The University of Melbourne

SWEN90010: High Integrity Software Engineering

Assignment 1

Due Date: 11:59pm, Wednesday 22 March, 2017

1 Introduction

This assignment is worth 10% of your total mark.

The aim of this assignment is to provide you with an opportunity to implement a simplified safety- and security-critical system in Ada. Doing so will help to explore the properties of Ada, and how they relate to safe programming.

Tip: Perhaps the most difficult part of this assignment is getting your head around the system itself, and the packages already supplied. Be sure to download the provided code early, and to understand the example scenario and how it uses the interfaces of the supplied packages.

2 System overview

The system to be produced as part of the assignment is part of a *simulation* of a real device called an *electronic braking system* (EBS).

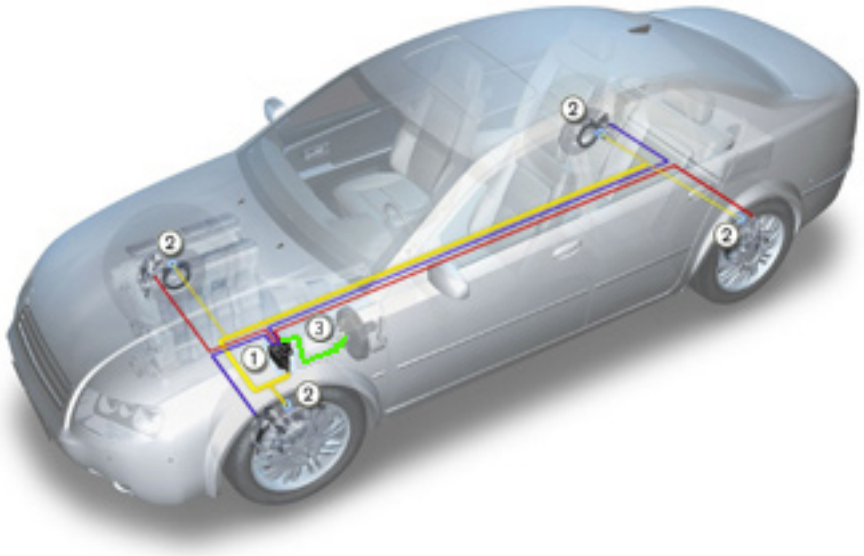

Figure 1: Electronic Braking System (EBS)

The EBS is a software-based system for controlling the brakes in a car. It monitors braking commands sent over an internal CAN bus between Embedded Control Units (ECUs) in the car. The EBS is run on an ECU that monitors the speed of the car's wheels via *speed sensors* and controls the wheels' brakes. The EBS controls the brakes in response to both the messages received on the CAN bus as well as changes in the wheels' speed. In particular, it implements

Anti-Lock Braking (ABS) functionality that detects when traction has been lost during braking, adjusting braking to regain traction by rapidly cycling the brakes on and off.

Figure 1 shows an illustration of an EBS as it would be in a car, identifying ① the ECU on which the EBS software runs, called the *EBS ECU*; ② the speed sensors attached to the car's wheels, and ③ the brake pedal. In this fictitious "brake by wire" scenario, when the driver presses their foot down on the brake pedal, a message is sent on the CAN bus (depicted as a green coloured line) to the EBS ECU instructing it to apply the brakes. The EBS ECU is connected to the car's brakes (via the red and blue lines in the figure). The EBS ECU receives information from the speed sensors (connected via the yellow lines) which it uses to detect when the wheels have lost traction and anti-lock braking needs to be engaged.

In this project we will consider a simplified setup in which there is only a single wheel, brake and speed sensor as depicted in Figure 2. Figure 2 shows the closed-loop control for the system's components. The *CAN Bus* transmits *CAN Messages* to and from the *Brake Controller* component (running on the EBS ECU). Some of these messages instruct the Brake Controller to change the amount of *Pressure* to apply to the *Brake*. The Brake in turn translates brake pressure into *Deceleration* applied to the *Wheel*, whose *Velocity* is detected by the *Speed Sensor*. The Speed Sensor allows the Wheel's *Measured Velocity* to be tracked by the Brake Controller. The Measured Velocity might differ from the Wheel's (actual) Velocity due to inaccuracies in the Speed Sensor, which are exacerbated at lower velocities.

