLSValye` and `rightLSValue`, are used to represent inputs from the line-detecting sensors on the left and the right respectively. Two further *monitored* variables, `leftEncoderValue` and `rightEncoderValue`, are used to represent input from the sensors used to count revolutions of the wheels. This information is necessary in order to calculate distance travelled, a requirement of the exercise.

Two *controlled* variables of type real, `leftMotorDC` and `rightMotorDC`, are used to produce signals to the motor for the left and right wheels respectively.

The contract also includes the five shared design parameters described for all groups in Section 9.1.

9.2.3 Discrete-event

In the DE model, the `System` class instantiates two instances of the `Sensor` class to represent sensors placed on the left (`leftLineSensor`) and the right (`rightLineSensor`). In addition `System` instantiates two instances of `Actuator` (`leftPWM` and `rightPWM`) to provide signals to the motors for the left and right wheels.

To provide a closed feedback loop, two more `Sensor` instances are created to represent the encoders monitoring actual revolutions turned by the wheels on the left (`leftEncoder`) and on the right (`rightEncoder`).

Finally, `System` creates an instance of `ModalController` to provide controller functionality, and deploys it onto a single CPU. `ModalController` provides the primary logic for the robot. A number of modes are defined; an initial mode sees the robot start moving forward, hunting for the initial pattern. Next is a calibration phase where the robot continues as it follows the initial pattern, and a line-following phase where the actual line begins. A final phase sets the robot at rest after the end of the line is identified. The robot reduces the speed of the left wheel to turn left if the left sensor detects the line on the ground, and it reduces the speed of the right wheel to turn right if the right sensor detects the line.

9.2.4 Continuous-time

As with all the other groups, the CT model includes blocks to provide an implementation of robot physics (`RobotPhysics`) and data for the 3D simulation (`DataFor3D`).

A `controller` class handles interaction with the DE model. The CT model includes two blocks to represent the encoders for the wheels on the left (`encoderLeft`) and on the right (`encoderLeftRight`). Each encoder accepts input (data representing the number of black or white areas on the wheel which have moved past the sensor), and incorporates blocks that convert this data into a count of the number of revolutions the wheel has made.

Input from the `controller` is passed to `RobotPhysics` for some processing and is then made available to the encoders. Output from `RobotPhysics` is also passed to a block `posConverter` which is responsible for calculating current position. This block interacts with blocks `floorMapLeft` and `FloorMapRight`, which act as lookup tables for interpreting what is currently visible in the map.

9.3 Group 2

9.3.1 Case Description

Group 2 employ four line-following sensors for their robot - two (for the left and for the right) placed forward and two (also for left and right) placed further back. The sensors mounted towards the back of the robot are used to identify sharp turns, when the course of the line may be so sharp that it turns back behind the robot's current position. Front-mounted sensors may find such a scenario difficult to detect, assuming that if the line cannot be seen ahead or on one side then it must have reached the finish point.

Group 2 also made the decision to vary the speed of the robot, so that on tighter corners or on more straightforward elements of the track, the robot can move more quickly. This is possible because the presence of two extra sensors towards the back of the robot in addition to two on the front allows corners and curves to be categorised as shallow (visible to the front sensors) or steep (visible to the back sensors and not to the front ones). Different turning speeds can then be selected for these two types of corner.

9.3.2 Contract

The contract contains six variables of type real. Four *monitored* variables, `lineSensorLeft`, `lineSensorRight`, `lineSensorLeftB` and `lineSensorRightB`, are used to represent inputs from the four line-detecting sensors (front left and front right, back left and back right respectively). Two further *monitored* variables, `wheelPosLeft` and `wheelPosRight`, are used to represent input from the sensors used to count revolutions of the left and right wheel. This information is necessary in order to calculate distance travelled, a requirement of the exercise.

Two *controlled* variables of type real, `motorVoltageLeft` and `motorVoltageRight`, are used to produce signals to the motor for the left and right wheels respectively.

Group 2 did not make use of any shared design parameters.

9.3.3 Discrete-event

The `System` class in Group 2's model begins by creating six instances of the class `AbstractSensorReal` for the six sensors (four for line-detecting and two to act as encoders for the wheels) and two instances of `AbstractActuatorReal` for the two actuators. The `Controller` class acts as a controller and has the main thread of control; the `System` class deploys the `Controller` to a single CPU.

At run time, `ActuatorReal` provides an implementation of `AbstractActuatorReal`. This class simply passes a signal to the motors. Implementations of the sensors are provided by different classes for the sensors used as encoders, and the sensors uses as line-detectors. For the encoders, the class `EncoderSensor` is used. This class includes some short logic to convert the input from the sensor representing the wheel encoder into a value for rotations of the wheel. For the line-detecting sensors, implementation is provided by the class `SensorReal`, which simply handles data produced by the sensor.

Like many other groups, Group 2 created different classes to contain the functionality required for the different modes of operation. The `Controller` class stores a variable internally (`ControllerMode`) which is of type `AbstractControllerMode`. At runtime different concrete implementations of this abstract class are used during different modes of operation. The first implementation is provided by the `InitilaizeControlMode` class, which proceeds until it can be confirmed that the initial pattern on the ground has been detected and that the robot has proceeded beyond it. During the next mode the concrete implementation of the `AbstractControllerMode` is provided by `LineFollowingControlMode`. This class proceeds on a step-by-step basis. For each step, the class first gathers readings from each of the four line-detecting sensors. In cases where both front sensors can see the line, then the robot continues ahead at top speed. If the right front sensor can detect a line but the left sensor cannot, then the robot should turn right. Similarly, if the left front sensor can detect a line but the right sensor cannot, then the robot turns left. `LineFollowingControlMode` checks for situations where one of the rear-mounted sensors can detect a line that is not visible to the front-mounted sensors. In this case, it assumes that there is a sharp corner in the line's course, and executes a sharp turn in that direction. If all four sensors detect no line, then the robot is deemed to have finished the course.

