COMPSYS 723 ASSIGNMENT 2 REPORT TEAM 10

Cruise Controller System Design using Esterel

Krishen Chovhan & Krithik Lekinwala

Team 10

The University of Auckland

Abstract

As cars become more and more futuristic, so do their autonomous capabilities, from electric seats to autopilot. This report will outline an implementation of a cruise control system using Esterel by utilising the concurrency of components that would typically be present in a cruise control system for a car. This report outlines how these components were implemented, the testing results, and the potential issues raised while developing this system.

1. Introduction

This report will explore and showcase the design process of developing a cruise controller system. Specifically, Esterel will be used to create the state machines and synchronous aspects of the system. Alongside this, C will be used for mathematical calculations needed for necessary inputs such as pedal detection and Kp/Ki throttle regulation.

2. Planning

2.1. Project setup

The project was completed using Esterel V6, VSCode, and VirtualBox running Ubuntu to compile the written code. GitHub was also used for version control and collaboration. Finally, LucidChart was used for designing a variety of function decompositions and state machines necessary for the cruise controller system.

2.2. Work division

All diagrams, including the system context diagram, refinement, and state machines, were completed together. Regarding the Esterel aspect of this project, there were four defined submodules alongside a testing aspect. Specifically, the Speed Management and Pedal Detection modules were completed by Krithik, whilst the Car Driving Control and Cruise State Management modules were done by Krishen. The testing of inputs and expected outputs were completed in tandem. The most challenging part of implementing the cruise control system was testing the various inputs and outputs correctly. Test cases were written thoroughly, using the brief as a guide, and checked that our expected outputs matched the system outputs. This was especially difficult for checking if the *throttleCmd* output was being calculated correctly using the C functions given to us. Approximately 20 hours were dedicated each to the design/implementation, and another 5 hours for the report completion.