The Brake Controller uses the Measured Velocity data to determine whether the Wheel has lost traction (i.e. whether it has become *locked*), and if so initiates Anti-Lock Braking by rapidly modulating the Pressure to the Brake, as described below, which will affect the Wheel's Velocity. Thus the Measured Velocity from the Speed Sensor creates a feedback loop in the system, making this system a simple example of a closed-loop control system[1]. This loop continues unbounded.

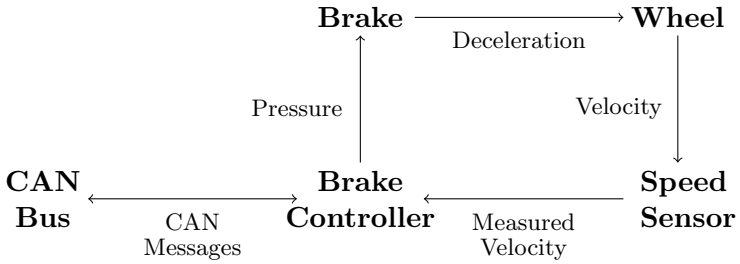

Figure 2: EBS configuration and data flows.

The safety-critical aspects of this system are clear: if the Brake Controller applies the wrong Pressure to the Brake at the wrong time, the car may fail to stop when required, amongst other unsafe effects.

[1] `http://www.electronics-tutorials.ws/systems/closed-loop-system.html`

3 User requirements

3.1 Definitions

D1 *kmph* (*kilometres per hour*): the unit of wheel velocity. For simplicity in this assignment, we use linear, rather than radial, velocity, and adopt the standard unit of vehicle speed.

D2 mps^2 (*metres per second per second*): the unit of *acceleration*.

D3 *deceleration*: negative acceleration. When Pressure is applied to the Brake, the Brake translates this Pressure into deceleration on the Wheel (a negative quantity in mps^2), reducing the Wheel's velocity (kmph).

D4 *Pressure*: A quantity representing the amount of deceleration the Brake applies to the Wheel. For simplicity, in this assignment, Pressure is measured in unspecified units between 0 (no pressure) and 1 (maximum pressure).

D5 *Wheel Lock*: A condition that occurs when the Wheel suddenly stops spinning while the Brake is applying deceleration to it, while the car is still moving. If nothing is done to correct the Wheel Lock, the car's ability to slow down properly is severely diminished, creating the potential for harm.

D6 *Anti-Lock Braking* (ABS): The process of rapidly modulating the Brake Pressure to correct Wheel Lock during braking, as described below.

D7 *clock tick* (or simply *tick*): A unit of time. For the purposes of this assignment, a clock tick is the most fine-grained unit of time, and is equal to $\frac{1}{10}$th of a second.

3.2 Detecting Wheel Lock

Methods for detecting Wheel Lock have evolved with the development of EBS systems over many decades.

For the purposes of this assignment, we will use just a simple method for detecting Wheel Lock that is applicable to our scenario with a single Wheel and Speed Sensor. The basic idea is to detect when the Wheel rapidly decelerates: in particular, when its Measured Velocity drops by an amount greater than or equal to the *velocity change threshold* during a *single* clock tick. The default velocity change threshold is 10 kmph; however this setting can be changed by authorised persons.[2] Writing this mathematically, detecting Wheel Lock involves checking that:

$$MeasuredVelocity(now) - MeasuredVelocity(now - 1) \geq VelocityChangeThreshold$$

where *now* is the most recent tick, and $MeasuredVelocity(t)$ denotes the Measured Velocity at tick t.

The Speed Sensor is known to be unreliable when the Wheel's actual Velocity is below 10 kmph, however. Therefore, the system should attempt to detect Wheel Lock as described above, only when the average of the Wheel's Measured Velocity as taken at each of the previous *four* (4) clock ticks is greater than or equal to the *average velocity threshold*. The default average

[2] If velocity drops by 10 kmph in a single clock tick ($\frac{1}{10}$th of a second), this implies a deceleration of 100 kmph per second, or ≈ 27.78 mps^2, which is well above the maximal deceleration of average cars (about 10 mps^2).

velocity threshold is 15 kmph; but this setting can be changed by authorised persons. Writing this mathematically, Wheel Lock should be detected only when:

$$\frac{\Sigma_{t=now-4}^{now-1}\ MeasuredVelocity(t)}{4} \geq AverageVelocityThreshold$$

3.3 Principals

There are three roles that take part in this system, each of which is also called a *Principal*:

1. *Driver*: A person who drives the car that has the EBS.
2. *Manufacturer*: A person who manufactures the EBS and might perform certain repair or maintenance activities on it. A manufacturer is able to perform certain actions on the EBS, such as reading and changing its settings.
3. *Mechanic*: A person who might perform certain maintenance activities on the EBS. A mechanic is able to perform certain actions on the EBS, such as reading its settings.