Attempts are made to try and keep the speed as high as possible whilst executing a turn; Group 2's robot keeps one wheel at `topSpeed` and drops the speed of the other. The model also differentiates between sharp corners and shallow turns, and uses different speed settings on the wheels to ensure that the robot turns more quickly in a shorter distance on a sharper corner.

9.3.4 Continuous-time

As with all the other groups, the CT model includes blocks to provide an implementation of robot physics (`RobotPhysics`) and data for the 3D simulation (`DataFor3D`) .

Group 2's CT model employs a `Controller` block to act as an interface with the DE model. The model also includes two blocks to represent the two encoders (`EncoderL` and `EncoderR` for the left and right encoders respectively). Data incoming from the `Controller` for the encoders is processed by `RobotPhysics` and is then passed to the encoders, which apply some processing and generate an output for the wheels.

There are four blocks to represent the four sensors (`Sensor_Left_Back` and `Sensor_Right_Back` are the left and right sensors at the back and `Sensor_Left_Front` and `Sensor_Right_Front` are mounted at the left and right front of the robot). Each sensor block accepts an input which is the robot's current position. It incorporates a lookup table `Table2D` to calculate the currently visible section of floor and produces an output representing the currently visible floor colour.

Signals generated by all four sensors are processed by `RobotPhysics` and results are passed back to the `Controller` for sharing with the DE model. Finally, the incoming data for the encoders, and the output generated by all four sensors, is also processed by `DataFor3D` to enable a 3D simulation.

9.4 Group 3

9.4.1 Case Description

Group 3 opted for a design with three line-detecting sensors. Two sensors are mounted in a row with one sensor on the left, one on the right. The third sensor is mounted between these two, but is positioned behind them.

9.4.2 Contract

The contract contains five *monitored* variables of type real. Three of these variables provide input from the line-detecting sensors to the DE model: `sensorLeft`, `sensorRight` and `sensorFront`, are used to represent inputs from the line-detecting sensors on the left, on the right, and in the centre respectively. Two further *monitored* variables, `wheelCountLeft` and `wheelCountRight`, are used to represent input from the encoders for the wheels.

There is one *controlled* variable - `motorVoltages` - which is an array with a length of 2. This variable is used to communicate the two signals for the motors (one for each motor) from the DE model to the CT model.

The contract also includes five shared design parameters. Some of these are described for all groups in Section 9.1 (`encoder_resolution`, `linefollow_lateral_offset` and `linefollow_longitudinal_offset`). In addition to these, Group 3 added two other shared design parameters: `r_wheel`, which is of type real, is used to store the calculated radius of the wheel and is used to compute the total revolutions of the wheel and `routeIndex`, which is also of type real but is not actively used in the model supplied here.

9.4.3 Discrete-event

The DE model created by Group 3 begins by creating three instances of the `Sensor` class: `sensorLeft`, `sensorRight` and `sensorMiddle` represent the left-hand, right-hand and central sensors respectively. It also creates two instances of the `Wheel` class, to represent the left (`wheelLeft`) and right (`wheelRight`) wheels. The `Sensor` class simply handles the production of data from the sensor, and the `Wheel` class handles the passing of a signal to the motor for the wheels.

The `System` class creates these instances as well as a controller, which holds the main thread of control.

The `Controller` class implements all the functionality necessary to move through the various modes of operation. The class begins with calibrating, which involves detecting the initial starting point and moving the robot forward past the initial pattern and onto the line itself, facing in the correct direction. Next the robot enters a line-following phase. If the left sensor can detect the line then the robot turns left, and if the right sensor can detect the line then the robot turns right. If none of the three sensors can detect the line, the robot assumes that it has reached the end of the line and halts.

Group 3's robot minimises the distance taken to turn by delivering the maximum negative voltage to one wheel and the maximum positive to the other. The negative voltage results in one wheel turning in reverse, minimising the distance travelled as the robot corners.

`System` deploys `Controller` to a single CPU.

9.4.4 Continuous-time

Group 3's CT model includes a `Controller` block which acts as an interface between the CT and DE models. As with all the other groups, the CT model includes blocks to provide an implementation of robot physics (`RobotPhysics`) and data for the 3D simulation (`DataFor3D`). There are blocks to represent the sensors: `LeftSensor`, `FrontSensor` and `RightSensor` represent the left, central and right sensors respectively. All sensors incorporate a lookup table `Table2D` and some processing to determine the currently visible floor colour.

Incoming signals from the DE model for the motors are passed from `Controller` to the `RobotPhysics` block first of all. `RobotPhysics` includes functionality to move the robot within its environment. Output from this block is passed to the sensors so that they can sample the currently visible section of floor, as well as to the `DataFor3D` block which handles 3D simulation. Finally, the `Controller` block accepts data back from the three sensors and from `RobotPhysics`. The data produced from each sensor is an indication of the colour currently visible on the floor.

9.5 Group 4

9.5.1 Case Description

Group 4 decided to use three sensors - a left, a right and a centre - for their line-following robot. They decided that the robot should attempt to keep the line lying directly below the central sensor, which should result in a more accurate path. Having three sensors rather than two also allowed for more fine-grained adjustments to be made to the course, which are not possible to achieve with fewer sensors.

9.5.2 Contract

The contract contains seven variables of type real. Three *monitored* variables are used to represent inputs from the line-detecting sensors on the left (`lineSensorLeftShared`), in the centre (`lineSensorCenterShared`) and on the right (`lineSensorRightShared`). Two further *monitored* variables are used to represent input from the sensors used to count revolutions of the left wheel (`positionSensorLeftShared`) and the right wheel (`positionSensorRightShared`).

Two *controlled* variables are included: `motorControlSignalLeftShared` to produce a signal for the left-hand motor, and `motorControlSignalRightShared` to control the right-hand motor.

The contract also includes the five shared design parameters described for all groups in Section 9.1.

9.5.3 Discrete-event

The DE model for Group 4's model begins with the `System` class, which instantiates three sensors for the line-sensing task and two sensors for the wheel encoders, using the `AbstractSensorReal` class for all sensors. It also instantiates two actuators, using the `AbstractActuatorReal` class, to power the wheels, and a single instance of `LineFollowerController` to provide control functionality. Finally it deploys the model to a single CPU.

