DESIGN OF A CRUISE CONTROLLER IN ESTEREL

Jacob Allen (jall229)
Logan Kenwright (lken274)
COMPSYS 723 Assignment 2

Abstract

This report describes the design processor for a cruise control module as part of the COMPSYS 723 course. First we define a set of functional specifications for a cruise controller, and elaborate them using context diagrams and concurrent finite-state machines. We then demonstrate an implementation of these specifications in the Esterel language.

1. Introduction

A cruise controller automatically regulates the speed of a vehicle to match a defined target speed. We attempt to tackle the design of a cruise controller using a model-based approach. Section 2 describes the specification around which the cruise controller was designed. Section 3 demonstrates the specific design that was developed, modelled using finite-state machines. Section 4 describes how the design was implemented in the Esterel language. Finally, section 5 describes the testing of the system.

2. Specifications

The I/O description is captured in the context diagram below. Dashed lines represent pure signals, and solids valued signals.

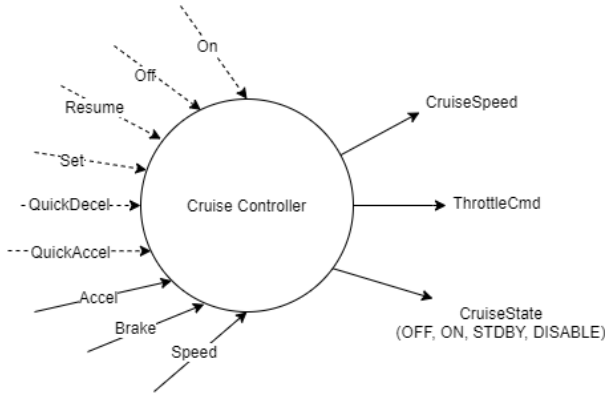

Figure 1 - Context Diagram of the Cruise Controller

To help implement the design, we first refined the system into smaller modules. We separated the inputs and outputs into those which were control dominated, and those which were data dominated. Our first refinement consists of a '*Cruise State*' controller and the '*Driving Control*' logic. The Cruise State module is responsible for managing the overall states of the cruise controller (*OFF*, *ON*, *STDBY*, *DISABLE*). The cruise controller begins in the off state until the 'On' button is pressed by the driver, moving it to the *ON* state. Similarly, the cruise controller can be moved to *OFF* by pressing the 'Off' button. If the car is within a speed limit and has neither pedal pressed, the cruise state will remain *ON*. If the accelerator is pressed or the car is outside the speed limits, the *CruiseState* will move to *DISABLE*. When none of those conditions are still true, it will return to the *ON* state. If the brake is pressed, *CruiseState* will move to *STDBY*, and will remain in *STDBY* until the resume button is pressed by the user. The cruise controller will return to either *ON* or *DISABLE* based on the aforementioned conditions.

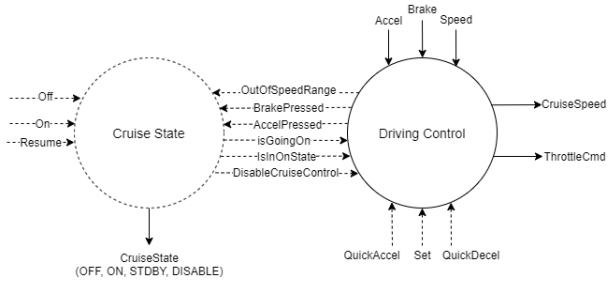

Figure 2 - First Level Refinement of Context Diagram

However, the '*Driving Control*' module is still responsible for the accelerator, brake, throttle, and cruise control logic. We further divide this module into its individual responsibilities, shown in figure 3 below.

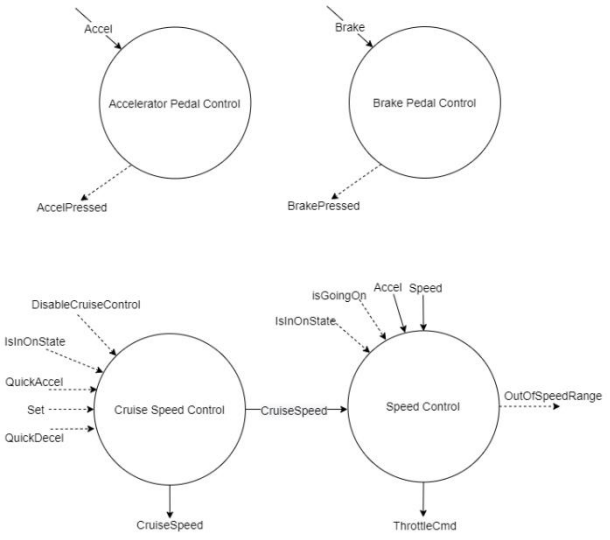

Figure 3 - Refinement of Driving Control module

Each of the pedals are managed by modules which compare the continuous pedal sensor input to a threshold *PedalsMin*, and emit a 'pressed' signal if the threshold is met.