4 CAN Bus Messages

The CAN Bus component transmits messages to and from the Brake Controller, and so provides the external interface to the closed-loop system. Some of these messages inform the Brake Controller about how to adjust the Brake's Pressure, while others are used to query or change the Brake Controller's settings or to read its data.

For instance, an application running on the entertainment system ECU might allow the Driver to obtain historical data about the Wheel's Measured Velocity. It could do so by periodically sending *VelocityRequest* messages (see below) to the Brake Controller, on behalf of the Driver, requesting it to respond with a message containing the Wheel's current measured velocity.

The following *message types* are supported:

BrakePressureUpdate: informs the Brake Controller that the amount of pressure applied by the Driver to the brake pedal has changed. This message contains a data field *Pressure* indicating the new pressure to apply to the Brake.

EngineOn: informs the Brake Controller that the car's engine has been turned on.

VelocityRequest: requests that the Brake Controller reply with a *VelocityResponse* message reporting the Wheel's Measured Velocity. The *Sender* data field of this message identifies which Principal it was sent on behalf of: the Driver, Mechanic or Manufacturer. The actions performed by the Brake Controller in response to this message vary depending on the Principal who sent the request.

VelocityResponse: the type of message that can be sent by the Brake Controller on the CAN bus in response to receiving a VelocityRequest message. It contains a data field *Destination* identifying which Principal the message is for (i.e. the Sender of the VelocityRequest message), plus a *CurrentTick* field, indicating the time at which the message was sent in

terms of the number of clock ticks that have elapsed so far. The *CurrentVelocity* field of this message contains the Wheel's last Measured Speed.

ReadSettingsRequest: requests the Brake Controller to reply with a *ReadSettingsResponse* message containing the Brake Controller's current settings as detailed below. The *Sender* data field of this message identifies which Principal it was sent on behalf of: the Driver, Mechanic or Manufacturer. The actions performed by the Brake Controller in response to this message vary depending on the Principal who sent the request.

ReadSettingsResponse: the type of message that can be sent by the Brake Controller on the CAN bus in response to receiving a ReadSettingsRequest message. It contains a data field *Destination* identifying which Principal the message is for (i.e. the Sender of the ReadSettingsRequest message). It contains two data fields: *VelocityChangeThreshold* and *AverageVelocityThreshold* which respectively hold the velocity change threshold and average velocity threshold for activating ABS.

ChangeSettingsRequest: requests the Brake Controller to alter its settings. The *Sender* data field of this message identifies which Principal it was sent on behalf of: the Driver, Mechanic or Manufacturer. It contains two data fields: *VelocityChangeThreshold* and *AverageVelocityThreshold* which respectively hold the new values for the velocity change threshold and average velocity threshold for activating ABS. The actions performed by the Brake Controller in response to this message vary depending on the Principal who sent the request.

ChangeSettingsResponse: the type of message that can be sent by the Brake Controller on the CAN bus in response to receiving a ChangeSettingsRequest message. It contains a data field *Destination* identifying which Principal the message is for (i.e. the Sender of the ChangeSettingsRequest message).

4.1 Modes

The system operates in one of two modes: *off* and *on*.

R1.1 *Off*: In the *off* mode, the closed-loop functionality of the EBS is off to allow its settings to be read or changed by authorised persons.

R1.2 *On*: In the *on* mode, the closed-loop functionality of the EBS is on, meaning that it may dynamically control the Brake, and settings cannot be changed or read.

R1.3 The initial mode is *off*.

R1.4 Turning on the car's engine causes an *EngineOn* message to be transmitted on the CAN Bus. Upon receiving this message, the EBS must immediately transition to the *on* mode.

R1.5 Once in the *on* mode, the EBS cannot transition to the *off* mode.[3]

[3]Until the car loses power when, upon regaining power, the EBS will restart in the off mode.

4.2 Off mode

The system must support the following requirements for the off mode:

R2.1 In the off mode, the Brake must be applied with maximum Pressure always.

