### **Design of a Cruise controller in Esterel**

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

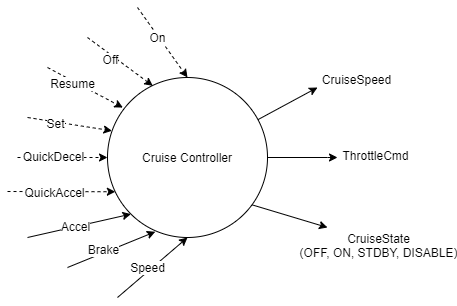
Abstract

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

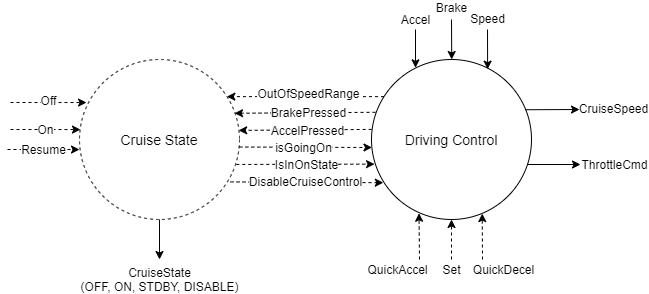
# Introduction

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. Finally, section 4 describes how this design was implemented using the synchronous Esterel language.

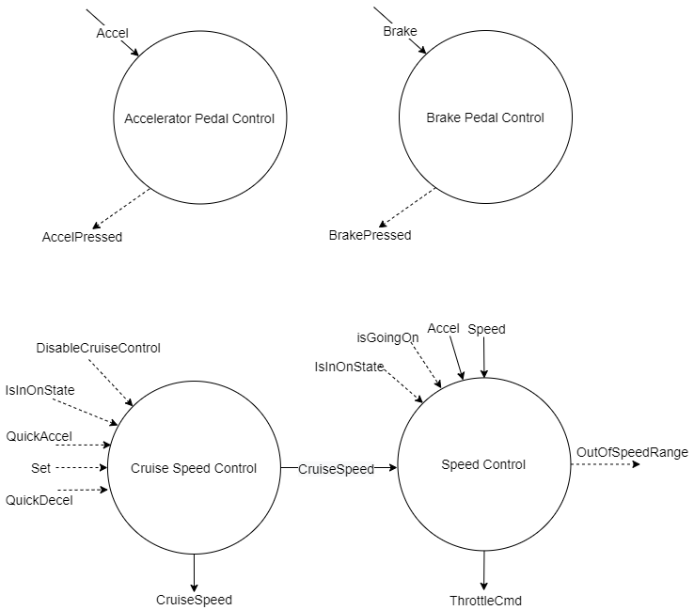
# Specifications

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

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.



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.



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.

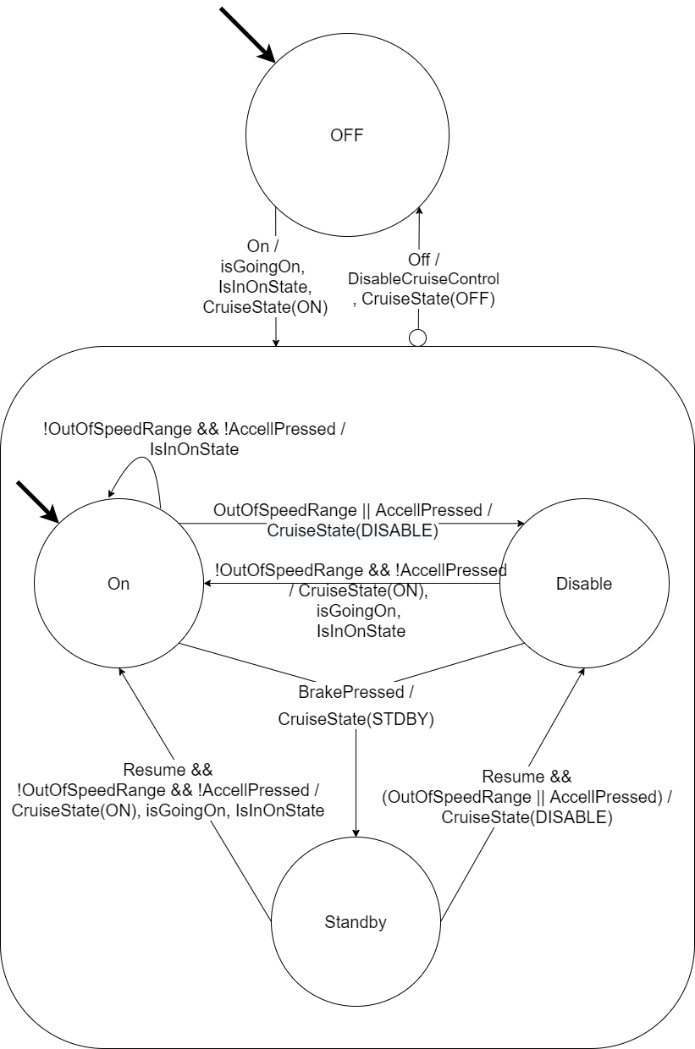
The ‘*Cruise Speed Control*’ module describes the logic determining the target speed for the cruise controller. The ‘*Speed Control*’ module uses this target speed to determine the amount of throttle needed to reach that speed comfortably.