At runtime, the `SensorReal` class provides a concrete implementation of the abstract `AbstractSensorReal`. This class simply returns a value representing whether or not light or dark was detected. `ActuatorReal` provides the implementation of `AbstractActuatorReal` at runtime; this class simply handles the passing of values to the actuators.

Like many other groups, Group 4 also decided to employ different classes to provide the functionality necessary for different 'modes' of operation. The `LineFollowerController` stores a variable of type `AbstractControllerMode` which provides the functionality needed during the current mode of operation. As the robot moves from one mode of operation to another, the concrete implementation of this variable is changed so that different functionality is available at different times. The `LineFollowerController` begins with a calibration mode, where the robot moves forward until it detects and proceeds past the anticipated initial pattern.

The concrete implementation of `AbstractControllerMode` during this phase is provided by `ControllerModeCalibration`. When `ControllerModeCalibration` has detected initial pattern, proceeded past it and detected the start of the line, it signals to the `LineFollowerController`, which changes the concrete implementation of `AbstractControllerMode` to `ControllerModeFollowLine`. This class repeatedly retrieves values from all three sensors and makes a decision about how to move forward. In general, the robot attempts to keep the line lying above the central sensor. Therefore, if the left sensor and right sensor detect background and the central sensor detects the line, then the robot can move forward. If the left and central sensors see background and the right sensor can see the line, then the robot turns right, whilst if the opposite happens and the right and central sensors can detect the background and the left sensor can detect the line, then the robot turns left. 'Turning' involves setting a zero voltage on one motor and continuing at full speed with the other motor.

If all three sensors detect background, it's assumed that the end of the line has been found and the concrete implementation of `AbstractControllerMode` is changed to `ControllerModeIdle`, which brings the robot to a standstill.

9.5.4 Continuous-time

As with other groups, the CT model includes blocks to provide an implementation of robot physics (`RobotPhysics`), data for the 3D simulation (`DataFor3D`) and lookup tables to process information from the 2D image representing the floor (`Table2D1`, `Table2D2` and `Table2D3`). `RobotPhysics` accept inputs from the motor control blocks (representing voltages to be applied to the motors), and produces outputs for the motor/position sensors. Data processed by `RobotPhysics` is also passed to the three line sensors, which apply some logic to interpret the area of the floor now currently visible.

Group 4's CT model does not have one central block to handle interaction with the DE model; the shared elements of the co-model are found individually in relevant blocks. The model includes two motors, `K`, one for each wheel. Each of these provides input for a position sensor (either `positionSensorLeftShared` or `positionSensorRightShared`). The two position sensors map directly to shared variables declared in the contract (`positionSensorLeftShared` and `positionSensorLeftShared`) and so they are responsible for passing information to the DE model about the current position of the wheel.

The model also includes two blocks handling the interface with the motor signal (`motorControlSignalLeftShared` and `motorControlSignalRightShared`), which accept inputs from the DE model representing voltages to be applied to the left and right motors respectively.

Finally, the model also contains three blocks to represent sensors for detecting the line on the ground: `LineSensorLeftShared`, `LineSensorCenterShared` and `LineSensorRightShared`. These three blocks map directly to shared variables declared in the contract (`lineSensorLeftShared`, `lineSensorCenterShared`, `lineSensorRightShared`) and so they are responsible for passing information to the DE model from the three line-detecting sensors. Data output by `RobotPhysics` is first processed by a block of logic (one for each sen-

sor: `calculateLeftLineSensorPosition` for the left-hand sensor, `calculateCenterLineSensorPosition` for the central sensor or `calculateRightLineSensorPosition` for the right-hand sensor), which calculate the current position within the known floor area. This information is passed to blocks which perform some lookup functionality (`Table2D1`, `Table2D2` and `Table2D3`) and finally an output is produced from this for the line sensor to interpret.

9.6 Group 5

9.6.1 Case Description

Group 5's robot employs two line-detecting sensors, positioned left and right. The group elected to employ faster turns in the shortest possible distance by setting one wheel to maximum voltage and the other to maximum negative voltage (i.e., reversing) when turning on corners.

9.6.2 Contract

The contract contains six variables of type real. Two *monitored* variables are used to represent inputs from the line-detecting sensors on the left (`SensorVisionLeft`) and on the right (`sensorVisionRight`). Two further *monitored* variables, `sensorRotationSpeedLeft` and `sensorRotationSpeedRight`, are used to represent input from the sensors used to count revolutions of the left and right wheel respectively.

Two *controlled* variables, `actuatorWheelLeft` and `actuatorWheelRight` are used to produce signals to the motor for the left and right wheels respectively.

The contract also includes the five shared design parameters described for all groups in Section 9.1.

9.6.3 Discrete-event

The DE model for Group 5's model begins with the `System` class, which creates two instances of `SensorVision` to represent sensors for the line-sensing task, and two instances of `SensorRotation` to represent sensors for the wheel encoders. It also instantiates two actuators, using the `ActuatorWheel` class, to power the wheels, and a single instance of `Controller` to provide control functionality. Finally it deploys the model to a single CPU.

The `ActuatorWheel` class simply handles passing data to the actuators and the `SensorRotation` class simply handles data coming from the sensors employed as encoders. The `SensorVision` class, however, which represents the line-detecting sensors, includes some extra functionality that analyses the colour values of the area of floor currently visible and makes a decision as to whether it represents line, background or some undertermined value. Group 5 have created some abstract classes for the actuators and sensors (`AbstractAcutatorReal` and `AbstractSensorReal` respectively) but these are not currently employed by the model.

The `Controller` also creates an instance of `ComputeSpeedAlg2`. This class calculates the optimal speed for each wheel, given some information about the current line-detecting sensors

and encoders (several alternative versions of this class were created but are not in current use in the model). `ComputeSpeedAlg2` collects the currently visible colours from the line-detecting sensors and makes a decision. Initially the class hunts for the known pattern. Once the line has been detected, it adopts slightly different logic. If both sensors see background, then it's assumed that the end of the line has been reached. If the right-hand sensor detects the line and the left-hand does not then the speed is calculated to turn the robot to the right. Likewise, if the left-hand sensor detects the line and the right-hand does not, then the robot turns to the left.

