Crescendo Technical Report Series
No. TR-002

January 2014

Crescendo Examples Compendium

by

Claire Ingram, Ken Pierce, Carl Gamble,

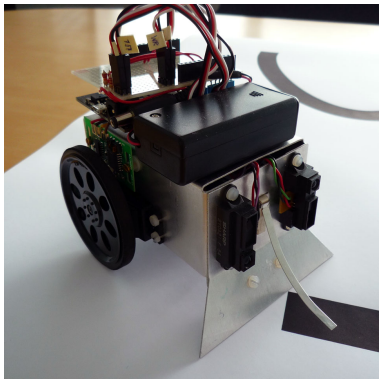
Sune Wolff, Martin Peter Christensen and Peter Gorm Larsen
Aarhus University, Department of Engineering
Finlandsgade 22, DK-8200 Aarhus N, Denmark

Chapter 8

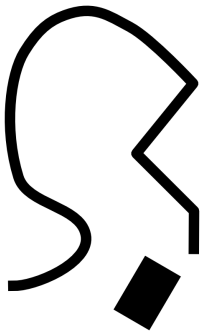
Line Following Robot

8.1 Case Description

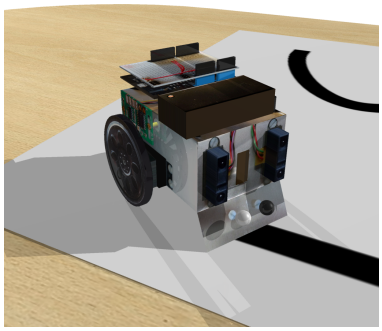
The line following robot is a co-model that demonstrates how design space exploration is supported in **Crescendo**. The co-model simulates a robot that can follow a line painted on the ground. It is assumed that the colour of the line is known and that it is differentiated from the background colour (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.