The '*Cruise Speed Control*' module describes the logic determining the target speed for the cruise

controller. The '*Speed Control*' module uses this target speed to determine the amount of throttle needed to reach that speed comfortably.

The cruising speed is managed only when the cruise controller is enabled. When the cruise controlled is first enabled, the cruising speed is set to the current speed. While enabled, the user can use the 'Set' button to update the cruise speed to the current speed. They can also increment or decrement the cruising speed by a fixed amount (*SpeedInc*) using the *'QuickAccel'* and *'QuickDecel'* buttons respectively. The cruising speed can not be set outside of some range between *SpeedMin* and *SpeedMax*. If attempted, the cruising speed will instead be limited to *SpeedMin* or *SpeedMax*.

When the cruise controller is off, the throttle is driven directly by the accelerator pedal value. When in the *ON* state, the target throttle is calculated using a proportional-and-integral (PI) controller as function of the target cruising speed and the current vehicle speed. This throttle output is capped at some limit *ThrottleSatMax* for passenger comfort. To prevent overshoot of the integral term, the integral action is reset whenever the cruise control moves to the *ON* state, and is frozen whenever the current speed is saturated at the target cruising speed.

3. Modelling Behaviour

Each of the specified modules was mapped to a finite-state machine to describe their behaviour. The '*Cruise State*' module is described in figure 4.

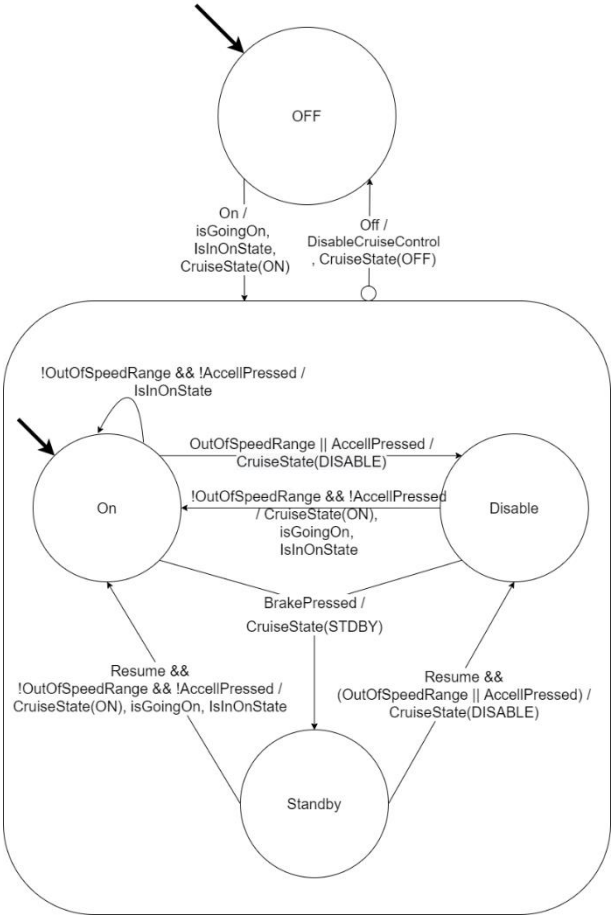

Figure 4 - Cruise State module described as an FSM

Each of the specified cruise controller states is mapped on one state. To prevent duplicate transitions, the *'OFF'* state was moved to a higher hierarchy. Each of the conditions described in section 2 are mapped to a transition on the FSM. Many of the system features depend on whether the system is in the *ON* state, but the speed controller integral action requires a signal specifically on transition. To account for both conditions, *IsInOnState* holds continuously, whereas *isGoingOn* is only emitted on the transitions into the *ON* state.

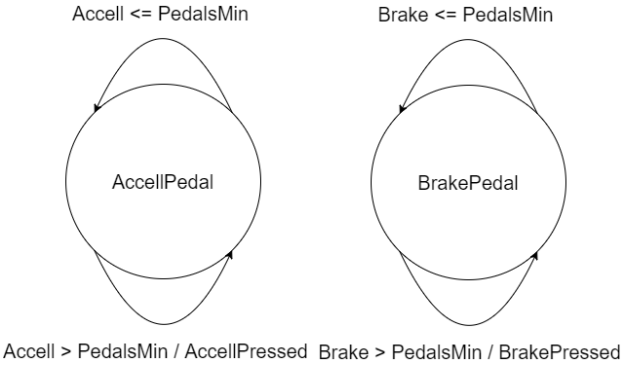

Figure 5 - Pedal monitors shown as concurrent FSMs

Each of the pedals can be described as a single mealy state due to the simplicity. When the analogue *Accel* input reaches some threshold, the binary control signal is emitted.