When turning, Group 5's robot produces a positive voltage for one wheel and a negative for the other, in order to optimise the amount of rotation in the shortest distance.

9.6.4 Continuous-time

As with all the other groups, the CT model includes blocks to provide an implementation of robot physics (`RobotPhysics`) and to produce data for the 3D simulation (`DataFor3D`). There is a `Controller` block which acts as an interface between the DE and CT models, accepting data from the DE model for the actuators and producing information from the sensors for the DE model to process.

There are separate blocks representing the two actuators/encoders (`scalingLeft` and `ScalingRight`). These blocks incorporate some logic to translate data from the encoders into number of revolutions the wheel has turned. There are also two (unfinished) blocks representing the line-detecting sensors (`DummyLineSensorLeft` and `DummyLineSensorRight`), which would produce data representing the colour of the currently visible area on the ground. Data from the `Controller` for the encoders passes first through the `RobotPhysics` block before being passed to the actuators/encoders. Output from the sensors and the `RobotPhysics` block is passed to the `DataFor3D` block for processing in a 3D simulation.

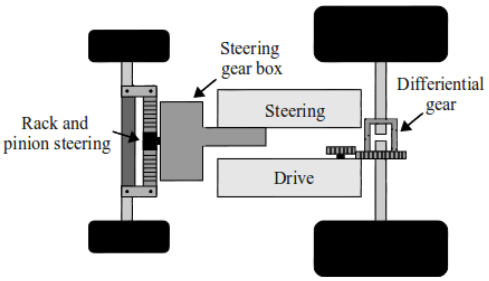
Chapter 10

Tractor Simple

10.1 Case Description

The ”Tractor Simple” demonstrates a model of a Lego®Mindstorms®NXT tractor (micro-tractor), used to demonstrate agricultural route planning and execution. A pre-planned route for the micro-tractor is supplied in advance (see figure 10.1(a)). The onboard system then aims to adjust the current position so it gets as close as possible to the pre-planned route. The ability to automatically correct the position helps deal with physical conditions, which may affect the vehicle's movements in unpredictable ways.

(a) Micro-tractor with an example route-plan



(b) Sketch of the micro-tractors steering and drive components.

Two DC-motors on the micro-tractor are used to steer the front-wheels and drive the back-wheels (see figure 10.1(b)). The current route segment (straight line, turn/circle) in the pre-planned route is followed until the controller determines a new segment must be loaded. When the micro-tractor has fished the pre-planned route, it stops and ends the execution phase.

10.2 External Links

The purpose of this model was to model the general behaviour of the micro-tractor, when running a pre-planned route and compared successfully against the real system in [CBEL12]. Using the

Crescendo model for tuning optimized control parameters for the route-following algorithm was determined.

10.3 Contract

The primary purpose of this model is to simulate the behavior of the micro-tractor, when following a route-plan. Different P-control parameter values can be set to determine the influence on the system output. P-control factor is set using shared variable of type real found in the contract as: `P_k`. The voltage output to the motors can change to simulate different input voltage level to the motors. Recommended voltage level is between **7.4-9.0** V, which is normal battery operation range. To set the output voltage in the contract, the parameter `Voltage_Power` of type real should be used.

The contract contains three monitored variables of type real used to read the rotational-angle of the motor-encoders and the IMU angle of the tractor: `ImuOrientation`, `wheelRotations`, `steerRotations`. To control the motors for the drive and steering system the DE controller uses the following two control signals of type real in the contract: `drivingControl`, `steeringControl`. Finally the contract contains the route number, used in the system-case for route following.

10.4 Discrete-event

The DE model consist of the `Controller` which is the main thread of control, as well as the task scheduler of the route-follower. The route follower `Route` is used to determine the current position and delegate the individual route segments to a low-level controller. Inputs from the motor-encoders and IMU are used to determine the current position of the tractor using kinematic estimates. Motor output-value rage is equal to the real range found on the actual micro-tractor. The low-level controller is a P-controller used to keep the micro-tractor on the wanted route and follow a specific lines.

10.5 Continuous-time

The CT model includes a controller block (`Controller`) which handles the interface with the DE model. There are blocks to represent the motors, the gearing and the wheels for the back of the tractor, and a separate set of blocks to represent motors, gearing and wheels for the front of the tractor. There are separate encoders for the front and back of the tractor.

The controller produces voltage signals to control both steering and drive motors on the micro-tractor. The motor encoders measure the current speed of the tractor, and the IMU measures the orientation and feeds this information back to the controller. The input to the motor encoders is the motor speed (produced by either the `Backend_motor` or the `Frontend_motor`) with some added noise to model a more accurate system and the output is a count indicating the number of

rotated degrees. IMU input data is based on the estimated kinematic position of the micro-tractor modeled in 20-sim and the output is the current angle of the tractor.

20-sim is used to 2D-plot the actual X,Y path the micro-tractor follows compared to the pre-planned route. This can be used to visually evaluate the effect of the current parameter setup. If the control parameters are set at random, scenarios will be encountered where the micro-tractor steers off the pre-planned route.

10.6 Usage

To load another route plan change the path to the *.csv file used to store the plan. Three example route-plans: (`route1.csv`,`route2.csv`,`route3.csv`) are provided by default to provide the user with an overview of the capabilities. Simulation time should be set high for the tool to be able to run the entire route-plan. `route1.csv` requires a simulation length of between 100-120 sec, depended on the voltage input. The format of the route-plane file is the following:

Type	Orientation	distance
1(Line)	90 (degrees)	2 (meter)
2 (Cycle)	-180 (degrees)	X (empty)

Table 10.1: Format of the route-plan csv files.

Any route length can be chosen, but be aware that longer routes require longer simulation time. To get the tractor to steer an unpredictable route, provide it with a line route-segment, with an orientation opposite its own current orientation.