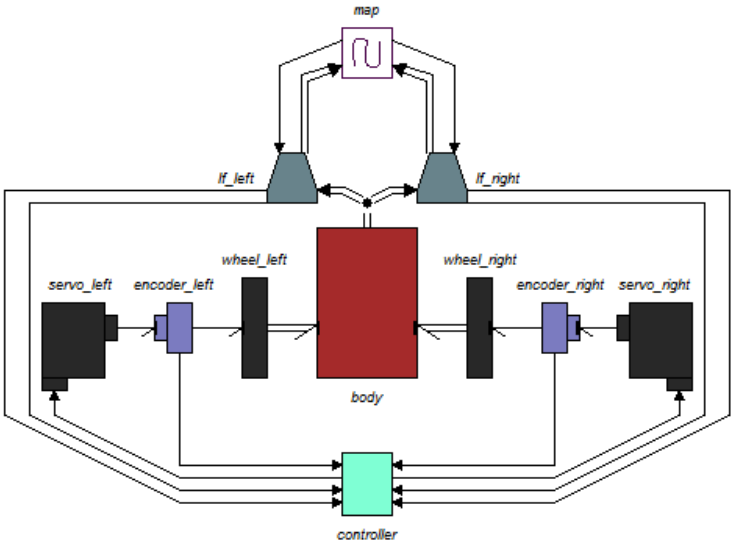


Figure 8.2: Continuous-time model of the R2-G2P

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

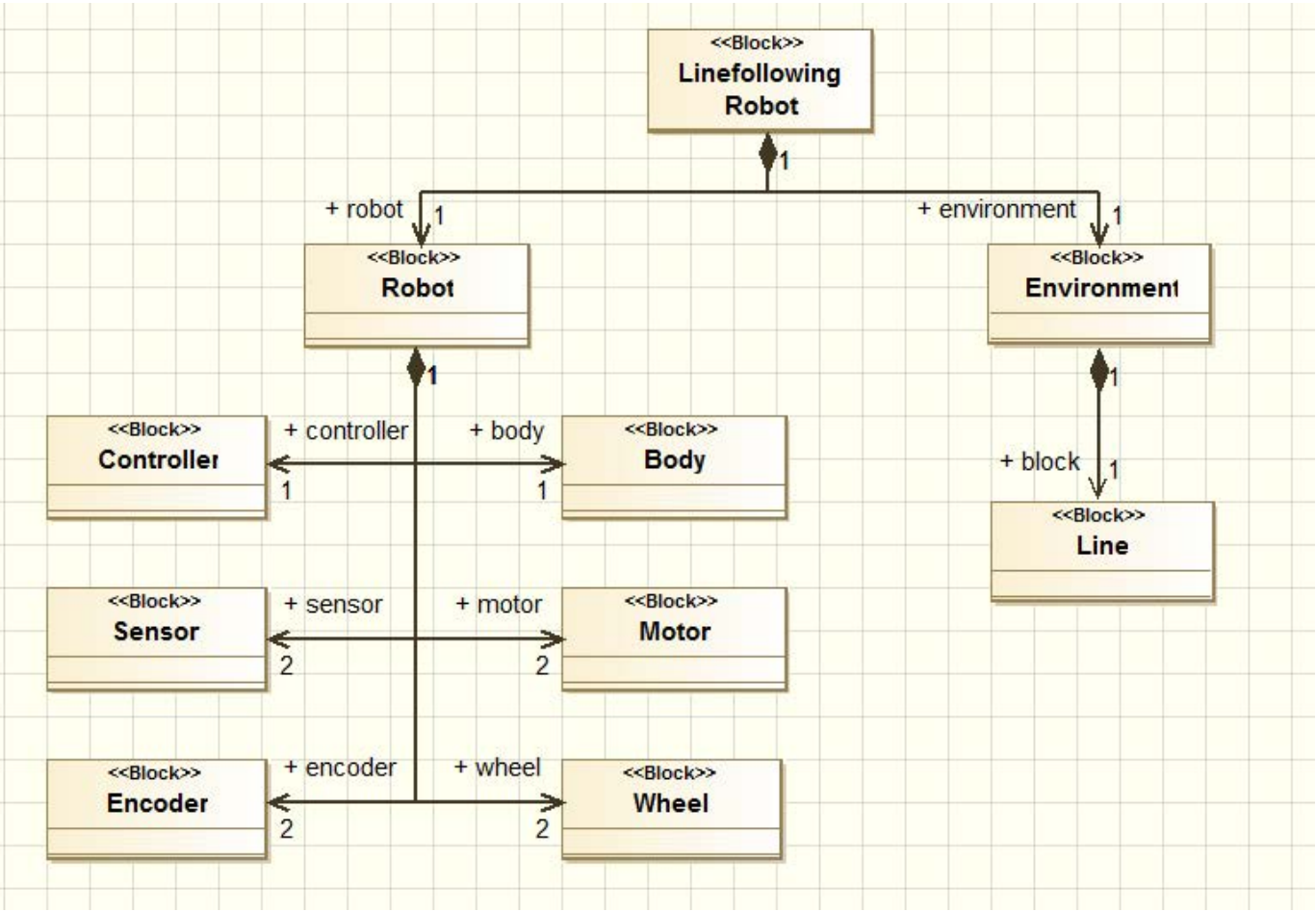
The line-following robot model is a good opportunity to see how **Crescendo** can support design space exploration. A good starting point is to run the co-model (using the `LineFollowingRobot` launch configuration), examining the speed and accuracy (distance) with which it can complete various lines. 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.



SysML Block Definition diagram of the line following robot in Modelio

Appendix A

Glossary

As might be expected in interdisciplinary projects, terms and concepts that are well known in one discipline may be unknown or understood quite differently in another. This appendix therefore contains common descriptions of core concepts that are used with the Crescendo technology.

abstract class (in object oriented programming) a class where one or more methods are defined abstractly using the text `is subclass responsibility` as their body.

actuator a component that produces a physical output in response to a signal [IEE00].

aggregate (in object oriented programming) the act of bringing together several objects into a single whole.

automated co-model analysis tool support for the selection of a single design from a set of design alternatives (including definition of scenarios, execution of co-simulations, and visualisation and analysis of co-simulation results).

automated co-model execution as automated co-model analysis except that is does not perform any analysis of the test results produced by the simulations

bond (in bond graphs) a directed point-to-point connection between power ports on submodels. Represents the sharing of both *flow* and *effort* by those ports.

bond graph a domain independent idealised physical model based on the representing energy and its exchange between submodels.

causality (in bond graphs) dictates which variable of a power port is the input (cause) for submodel's equations and which is the output (effect).

class (in object oriented programming) the definition of the data field and methods an object of that class will contain.

code generation the process of implementing a system controller by automatically translating a model into a representation (in some programming language) which can then be executed on the real hardware of the system.

co-model a model comprising two constituent models (a DE submodel and a CT submodel) and a contract describing the communication between them.

consistency a co-model is consistent if the constituent models are both syntactically and semantically consistent.

constituent model one of the two submodels in a co-model.

continuous-time simulation a form of simulation where "the state of the system changes continuously through time" [Rob04, p. 15].

contract a description of the communication between the constituent models of a co-model, given in terms of shared design parameters, shared variables, and common events.

controlled variable a variable that a controller changes in order to perform control actions.

controller the part of the system that controls the plant.

controller architecture the allocation of software processes to CPUs and the configuration of those CPUs over a communications infrastructure.

co-sim launch the type of debug configuration used in the Crescendo tool to define and launch a single scenario.

co-simulation baseline the set of elements (co-model, scenario, test results etc.) required to reproduce a specific co-simulation.

co-simulation engine a program that supervises a co-simulation.

co-simulation the simulation of a co-model.

cost function a function which calculates the "cost" of a design.

debug config (Eclipse term) the place in Eclipse where a simulation scenario is defined.

design alternatives where two or more co-models represent different possible solutions to the same problem.

design parameter a property of a model that affects its behaviour, but which remains constant during a given simulation.

design space exploration the (iterative) process of constructing co-models, performing co-simulations and evaluating the results in order to select co-models for the next iteration.