3. Design Software Architecture

3.1. System context diagram

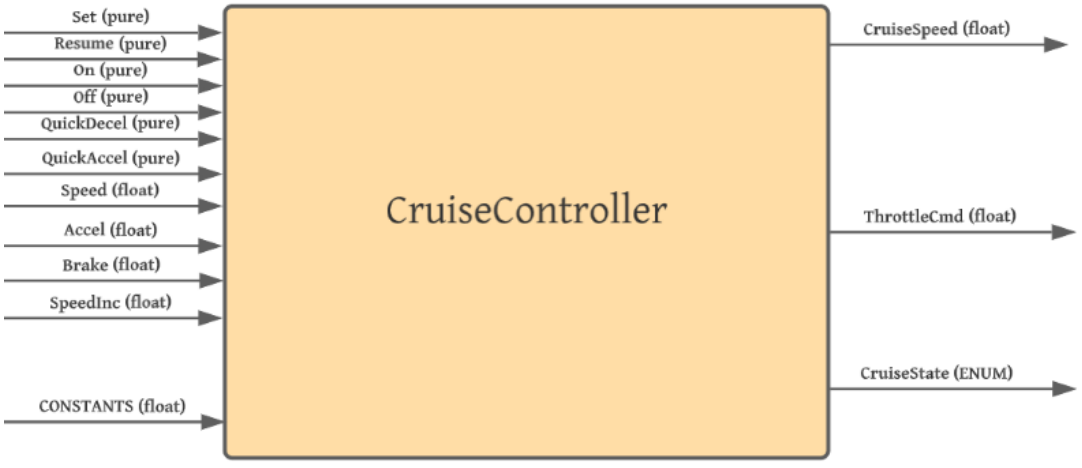

Figure 1: System context diagram

3.2. Control flow diagram

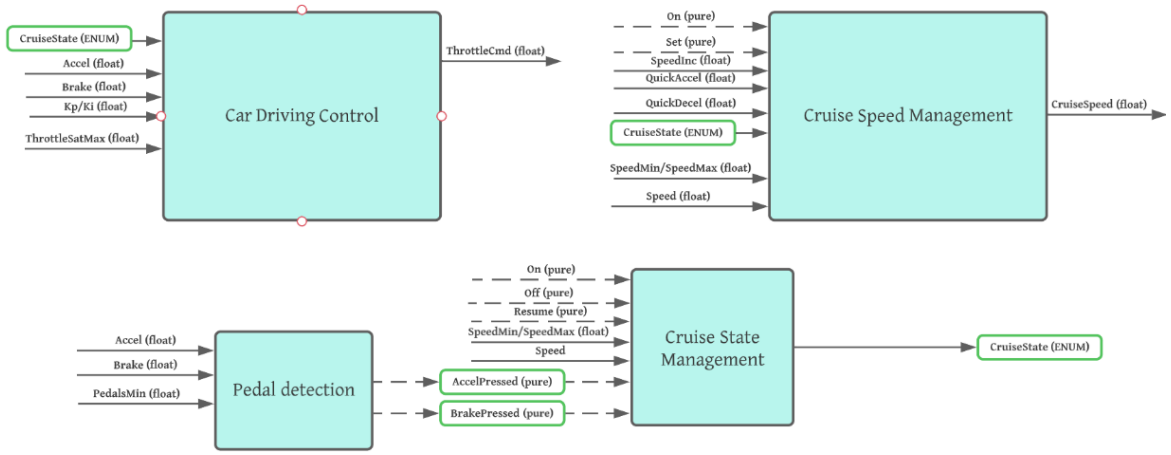

Figure 2: Control flow refinement

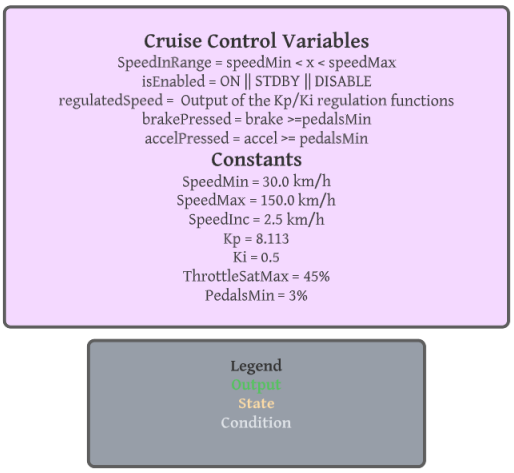

Figure 3: Cruise control variables and legend

3.3. Finite state machine

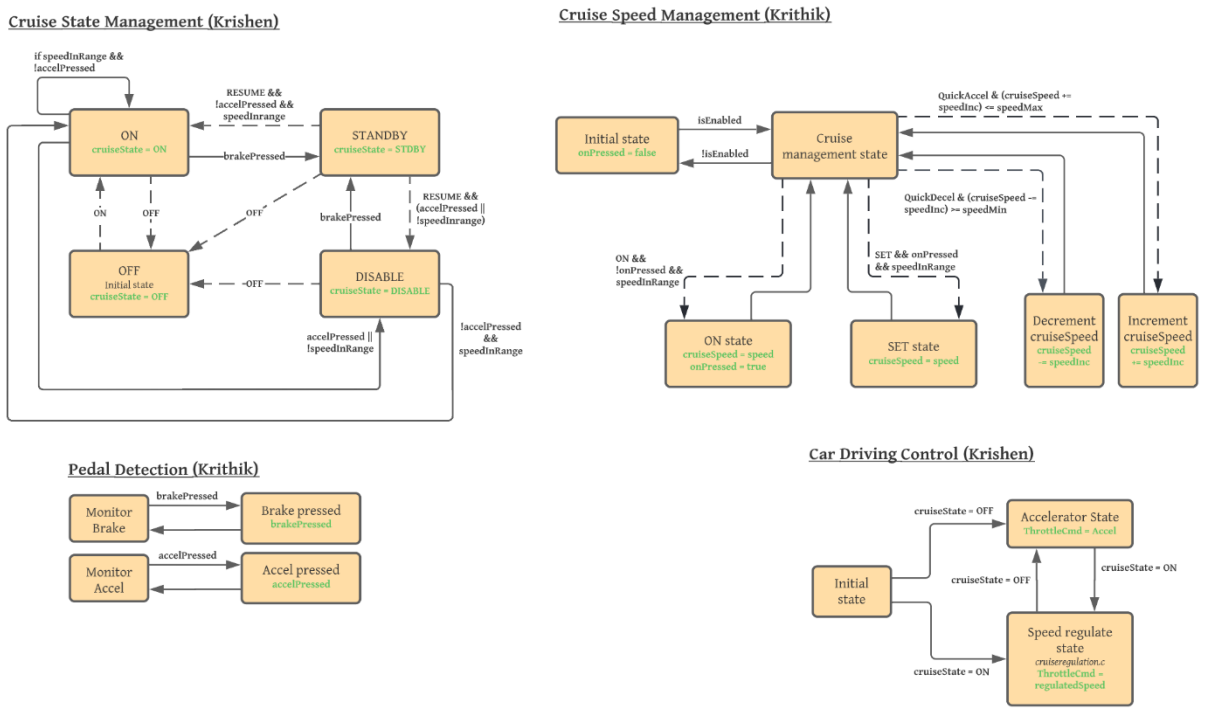

Figure 4: Finite state machine

3.4. System components

Pedal detection

This component had a relatively straightforward implementation. If the accelerator or brake sensor value had a value higher than 3%, the respective C function would output true. Otherwise, the function will output false. The function output is an integer but is read as a Boolean in Esterel. This design consideration was taken because the *bool* keyword was unrecognised as a type in C when compiling the Esterel file.

Initially, pedal detection was designed to be a submodule that would work concurrently with the rest of the modules. However, it was soon realised that to use our implementation correctly; the pedal detection cannot have an input and output to obey the assignment requirements. Due to this, the pedal detection is implemented as functions in C and referred to in Esterel using the *function* keyword.

Speed management

The speed management component oversees what the cruiseSpeed will be given a series of inputs. A specific flag called *isEnabled* is used to check if the cruiseState is either ON, STDBY, or DISABLE. Another flag, *onPressed*, is used to track if the on button has been pressed. The value of *onPressed* is also reset each time the value of *isEnabled* is set to true. Initially, if the on button is pressed, the cruiseSpeed will be equal to the *speed* of the vehicle. If the user wishes to change the cruise speed after using the on button once, the *set* input must be true, and the current car speed must be in range.