R2.2 A Mechanic or Manufacturer can read the settings (the average velocity threshold and the velocity change threshold for enabling ABS), by sending a *ReadSettingsRequest* message on the CAN Bus. When the EBS system receives this message, if the message's *Sender* field is Mechanic or Manufacturer, the system should respond by sending a correct *ReadSettingsResponse* message on the CAN Bus.

R2.3 A Manufacturer can change these thresholds by sending a *ChangeSettingsRequest* message on the CAN Bus. When the EBS system receives this message, if the message's *Sender* field is Mechanic, the system should change the settings to the new ones contained in the message and respond by sending a correct *ChangeSettingsResponse* message on the CAN Bus.

R2.4 The initial *average velocity threshold* for enabling ABS is 15 kmph.

R2.5 The initial *velocity change threshold* for enabling ABS is 10 kmph.

4.3 On mode

The system must support the following requirements for the on mode:

R3.1 If a *BrakePressureUpdate* CAN bus message is received, and ABS is not activated (see the following requirement), the Brake should immediately be instructed to deliver the new pressure. Otherwise, if ABS is enabled, the Brake should deliver the new pressure at the next time that the system instructs it to deliver non-zero pressure.

R3.2 If Wheel Lock is detected (see Section 3.2), ABS should be *activated* immediately as follows. The system should (step 1) immediately remove all Brake Pressure, i.e. instruct the Brake to deliver the minimum Pressure, 0. (step 2) Brake pressure should then be restored as soon as the Wheel has begun to accelerate. (step 3) As soon as the Wheel then decelerates, all Brake Pressure should be removed again, and then step 2 should be repeated. This cycle, of rapidly modulating the brake pressure in this way, should continue until a *BrakePressureUpdate* CAN Bus message is received indicating a zero pressure (i.e. the driver has removed their foot from the brake pedal), at which point ABS is *deactivated* and the Brake instructed to release all pressure.

R3.3 If no *BrakePressureUpdate* message is received on the CAN Bus and ABS is not activated, the system should not adjust the Brake pressure.

R3.4 Settings cannot be read or changed in the *on* mode.

R3.5 The Driver should be able to obtain the Wheel's current Measured Velocity in the *on* mode, by sending a *VelocityRequest* message on the CAN Bus. When the EBS system receives this message, if the message's *Sender* field is Driver, the system should respond by sending a correct *VelocityResponse* message on the CAN Bus.

4.4 Speed Sensor

The Speed Sensor has the following constraints/properties:

1. As is typical of some wheel speed sensors, the Speed Sensor tends to give unreliable readings for the Measured Velocity when the Wheel's actual Velocity is low. For this assignment, this happens when the Wheel's actual Velocity is below 10 kmph. Regardless, there is always a small margin of error in the Speed Sensor's readings of Measured Velocity compared to the Wheel's actual Velocity.

5 Existing packages

The following packages are available for download from the LMS. The first three simulate the CAN Bus, Brake, Wheel, and Speed Sensor respectively:

1. `CANBus`: A simulation of the EBS system's interface to the CAN Bus. Each clock tick the CAN bus can possibly deliver a new message to the EBS system. In this simulation, its behaviour is random.
2. `Brake`: A simulation of the Brake. The Brake is connected to the Wheel and can be instructed to deliver a certain amount of Pressure to the Wheel. It then applies a corresponding amount of deceleration (negative acceleration) to the Wheel.
3. `Wheel`: A (very crude) simulation of a Wheel. The Wheel has a current velocity and acceleration and knows whether it is currently locked. Each clock tick, if it is not locked, it updates its velocity according to its current acceleration (which could be negative). When the Wheel is locked, its Velocity as communicated to the Speed Sensor is zero. For this very crude model, when the Wheel becomes unlocked, it regains the full Velocity it had when it became locked (i.e. we don't model the changes to the car's velocity that might occur during Wheel Lock due to friction with the road).
4. `SpeedSensor`: A simulation of a Speed Sensor. The Speed Sensor is connected to the Wheel. Each clock tick it learns the Wheel's current velocity, which it can then report. The Speed Sensor always has a small margin of error; however at low velocities this margin of error increases.
5. `RandomNumber`: A package used to generate some random values for simulation purposes. You do not need to use this directly as part of your submission.
6. `Measures`: A package containing the Ada types used to represent velocity, acceleration and brake pressure etc.
7. `ClockTick`: A package that defines constants to do with clock ticks (e.g. how many clock ticks are in a second etc.)
8. `ManualSample`: A procedure containing an example simulating a scenario in which the various components are used, but not in a proper control loop.