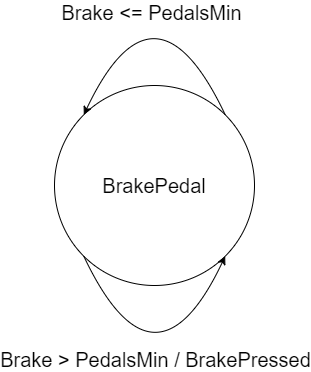
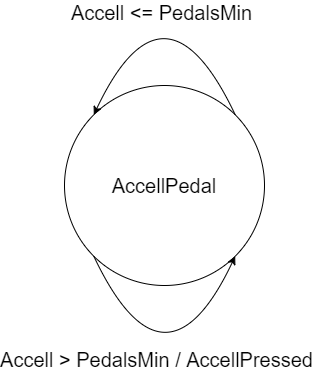
The cruising speed is managed only when the cruise controller is enabled. When the cruise controlled is first enabled, the cruising speed is set to the current speed. While enabled, the user can use the ‘Set’ button to update the cruise speed to the current speed. They can also increment or decrement 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.

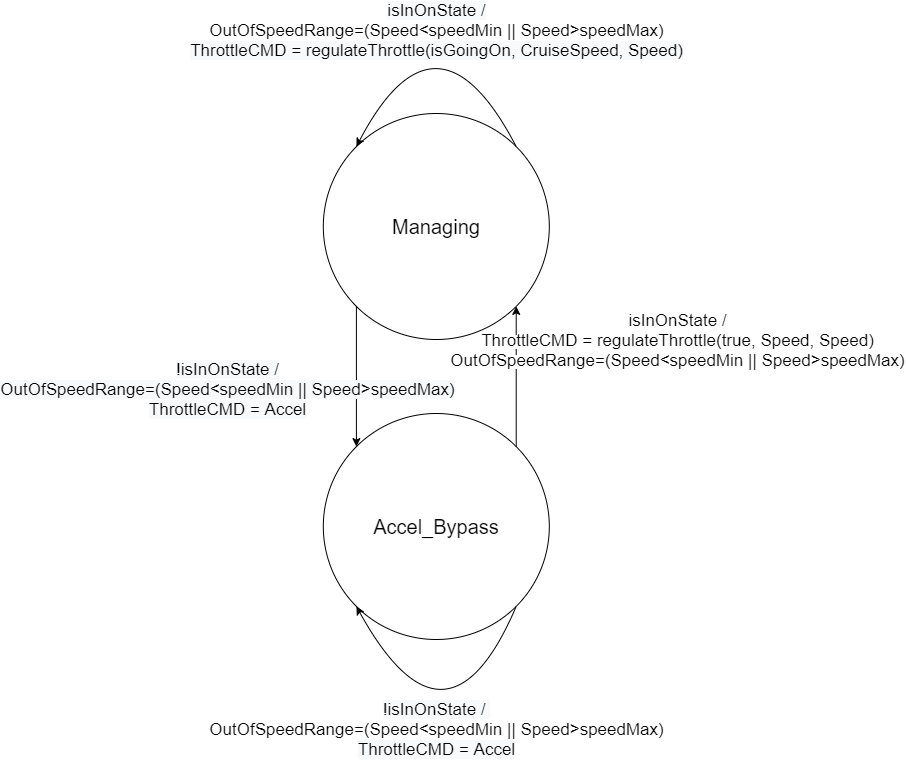
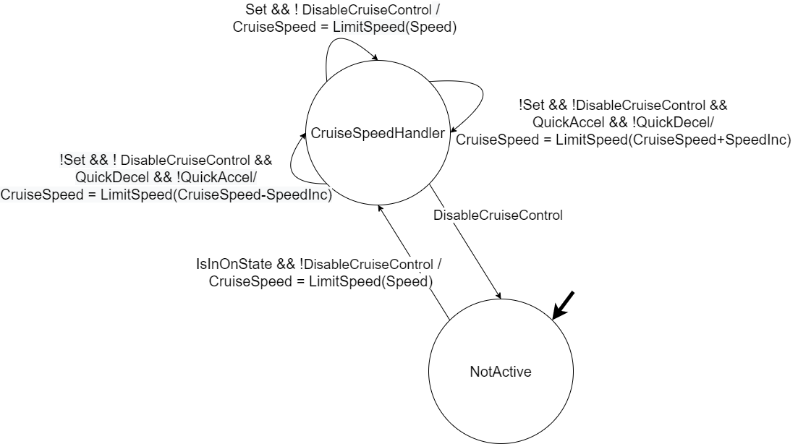
When the cruise controller is off, the throttle is driven directly by the accelerator pedal value. When in the ON state, the target throttle is calculated using a proportional-and-integral (PI) controller as function of the target cruising speed and the current vehicle speed. This throttle output is 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.

# Modelling System as FSMs

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. To prevent duplicate transitions, the ‘OFF’ state was moved to a higher hierarchy.







# Implementation

* FSM to Esterel mapping (one example?)
* Design decisions
  + Separate files allowed easily independent testing
  + Esterel constructs – traps, variables
  + Challenges – working with different constructs for valued/pure signals?

# Conclusions