If *quickAccel* or *quickDecel* are pressed, the program will compute if the respective increment or decrement of the *cruiseSpeed* will result in a value outside of the speed range. If this is the case, the speed will not increment or decrement. If the projected speed is within the range, the speed will be updated and output. To implement this state to avoid causality issues, most of which came from accessing the speed directly instead of only emitting it once. The *cruiseSpeed* is emitted once within the system to prevent multiple emits in the XES GUI. 2 ticks are required for the speed to settle.

Cruise state controller

The purpose of the cruise state controller is to keep track of and determine the value of the cruise state. Specifically, the cruise state controller analyses the current speed, current acceleration, current brake value, speed limits, the on button, resume and the off button. It is important to note that the pedal detection occurs via a C function, as discussed in the 'Pedal Detection' section. The cruise state controller ultimately outputs the current cruise state as an integer where '1' represents the OFF state, '2' represents the ON state, '3' represents the STDBY state and '4' represents the DISABLE state. Originally the output was going to be an enumeration type, however, due to many difficulties in trying to define a type and compiler errors an integer was used instead.

Upon initialisation, the state is set to be OFF/'1'. Most of the state transitions are based on the current car speed and if the accelerator pedal is pressed. It is important to note that pressing the on button will from this state will cause the output to be ON/'2'. Pressing the off button from any state will cause the output to be OFF/'1'. Pressing the brake from either the ON or DISABLE state will result in the output being STDBY/'3'. If the brake is pressed when the state is OFF, the *cruiseState* will remain unchanged. The reasoning is that if the cruise control is OFF, the system can't really change to a standby mode (a type of active mode). The off button has the highest priority, followed by the on the button and lastly the brake –which was a deliberate design choice to ensure a deterministic output if all three are pressed. It was assumed that the brake and accelerator pedals will not be pressed simultaneously. For a more detailed view of how the cruise state controller operates, please refer to Figure 4.