Chapter 11

ChessWay Simple

11.1 Case Description

The ChessWay is a two-wheeled, self-balancing scooter which utilises a gyroscope to maintain a stable upright position. The scooter consists of a platform on which the rider stands, two parallel wheels and a handlebar. A model of the scooter can be seen in Fig 11.1.

Figure 11.1: Image of the ChessWay personal scooter

The ChessWaySimple model is a simplified and abstracted model of the scooter. It features a sensor which can be used to calculate current forward velocity and an acceleration sensor which can be used to determine the current angle. Each wheel has its own motor, although there is a single controller for both motors, so the scooter travels in a straight line only (see the ChessWay **Crescendo** model in Chapter 13 for a slightly more complex model that permits separate inputs to motors). Users simply lean forwards to increase forward motion. This is a trivial and simple

version of the ChessWay model; for more advanced versions of the same vehicle we recommend studying Chapters 12 and 13.

11.2 Contract

The contract contains four shared state variables of type real. Two monitored variables are responsible for reading in signals from the sensors used for calculating velocity (`v_in`) and acceleration (`a_in`). Two controlled variables are responsible for producing a signal to the motors to alter forward velocity (`v_out`) or to make adjustments to the current angle (`a_out`). Both of the controlled variables produce a signal to power the motors.

11.3 Discrete-event

The DE model includes a `Controller` class which manages the main thread of control. The Controller instantiates two `IActuatorReal` abstract classes, to represent the actuator signal for adjusting vertical position (`acc_out`) and the actuator signal for adjusting forward velocity (`vel_out`). It also instantiates two `ISensorReal` abstract classes to represent input signals from the velocity sensor (`vel_in`) and the acceleration sensor (`acc_in`). At run-time, the `Actuator` class provides a concrete implementation of `IActuatorReal` and the `Sensor` class provides an implementation of `ISensorReal`.

The ChessWay scooter implements a closed feedback loop to produce the self-balancing functionality. The current angle at any one time is calculated (based on the input from the `acc_in` class that represents the gyroscope's readings), and then based on this, the scooter's motor is fed a signal to make corrections and ensure the scooter stays upright.

The `System` class deploys the `Controller` onto a single CPU.

11.4 Continuous-time

On the top level of the hierarchy, the model consists of three main blocks: the controller (the `cosim` part of the model), the plant and a block used for `IO` which resides in between the `cosim` and the plant. The controller handles the passing of variables to/from the DE model. The plant block contains three elements, to represent the axis, the person frame (the unit consisting of the platform and handlebar) and the wheels. And finally the `IO` block primarily contains D-A and A-D conversion functionality.

11.5 Usage

The ChessWay Simple model is designed to be a simple introduction to a more complex version of the same scooter (see Chapter 13). Running a simulation of the Simple model demonstrates how

the scooter responds to small changes in velocity by altering signals to the actuators so that it can maintain an upright position by way of many small corrections.

Chapter 12

ChessWay SL

12.1 Case Description

The ChessWaySL model is an extension of the previous model (see Chapter 11), and acts as a bridge between the very simple model presented there and the much more complex model of the same scooter which is presented in the following chapter (Chapter 13). The model in this chapter includes the same basic features of a self-balancing scooter described in Chapter 11. However, in this model we are beginning the process of adding some extra safety features, additional controls and some fault tolerant functionality. As with the previous model, the ChessWaySL scooter consists of a platform on which the rider stands, with two parallel wheels and a handlebar. There is an ignition switch which the user can use to power the system down explicitly. There is also a removable safety key device, which is always inserted into a slot in the handlebar unit while the scooter is in use. At the same time the safety key will be attached to the user's wrist by a cord, so that if the user is unfortunate enough to fall off the scooter, the safety key will be pulled away from its slot in the handlebar unit. The removal of the safety key can therefore assumed to mean that the scooter is riderless, and the vehicle must immediately come to a safe, stationary position.

A distributed controller architecture was chosen for the more advanced ChessWay, whereby each wheel has its own controller. Each motor controller is guarded by a so-called safety monitor, which has the task to intervene and put the system in a fail-safe state in case any fault is detected. The failsafe state condition for the ChessWay is the situation where neither motor is actuated. Furthermore, the system should be able to recuperate from such an intervention and return to normal operating mode, in case the root cause of the fault has been removed. Figure 12.1 shows the states and transitions we require for the controller, and Figure 12.2 shows the events and transitions needed for the plant.

In order to develop the simple version of the model (Chapter 11) into a model that incorporates all of the above features, some key design questions must be answered:

- What constitutes the system?
- What constitutes the environment?