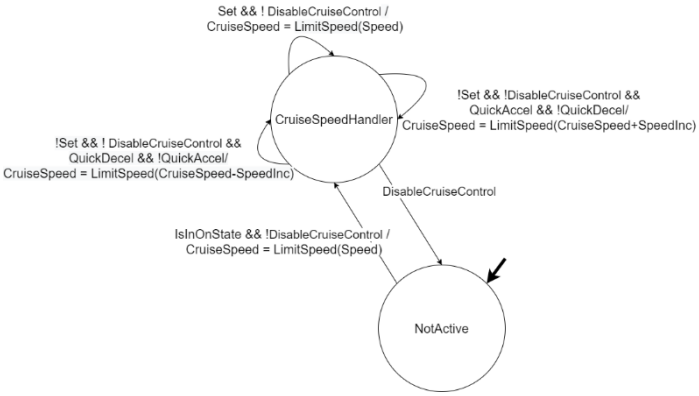

Figure 6 – Cruise Speed Control described as an FSM

The '*Cruise Speed Control*' module is separated into two states. Whenever the '*Cruise State*' FSM is in the *OFF* state, that will be mirrored in the '*NotActive*' state of the '*Cruise Speed Control*' FSM.

Whenever the cruise controller is enabled (that is, in *ON*, *STDBY*, or *DISABLE*), the target cruise speed is managed from within the *CruiseSpeedHandler* state. When the cruising speed is attempted to be modified from the Set, *QuickAccel*, or *QuickDecel* buttons, it must be limited between *SpeedMin* and *SpeedMax* per the specifications. To achieve this, we invoke an external function *LimitSpeed* (described in section 4) which is more suitable for describing data-driven actions than an FSM.

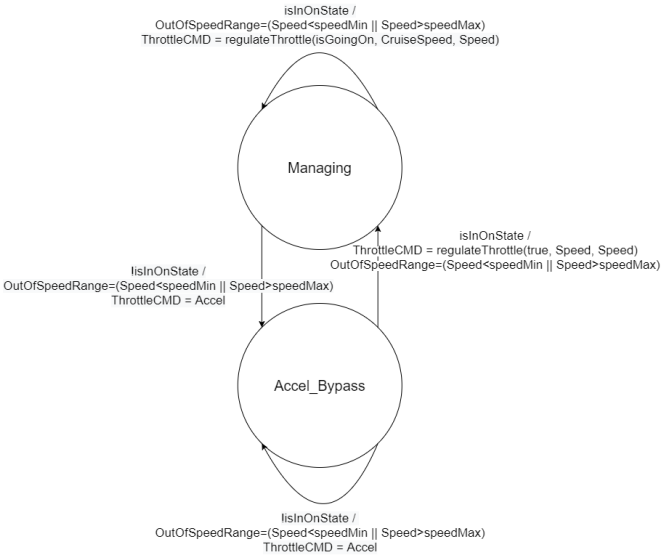

Figure 7 - Speed Control module described as an FSM

The output to the throttle can be sourced from either the accelerator pedal, or automatically managed using a Proportional Integral (PI) controller when the cruise controller is in the *ON* state. Each of these conditions are represented by a state in the FSM. Instead of incorporating the throttle limit into the FSM, we delegate this operation to an external function *regulateThrottle.*

4. Implementation

Esterel syntax allows for a very literal mapping of FSMs.

```
if state = 1 then
    present BrakePressed then
        state := 2;
        emit CruiseState(state);
    else
        present (OutOfSpeedRange or AccelPressed) then
            state := 3;
            emit CruiseState(state);
        else
            emit IsInOnState;
            emit CruiseState(state);
        end present;
    end present;
elsif state = 2 then
    present (Resume and not BrakePressed) then
        present (OutOfSpeedRange or AccelPressed) then
            state := 3;
            emit CruiseState(state);
        else
            state := 1;
            emit IsInOnState;
            emit CruiseState(state);
            emit isGoingOn;
        end present;
    end present
else
    trap t3 in
        present (BrakePressed) then
            state := 2;
            emit CruiseState(state);
            exit t3;
        end present;
        present (not OutOfSpeedRange and not AccelPressed) then
            state := 1;
            emit IsInOnState;
            emit CruiseState(state);
            exit t3;
        end present;
    end trap;
end if;
```

Figure 8- An extract from the Esterel implementation of the Cruise State FSM

We faced several design challenges when working with Esterel. The constructs to evaluate conditions for valued and pure signals vary, in some cases it was necessary to used a combination of both ‘if’ and ‘present’ statements to evaluate one condition. FSM states and temporary data were represented using variables. Our design didn’t demonstrate any inherent causality issues, so the use of the *pre* operator wasn’t necessary.