Car driving control

The purpose of the car driving control module is to manage how the car speed is regulated. The input signals to this module are the accelerator pedal value, current car speed, cruise speed (float) and the cruise state (integer). The output is the *throttleCmd* which is a float. When the cruise state is not on, the car speed is driven by the accelerator pedal. If the cruise state is on, the car speed is managed by a proportional and integral algorithm –which is represented by a C function. This function takes in the cruise speed, the car speed and the *isGoingOn* signal. *isGoingOn* is an internal variable and is important because it is utilised to reset the integral error every time the ON state is just entered. The output of this function is the *throttleCmd*, which is emitted as an output to this module. The t*hrottleCmd* is equal to the *regulatedSpeed* where the output of this value will be dependent on if the throttle has saturated or not.

isGoingOn is set to zero whenever the cruise state is not ON/'2'. When the cruise state is set to ON/'2', *isGoingOn* is set to one – which allows the integral error to be reset. It is important however that if the cruise state remains ON, the integral error should not constantly be reset. To prevent constant resetting, a flag – *enteredON* is set. Once this flag is set, *isGoingOn* is set to zero thus preventing constant resets. *enteredOn* is then reset whenever the state changes away from the ON state. It is important to

note that the regulation is frozen when the throttle is saturated.

The regulate functions input types were initially Boolean but were changed to an integer type due to the type not being recognised within the pedal detection component. Setting the on button, accelerator pedal, and speed in range, the output cruise state will take 2 ticks to settle to the DISABLE/4 state. This design is by choice to see how the state changes with respect to different inputs being applied at the same time.

Design choices and assumptions

The top-level utilises all the submodules created above in parallel to exhibit concurrency. The program is designed so there is only 1 transition when a tick happens. This is to avoid multiple state changes within the same tick, which could output the incorrect values. Pure signals allow us to *wait* using *present* for the signal before executing code. This avoids casualties within the code and ensures that we don't read or write to the same signal at the same time in different areas. A few assumptions made were making sure the brake moves to the STDBY state when in the ON || DISABLE states only.

Causality is the main issue when it comes to designing a concurrent system, but so are multiple emissions of the signal using *emit*. To avoid this, the outputs have a local variable which is only emitted *once* at the very end of the module. This avoids multiple emissions and keeps the code cleaner as well.

4. Results

4.1. Simple test cases

Output			Expected Output				Pass/Fail
CruiseSpeed	hrottleCm	CruiseState	ruiseSpee	ThrottleCmd	CruiseState	Comments	
0	0	1	0	0	1		PASS
30	45	2	30	45	2	AFTET TWO TICKS goes to disable	PASS
30	51.234	1	30	51.234	1	CHECK BUTTONS HAVE NO EFFECT	PASS
30	51.234	1	30	51.234	1		PASS
30	51.234	1	30	51.234	1		PASS
30	51.234	1	30	51.234	1		PASS
30	51.234	1	30	51.234	1		PASS
30	51.234	1	30	51.234	1		PASS
30	51.234	1	30	51.234	1		PASS
30	0	1	30	0	1	TEST BRAKE DOESNT WORK WHEN OFF	PASS
30	45	2	30	45	2		PASS
30	45	3	30	45	3	CHECK IF BRAKE WORKS	PASS
30	45	4	30	45	4		PASS
30	45	3	30	45	3		PASS
30	45	4	30	45	4	NOT IN RANGE	PASS
30	45	3	30	45	3		PASS
30	0	2	30	0	2		PASS
36.158	0	2	36.158	0	2		PASS
36.158	0.9388	2	36.158	0.9388	2		PASS
36.158	0.9388	4	36.158	0.9388	4	Go to disable	PASS

Figure 5: Simple test cases

The simple test cases involved basic speed increments from 0 to 30. The last two cases involved ON being set to true, with a speed value of 36.049, which is within the speed range. The output value of the speed should be 36.049 with a *throttleCmd* value of 0.9388. This value was calculated by hand first and then compared to our output values which were identical. The output cruiseState was also the same as 2 (ON). It is important to note that the *throttleCmd* takes 2 ticks to settle due to our design choices. These considerations were considered to avoid causality issues when reading/writing to multiple shared variables at once, particularly *cruiseState*.