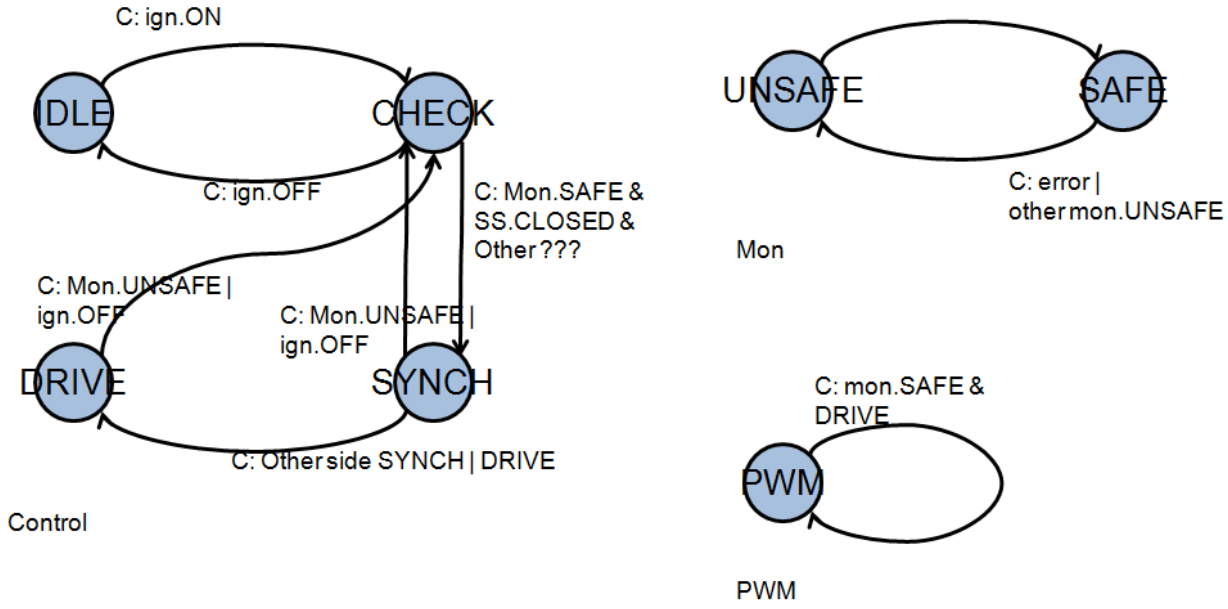


Figure 12.1: States and transistions needed for the ChessWay controller model

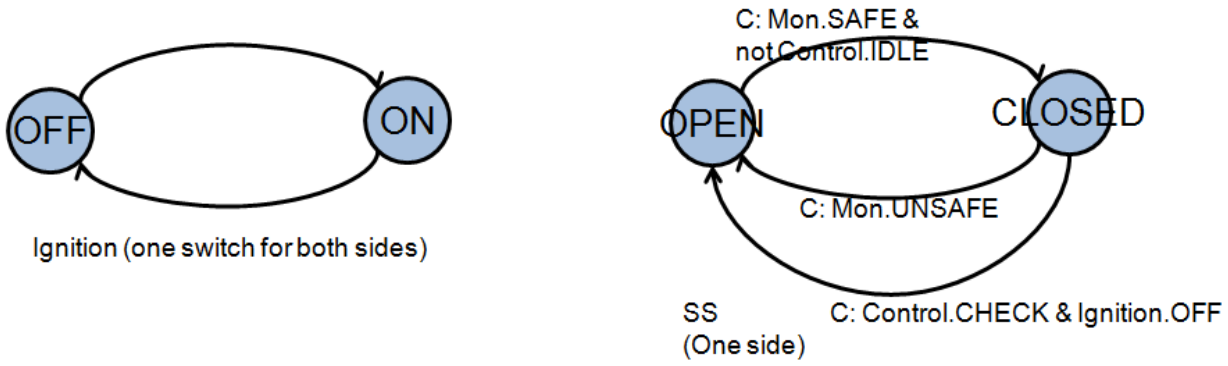


Figure 12.2: States and transistions needed for the ChessWay plant model

- What should constitute the DE side of the simulator?
- What should constitute software in the controller?
- What is the minimum number of sensors we need and what is necessary for the plant?

In order to answer these questions, engineers initially adopt a DE-first approach, designing the DE side of the model only and assuming that there is a CT model available which is always correct. This allows them to concentrate on the discrete event changes that will be encountered during start-up and shut down, and during fault-handling and recovery. The added value of the intermediate model is that is focuses on state changes only: it shows which sensors are deeded to

be able to detect certain faults; it changes state from detecting a fault to a safe sate and tries to recover to an operational state; and it shows the interaction between user and device. We present the intermediate, DE-only model in this chapter, and we present a more complete version of the model with a fully-implemented CT model in Chapter 13.

12.2 Defining Data Types

The first step is to define a set of data types which will act as a convenient vocabulary in our model. Simple types (omitted here) for describing properties from the physical domain start with a capital **R** and for the sensed properties we use a prefix **S**. The actuator types are preceded by **A**. The safety monitor observes the system health and manages variables of type `SafetyMode`. The controller operates the system and manages variables of type `OperatingMode`.

```
types
  Plant ::
    poleAngle : RPoleAngle
    angleVel : RAngleVel
    safetyKey : RSafetyKey
    powerSwitch : RPowerSwitch
    pwmL : RPwm
    pwmR : RPwm
    safetySwitchL : RSafetySwitch
    safetySwitchR : RSafetySwitch;

  Sensors ::
    ignition : SPowerSwitch
    poleAngle : SPoleAngle
    angleVel : SAngleVel
    safetyKey : SSafetyKey;

  Actuators ::
    pwm : APwm
    safetySwitch : ASafetySwitch;

  Controller ::
    mode : OperatingMode;

  Monitor ::
    mode : SafetyMode;
```

The structured types show that the `Plant` type holds the actual physical properties, while `Sensors` contains the observed values. The real and observed values may differ due to timing, as the real physical property changes in real-time while the sensed values only changes when triggered to do so. And furthermore, the values may differ due to sensor faults. A similar argument holds for the actuators.

12.3 Defining State

We can now define the state of our model (some states are omitted here).

```
state Sigma of
  -- real world
  plant  : Plant

  -- modelled world
  sens   : Sensors
  actL   : Actuators
  actR   : Actuators
  ctrlL  : Controller
  ctrlR  : Controller
  monL   : Monitor
  monR   : Monitor

  -- keep track of the error state
  errors : map ErrorTypes to bool
```

Note that the variable `plant` represents the physical system properties. The `sens` variable is used to hold the sampled sensor value at some specific point in time, while the variables `actL` and `actR` hold the most recent control signal which is send to the motors that drive a wheel each. Finally there is a pair of *controllers* and a pair of *monitors*. The controller resembles the digital controller and the monitor represents the safety monitor for that controller. The `plant` and `sens` variables occur once, while all other system elements occur twice, for the left (**L**) and the right (**R**) wheel respectively. The controller and monitor variables are containers for the controller and monitor state machines. They will become the core processes in co-simulation and in the implementation.