Certain data-oriented operations were delegated to external C functions, as C is more powerful for describing data-dominated algorithms. We used the provided *regulateThrottle* C function to implement the PI controller which determines throttle output. In addition, we implemented the *limitSpeed* function which prevents the cruise speed from being set outside of the *SpeedMin* and *SpeedMax* bounds.

```
float limitSpeed(float speed)
{
    static const float SPEED_MAX = 150.0;
    static const float SPEED_MIN = 30.0;
    if (speed > SPEED_MAX)
    {
        return SPEED_MAX;
    }

    if (speed < SPEED_MIN)
    {
        return SPEED_MIN;
    }

    return speed;
}
```

Figure 9 - limitSpeed C function

An equivalent module of each FSM was created for:

- *AccelPedal* and *BrakePedal*
- The main ‘state’ state machine, *StateControl*
- The target cruising speed, *CruiseSpeedControl*
- The throttle control, *SpeedControl*

In addition, some top-level modules were required to define the concurrent execution of the low level modules:

- *PedalControl* runs each pedal FSM concurrently
- *DrivingControl* executes *PedalControl* and *SpeedInterface* concurrently
- *SpeedInterface* executes the *SpeedControl* and *CruiseSpeedControl* FSMs concurrently
- *CruiseController*, at the top level, runs *StateControl* and *DrivingControl* currently

Figure 10 demonstrates how concurrent FSMs were executed in Esterel. In particular, note how Esterel allows both implicit and explicit port mapping of signals. Inputs such as Speed and Accel do not need to be mapped due to sharing a common name. In the case of CruiseControl, although the name is shared, the signal needs to be emitted to both the environment and between modules. To solve this, we employ a local signal *LocalCruiseSpeed* which connects the two modules, as well as concurrent logic which emits CruiseSpeed to the environment. Note how concurrent execution in Esterel can combine direct statements and the dispatch of lower-level modules.

```
module SpeedInterface:

% Inputs
input IsInOnState;
input isGoingOn;
input Speed : float;
input Accel : float;

input Set;
input QuickAccel;
input QuickDecel;
input DisableCruiseControl;

% Outputs
output OutOfSpeedRange;
output ThrottleCmd : float;
output CruiseSpeed : float;


signal LocalCruiseSpeed := 0.0f : float in
    loop
        present LocalCruiseSpeed then
            emit CruiseSpeed(?LocalCruiseSpeed);
        end present;
        pause
    end loop
    ||
    run CruiseSpeedControl [signal LocalCruiseSpeed/CruiseSpeed]
    ||
    run SpeedControl [signal LocalCruiseSpeed/CruiseSpeed]
end signal
end module
```

Figure 10 - The SpeedInterface module demonstrating concurrent execution of lower-level modules

The provided makefile expects the input of a single Esterel file. We quickly found this lack of separation between modules limiting. It increased the difficulty of testing individual components and would cause merge conflicts during collaborative development. To solve this issue, we developed a build script which collates multiple Esterel files into a single file at compile time, allowing us to develop and test modules in individual files.

```
# Copy then delete each strl files into RUN_FILE..
for EsterelFile in *.strl; do
    if [ $EsterelFile != $TARGET_ESTEREL_FILE ]; then
        echo "Moving $EsterelFile --> $TARGET_ESTEREL_FILE"
        echo "" >> $TARGET_ESTEREL_FILE
        cat $EsterelFile >> $TARGET_ESTEREL_FILE
        rm -f $EsterelFile
    fi
done
echo ""

# Move the delete all C files into 1
for CFile in *.c; do
    if [ $CFile != $TARGET_C_FILE ] && [ $CFile != 'ctype.c' ]; then
        echo "Moving $CFile --> $TARGET_C_FILE"
        echo "" >> $TARGET_C_FILE
        cat $CFile >> $TARGET_C_FILE
        rm -f $CFile
    fi
done

# ensure a .h is present
touch $RUN_FILE.h

echo -e "\n\nBuild File Setup Complete\n"
make $RUN_FILE.xes
./$RUN_FILE.xes
```

Figure 11 - Custom build script allowing multiple file compilation

5. Testing

A basic test case was provided in the form of text files containing some inputs and corresponding outputs. Due to historic complications with the automatic testing tools, we were encouraged to enter the values manually. The provided test cases were not comprehensive, only testing the *OFF* and *ON* states, meaning we had to create additional test scenarios based on the specifications. A classmate later provided a testing script which could be used to pipe input from a file into the Esterel executable, allowing us to run tests systematically. Our modified test script has been included with the submission for completeness, but we do not claim credit for its development (see readme).

6. Conclusions

We propose an implementation of a cruise control system in the Esterel language based on a model-based approach. This implementation was developed based on a series of high-level and FSM models presented in this report. This implementation satisfies all specifications and requirements outlined in the project brief. Thorough manual and automatic test cases have been applied to verify functional correctness.