4.2. Complex test cases

These cases were designed to be significantly more complex to ensure the robustness of our implementation. Notably, the cruise state switching, pure signals, and their associated outputs were tested. The main test case which caused concern is shown in Figure 6 where there was constant switching between states. This was present when switching from the ON or DISABLE states, to the STDBY state when the brake value exceeds 3. Through these test cases, it was noted that the system was still switching to the STDBY state even when in the OFF state. This is incorrect from the interpretation of the brief. When the cruise control is off, there should be no option to move the cruise control state to the STDBY position. Comments alongside the test cases were also added to show what was noticed when testing. A notable observation was the output values taking a maximum of 2 ticks to settle, which is not a cause for concern.

Output			Expected Output				Pass/Fail
CruiseSpeed	hrottleCm	CruiseState	ruiseSpee	ThrottleCmd	CruiseState	Comments	
0	0	1	0	0	1		PASS
0	0	1	0	0	1		PASS
0	51.234	1	0	51.234001	1		PASS
0	51.234	1	0	51.234001	1		PASS
0	51.234	1	0	51.234001	1		PASS
0	51.234	1	0	51.234001	1		PASS
0	51.234	1	0	51.234001	1		PASS
0	89.687	1	0	89.687	1		PASS
0	89.687	1	0	89.687	1		PASS
0	89.687	1	0	89.687	1		PASS
0	89.687	1	0	89.687	1		PASS
0	89.687	1	0	89.687	1		PASS
0	89.687	1	0	89.687	1		PASS
0	89.687	1	0	89.687	1		PASS
0	89.687	1	0	89.687	1		PASS
0	89.687	1	0	89.687	1		PASS
0	89.687	1	0	89.687	1		PASS
0	0	1	0	0	1		PASS
36.049	0	2	36.049	0	2	Took 2 Ticks to settle	PASS
36.049	0.9388	2	36.049	0.9388	2		PASS

Figure 6: Cruise state testing

Output			Expected Output				Pass/Fail
CruiseSpeed	hrottleCm	CruiseState	ruiseSpee	ThrottleCmd	CruiseState	Comments	
0	0	1	0	0	1		PASS
30	45	2	30	45	2		PASS
30	45	4	30	45	4		PASS
30	0	2	30	0	2	Uses last set cruise speed (30)	PASS
32.5	0	2	32.5	0	2		PASS
30	0	2	30	0	2		PASS
30	0	2	30	0	2	CruiseSpeed caps at 30 (speedMin)	PASS
145	0	2	145	0	2		PASS
147.5	21.5325	2	147.5	21.5325	2		PASS
150	44.315	2	150	44.315	2	Throttle command maxes out	PASS
150	45	2	150	45	2	SpeedMax limit reached	PASS
150	45	4	150	45	4		PASS
150	0	2	150	0	2	Goes to ON, speed in range	PASS
150	0	4	150	0	4		PASS
150	45	2	150	45	2	Should use previous cruise speed value	PASS

Figure 7: Pure signal testing

Testing pure signals was the most complex task of all. This involved making sure that the boundary speed values were always adhered to, and that the *throttleCmd* is correctly adjusted based on how close the vehicle speed is to the cruise speed. As the vehicle speed nears the cruise speed every tick, the *throttleCmd* will reduce. If the vehicle speed

remains the same, the *throttleCmd* will accumulate errors over time due to no change in the speed.

5. Conclusion

In conclusion, thorough testing and proper design choices resulted in a robust and effective cruise control system designed using Esterel. Despite the few hiccups we encountered when implementing, the design of the finite state machine and system context design aided thoroughly in creating the final product. In the future, test cases could be automated to save time and ensure more robustness which is a necessity due to being used actively on the road with humans.