12.4 Defining Auxiliary Functions

We can now define some auxiliary functions and operations that can be used as encodings of transitions in a state machine. Using some simple functions (omitted here) to read sensors, whereby we treat the user interface button (on/off switch) as an ordinary sensor, we can now define the transition engine that is at the heart of the controller state machine.

```
  -- process to migrate from state to state = statemachine
  switchState : Controller * Monitor * Actuators *
    Controller * Monitor ==> OperatingMode
  switchState(c, m, act, cOther, mOther) == (
      -- check monitor first
      -- check ignition
      -- change state
      cases c.mode :
```

```
      <IDLE> ->  if sens.ignition = <OFF>
                 then return <IDLE>
                 else return <CHECK>,
      <CHECK> -> if sens.ignition = <OFF>
                 then return <IDLE>
                 else if m.mode = <SAFE> and
                         act.safetySwitch = <CLOSED>
                      then return <SYNC>
                      else return <CHECK>,
      <SYNC> -> ( if m.mode = <SAFE> and
                     mOther.mode = <SAFE> and
                     ( cOther.mode = <SYNC> or
                       cOther.mode = <DRIVE> )
                  then return <DRIVE>;
                  if m.mode <> <SAFE> or
                     mOther.mode <> <SAFE>
                  then return <CHECK>;
                  return <SYNC> ),
      <DRIVE> -> if m.mode <> <SAFE> or
                    sens.ignition = <OFF>
                 then return <CHECK>
                 else return <DRIVE>
    end;
    -- default return value
    return <CHECK>;
  );
```

The core process of the controller. The On/Off switch makes the state switch between IDLE and the other states. The CHECK state is the landing place after switching on and after detection of an error. The SYNC state is introduced to synchronize between the left and right controller. The controller being in a safe state, a check is made before powering the motor that the other side is in the same state as well. If so then both will move to the DRIVE state and power the two motors. In case of an error, or desire to switch off, the mode will change to CHECK again.

12.5 Usage

Finally, we can create an operation that simulates the behavior of the state machine.

```
-- simulator
simulator : () ==> ()
simulator() == (
  dcl tick : int := 0;

  while tick < 25 do (
    IO`print(tick);

    -- script
```

```
      if tick = 2
        then switchOn();
      if tick = 4
        then pullSafetyKey();
      if tick = 7
        then restoreSafetyKey();
      if tick = 16
        then pullSafetyKey();
      if tick = 19
        then restoreSafetyKey();
      if tick = 23
        then switchOff();

      -- read sensor values
      sens := sense();

      -- sense & monitor & activate
      if ctrlL.mode <> <IDLE>
        then (
          monL.mode := switchMonitorState(ctrlL);
          actL.safetySwitch :=
            switchSafetySwitch(monL.mode, ctrlL.mode);
          actL.pwm := pwmOut(ctrlL.mode, monL.mode, sens);
          plant.pwmL := setPwm(actL);
          plant.safetySwitchL := setSafetySwitch(actL);
      );
      if ctrlR.mode <> <IDLE>
      then (
          monR.mode := switchMonitorState(ctrlR);
          actR.safetySwitch :=
            switchSafetySwitch(monR.mode, ctrlR.mode);
          actR.pwm := pwmOut(ctrlR.mode, monR.mode, sens);
          plant.pwmR := setPwm(actR);
          plant.safetySwitchR := setSafetySwitch(actR)
        );

      -- control
      ctrlL.mode := switchState(ctrlL, monL, actL, ctrlR, monR);
      ctrlR.mode := switchState(ctrlR, monR, actR, ctrlL, monL);

      tick := tick + 1;
    );
  )
end Chessway
```

The simulator process demonstrates that changes in the environment (user input) make the system pass the states in a desired manner. Switch On and Off, and pulling the safety key at specific moments to trigger errors at all times in the state machine. Code coverage analysis showed that initial tests missed some of the possible execution paths. By adding more tests manually, more

confidence is gained in the design and analysis. The experiments show that this model is at the right level of being able to add faults and to analyse fault handling.

Chapter 13

ChessWay Crescendo

13.1 Case Description

The ChessWay **Crescendo** model is an extension of the previous two models (see Chapters 11 and 12). Like the model presented in Chapter 12, the vehicle described by the ChessWay**Crescendo** model is a self-balancing scooter with two parallel wheels and a handlebar, an ignition switch and a removable safety key.

The scooter employs sensors to indicate whether the safety key has been removed from the handlebar unit, or whether the user has switched off the ignition. There are some differences in how the scooter should respond when ignition is switched off, or when the safety key is removed. If the user turns off the scooter, then the model should first check the current state of the vehicle, and if it is currently in active motion it should bring it to a safe, stationary position before deactivating. In contrast, if the safety key is removed from the handlebar unit, then we assume that the user has fallen from the scooter and so it should be immediately brought to rest with no need to check current status.

Unlike the ChessWaySimple model, there are separate controllers for each of the wheels. The controllers for the motors are connected wirelessly, so the model must cope with occasional lost data packets. As before, there is a sensor which can be used to calculate current forward velocity of the scooter, and a gyroscope (acceleration) which can be used to determine the current vertical angle.

13.2 Contract

The contract contains six shared state variables of type real. Two monitored variables are responsible for reading in detected error in velocity (`rmsVelocityError`) and the current angle of the handlebar mast (`poleAngle`). Four more controlled variables are responsible for producing signals to the motors to alter velocity for the left (`pwmSettingL`) and right (`pwmSettingR`) wheels, and for powering the safety key override on the left (`safetySwitchL`) and on the right (`safetySwitchR`).

The contract also contains 6 display variables, all of type controlled real. The display variables are shared between the DE and CT models and are available for 3D modelling, if needed.

Finally, there are two shared design parameters, `pidGain` and `sdpSlip`, the values of which can be set or altered to simulate different errors.

13.3 Discrete-event

We recommend also reading Chapter 12, which describes the process of implementing this model in more detail. The DE model includes a number of classes to designed to represent various sensors, including one each for `Gyro`, `Ignition`, `SafetyKey`, `VelocitySensor` and one to detect `VelocityError` (it's assumed that the hardware is capable of reporting self-detected errors). There are also two classes representing actuators: `Pid` handles the standard actuators powering the wheels. And `SafetySwitch` handles an override of the `Pid` class, if the safety key has been withdrawn.