Note that the first four of these each contain a procedure called `Tick`, which is used to simulate the passage of a single clock tick of time. So, for instance, when the Brake's Pressure is changed, the Brake's `Tick` procedure must be called to then subsequently cause this change to take effect by altering the Wheel's acceleration. Likewise, having done so, the Wheel's `Tick` procedure then must be called to update the Wheel's velocity. Similarly, the Speed Sensor's `Tick` procedure must be called to have it measure the Wheel's new velocity. Finally, calling the CAN Bus's `Tick` procedure causes it to randomly generate a new CAN Bus message, or none at all, which can then be queried.

6 Your tasks

The tasks for the assignment are listed below:

1. Implement an Ada package called `BrakeController`, which implements the functionality to provide the necessary calculations of the Pressure to be delivered by the Brake based on the Wheel's Measured Velocity and Brake Pressure updates from the CAN Bus.

   Note: Your solution should be more general than simply adjusting the Pressure for the single Wheel that is implemented in the `Wheel` package. That is, it should support Wheels with different behaviour.

2. Implement an Ada package called `ClosedLoop` that encapsulates the `BrakeController`, `Brake`, `Wheel`, `SpeedSensor`, and `CANBus` packages to fulfil the behaviour of an EBS system. The package interface should offer just two procedures, neither of which take any arguments:

   (a) `Init`: for initialising the `ClosedLoop`.

   (b) `Tick`: which simulates a single clock tick passage of time. This function is for the purpose of stepping the system through a simulation, and in its implementation it will, amongst other things, call the `Tick` procedure of the `CANBus`, `BrakeController`, `Brake`, `Wheel`, `SpeedSensor`.

   The functions should use the packages outlined in Section 5 to implement the behaviour of the Brake, Wheel, Speed Sensor and CAN Bus respectively.

You are encouraged to modify the code for the purpose of testing etc., however, we will use our own implementations of `Brake`, `Wheel`, `SpeedSensor`, and `CANBus` to run tests as part of the assessment.

7 Criteria

Criterion	Description	Marks
Design	The design of the system is of high quality. The correct components have been included, the design is loosely coupled, and suitable information hiding strategies have been used.	2 marks
Correctness	The implementation behaves correctly with respect to the user requirements. The on mode correctly implements braking, and correctly performs Wheel Lock detection and acts on it appropriately.	2 marks
Completeness	The implementation is complete. All components have been implemented and all user requirements have been addressed.	1 marks
Clarity	The design and implementation are clear and succinct.	1 marks
Code formatting	The implementation adheres to the code format rules (Appendix A).	2 marks
Tests	The implementation passes our tests.	2 marks
Total		10 marks

8 Submission

Create a zip file called *your_username*`.zip` or *your_username*`.tgz`. The file should contain your code for the *complete* EBS system, including code provided by subject staff.

Submit the zip file to the LMS.

9 Academic Misconduct

The University misconduct policy applies to this assignment. Students are encouraged to discuss the assignment topic, but all submitted work must represent the individual's understanding.

The subject staff take plagiarism very seriously. In the past, we have successfully prosecuted several students that have breached the university policy. Often this results in receiving 0 marks for the assessment, and in some cases, has resulted in failure of the subject.

A Code format rules

The layout of code has a strong influence on its readability. Readability is an important characteristic of high integrity software. As such, you are expected to have well-formatted code.

A code formatting style guide is available at `http://en.wikibooks.org/wiki/Ada_Style_Guide/Source_Code_Presentation`. You are free to adopt any guide you wish, or to use your own. However, the following your implementation must adhere to at least the following simple code format rules:

- Every Ada package must contain a comment at the top of the specification file indicating its purpose.
- Every function or procedure must contain a comment at the beginning explaining its behaviour. In particular, any assumptions should be clearly stated.
- Constants and variables must be documented.
- Variable names must be meaningful.
- Significant blocks of code must be commented.

  However, not every statement in a program needs to be commented. Just as you can write too few comments, it is possible to write too many comments.
- Program blocks appearing in if-statements, while-loops, etc. must be indented consistently. They can be indented using tabs or spaces, and can be indented any reasonable number of spaces (three is default for Ada), as long as it is done consistently.
- Lines must be no longer than 80 characters. You can use the Unix command "`wc -L *.ad*`" to check the maximum length line in your Ada source files.