design step a co-model which is considered to be a significant evolution of a previous co-model.

discrete-event simulation a form of simulation where "only the points in time at which the state of the system changes are represented" [Rob04, p. 15].

disturbance a stimulus that tends to deflect the plant from desired behaviour.

edges (in bond graphs) see *bond*.

effort (in bond graphs) one of the variables exposed by a power port. Represents physical concepts such as electrical voltage, mechanical force or hydraulic pressure.

environment everything that is outside of a given system.

error part of the system state that may lead to a failure [ALRL04].

event an action that is initiated in one constituent model of a co-model, which leads to an action in the other constituent model.

executable model a model that can be simulated.

failure a system's delivered service deviates from specification [ALRL04].

fault injection the act of triggering faulty behaviour during simulation.

fault modelling the act of extending a model to encompass faulty behaviours.

fault the adjudged or hypothesized cause of an error [ALRL04].

fault behaviour a model of a component's behaviour when a fault has been triggered and emerges as a failure to adhere to the component's specification.

fault-like phenomena any behaviour that can be modelled like a fault (e.g. disturbance).

flow (in bond graphs) one of the variables exposed by a power port. Represents physical concepts such as electrical current, mechanical velocity, fluid flow.

ideal behaviour a model of a component that does not account for disturbances.

inheritance (in object oriented programming) the mechanism by which a subclass contains all public and protected data fields and methods of its superclass.

input a signal provided to a model.

interface (in object oriented programming) a class which defines the signatures of but no bodies for any of its methods. Should not be instantiated.

junction (in bond graphs) a point in a bond graph where the sum of flow (1-junction) or effort (0-junction) of all bonds to that point is zero.

log data written to a file during a simulation.

metadata information that is associated with, and gives information about, a piece of data.

model base the collection of artefacts gathered during a development (including various models and co-models; scenarios and test results; and documentation).

model management the activity of organizing co-models within a model base.

model structuring the activity of organizing elements within a model.

model synthesis see **code generation**.

model a more or less abstract representation of a system or component of interest.

modelling the activity of creating models.

modularisation construction of self-contained units (modules) that can be combined to form larger models.

monitored variable a variable that a controller observes in order to inform control actions.

object (in object oriented programming) an instantiation of a class, contains data fields and methods.

objective function see **cost function**.

ontology a structure that defines the relationships between concepts.

operation (in object oriented programming) defines an operation that an object may perform on some data. Operations may be *private, public* or *protected*.

output the states of a model as observed during (and after) simulation.

non-normative behaviour behaviour that is judged to deviate from specification.

physical concept (in bond graphs) a class of component or phenomena that could exist or be observed in the real world, e.g. an electrical resistor or mechanical friction.

plant the part of the system which is to be controlled [IEE00].

power port (in bond graphs) the port type connected in a bond graph. Contains two variables, *effort* and *flow*. A power port exchanges energy with its connected models.

private (in object oriented programming, VDM) the method or data field may only be accessed from within the containing class.

protected (in object oriented programming, VDM) the method or data field may only be accessed by its containing class or any of its subclasses.

public (in object oriented programming, VDM) the method or data field may be accessed by any other class.

ranking function a function that assigns a value to a design based on its ability to meet requirements defined by the engineer.

realistic behaviour a model of a component which includes disturbances defined by the tolerances associated with that component.

repository a shared store of data or files.

response a change in the state of a system as a consequence of a stimulus.

revision control the activity of managing changes (revisions) to computer data or files.

scenario test of a co-model.

signal domain where models share a single value or array at each port and where those ports are uni-directional, unlike bond graphs where the ports are bi-directional.

sensor a component whose input is a physical phenomenon and whose output is a quantitative measure of the phenomenon.

shared design parameter a design parameter that appears in both constituent models of a co-model.

shared variable a variable that appears in and can be accessed from both constituent models of a co-model.

simulation symbolic execution of a model.

semantically consistent the state when the constituent models of a co-model agree on the semantics of the varaibles, parameters and events they share. The nature of these semantics is not yet described.

static analysis a method for checking some property of a model without executing that model.

state event an event triggered by a change within a model.

stimulus a phenomenon that effects a change in the state of a system.

subclass (in object oriented programming) a class that is defined as extending another class. The other class becomes its superclass. The subclass inherits all non private data fields and methods.

submodel a distinct part of a larger model.

superclass (in object oriented programming) the class from which a subclass is defined.

syntactically consistent the state when the constituent models of a co-model agree on the identities and data types of all shared variables, parameters and events.

system boundary the common frontier between a system and its environment.

system under test (SUT) the part of a model that represents the system we wish to build, as opposed to parts of the model which are not part of this system.

system an entity that interacts with other entities, including hardware, software, humans and the physical world [ALRL04].

tag to associate metadata with a piece of data.

test result a record of the output from a simulation of a model (see also **log**).

time event an expected event that occurs at a predetermined time.

variable part of a model that may change during a given simulation.

vertices (in bond graphs) the joining points of bonds. May be manifested as either a *junction* or a submodel.