The DE model starts with a `System` class, which creates single instances of each of the sensors for monitoring: the gyroscope; the safety key; the ignition switch; the velocity; and detected velocity errors. `System` then creates two instances of a `Monitor` class (one for each wheel/-motor) to monitor these sensors and determine the correct current state for the scooter. `System` also creates two instances of the `RTControl` class, which act as controllers (one for each motor/wheel). The `RTControl` instances each have their own copy of a `Pid` class (representing the actuator for activating the motor) and a `SafetySwitch` class (for overriding the actuator).

Finally, `System` also creates a single instance of a class that represents the wireless data connection. An abstract class `Ether` is provided for this purpose along with several concrete implementations, each of which behaves slightly differently. Currently the model uses the concrete implementation `LossyEther`, which 'drops' randomly selected data packets at runtime.

13.4 Continuous-time

The CT model is presented in more detail than other examples, and several possible 20-sim models are provided. There are some small variations in detail in each of the 20-sim models, but for each one the blocks are gathered into collections of actuators (which receive signals for the motors driving the wheels and for the safety switch circuit) and sensors (which produce outputs relating to the current angle, the velocity and detected error). To maintain a high level of dependability, we assume that the safety key functionality is deployed onto a separate CPU to the main plant. Both the main plant and the safety monitor receive copies of the sensor input; should the safety monitor detect anything amiss it overrides the main plant's outputs with its own signal to deactivate the scooter.

A single block (`sbsInterface`) handles imports from, and exports to, the DE model; this block produces outputs for the actuators and receives inputs from the sensors. In each model a collection of `Cycloid` blocks is used to calculate whether the gyroscope is accurately detecting

vertical. A single block (either `Chessway` or `SBS`) represents a detailed submodel of the plant, and interacts with the actuators, sensors and error-detecting functionality.

The actuators for each wheel communicate via a wireless connection in this model, allowing some fault tolerance behaviour to be tested as the model must cope with packets occasionally lost in transmission. In the CT model each of the receivers for the motors is linked to a fault-generating block that loses occasional packets.

13.5 Usage

This is a complex model, and many scenarios are also possible. The **Crescendo** co-model provides alternative implementations of the `Ether` class, for example, which exhibit different behaviours and may be tried separately to test fault tolerance. The difference between activating the safety key and deactivating the ignition can be seen by changing the script above to shutdown the ignition instead of removing the safety key.

13.5.1 Scenarios

A simple scenario is provided for the ChessWay **Crescendo** model, demonstrating some aspects of the scooter's safety.

```
when time = 0.0 do (de real safetyKeyEnabled := 1.0; );

when time = 0.2 do (de real safetyKeyEnabled := 0.0;);

when time = 0.3 do (de real safetyKeyEnabled := 1.0;);

when de real poleAngle < 0.0-0.2
    or de real poleAngle > 0.2
    do ( print "QUIT ... "; quit; );
```

In this scenario, the safety key is withdrawn from the handlebar unit (`when time = 0.2`) and then quickly replaced (`when time = 0.3`). The scooter responds to the removal of the safety key by beginning to shut down, but the replacement of the key results in abandoning the shut down and the scooter attempting to recover its stable vertical position. The timings provided in this scenario should demonstrate that the scooter is capable of recovering and returning to vertical safely. The scenario can be altered and executed with different timings, allowing us to determine the maximum length of time that may elapse between withdrawing the safety key and replacing it and still have the scooter recover its upright position.

Another possibility is to try out the gyroscope fault detection functionality. The script below simulates the detection of an error in the gyroscope.

```
when time = 0.0 do (de real safetyKeyEnabled := 1.0; );
```

```
when time = 0.1 do (de int gyroError := 1;);

when time = 0.13 do (de int gyroError := 0;);
```

Bibliography

[BFG+12] Jan F. Broenink, John Fitzgerald, Carl Gamble, Claire Ingram, Angelika Mader, Jelena Marincic, Yunyun Ni, Ken Pierce, and Xiaochen Zhang. D2.3 — Methodological Guidelines 3. Technical report, The DESTECS Project (CNECT-ICT-248134), available from `http://www.destecs.org/`, December 2012.

[BHLM94] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous System. In *Int. Journal of Computer Simulation*, 1994.

[CBEL12] Martin Peter Christiansen, Kim Bjerge, Gareth Edwards, and Peter Gorm Larsen. Towards a methodology for modelling and validation of an agricultural vehicle's dynamics and control. In Sergio Junco, editor, *The 6th International Conference on Integrated Modeling and Analysis in Applied Control and Automation*, IMAACA, pages 112–119, September 2012.

[DGG+99] J. Davis, R. Galicia, M. Goel, C. Hylands, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong. Ptolemy-II: Heterogeneous concurrent modeling and design in Java. Technical Memorandum UCB/ERL No. M99/40, University of California at Berkeley, July 1999.

[DLSV11] Patricia Derler, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. Modeling cyber-physical systems. *Proceedings of the IEEE special issue on CPS*, December 2011.

[EJ95a] R.Johnson E.Gamma, R.Helm and J.Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, 1995.

[EJ95b] R.Johnson E.Gamma, R.Helm and J.Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, 1995.

[EJL+03] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming Heterogeneity – the Ptolemy Approach. *Proc. of the IEEE*, 91(1):127–144, January 2003.

[JGSIS07] Juan F. Jimenez, Jose M. Giron-Sierra, C. Insaurralde, and M. Seminario. A simulation of aircraft fuel management system. *Simulation Modelling Practice and Theory*, 15(5):544 – 564, 2007.

[PFG11] Ken Pierce, John Fitzgerald, and Carl Gamble. Modelling faults and fault tolerance mechanisms in a paper pinch co- model. In *Proceedings of the ERCIM/EWICS/Cyber-physical Systems Workshop at SafeComp 2011, Naples, Italy (to appear)*. ERCIM, September 2011.

[WPD12] Sune Wolff, Ken Pierce, and Patricia Derler. Multi-domain modelling in destecs and ptolemy — a tool comparison. *Submitted to: ACM Transactions on Modeling and Computer Simulation, TOMACS*, 2012.