COMPSYS 723 Assignment 2 Report

Beck Busch & Francis Cho
Department of Electrical and Computer Engineering
University of Auckland, Auckland, New Zealand

1. Abstract

For this assignment, we designed and implemented a simple cruise control system using the synchronous programming language Esterel. Our implementation consisted of 3 main sections; a system state controller, a cruise speed regulator, and a car speed controller. Alongside these main components, a head module runs everything in parallel, and auxiliary modules provide signals for speed and pedal detection. Rigorous testing using the executable reactive program was carried out to ensure all functional requirements were being met.

2. Introduction

Cruise control is a system that is relatively common in modern cars that allows the car to maintain a constant speed without any input from the driver. This is extremely useful for driving long distances on highways or country roads, as the driver no longer needs to manage the accelerator pedal and can drive more relaxed. Another benefit to cruise control systems is fuel usage, as the car's computer can often provide finer control over the engine than a human driver.

Cruise control systems need to balance comfort for the driver with maximum control over the car, as it can be hazardous to take control away from the driver at any time. This is done by creating a system where the driver can resume control of the car at any time simply by using the pedals, which is already the first step in avoiding a dangerous situation. This means that the driver can simply rely on their existing training and instincts, and the cruise control system will automatically yield control of the vehicle.

2.1. Learning Objectives

The objectives for this assignment are to provide an introduction to high-level synchronous programming in Esterel. Focus is placed not only on the development of Esterel code but also on the preparation and creation of state machine diagrams to assist with the design.

3. Specification

Embedded systems software is often used for precise and safety-critical applications. Due to this nature, it is essential that the specifications of the system are carefully defined and adhered to.

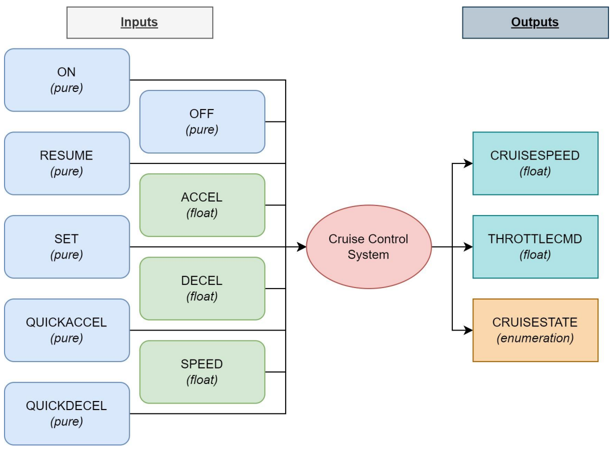
3.1. System Design Requirements

The requirements of this system outline seven parameters for managing the operation of the state machine and speed algorithms. These can be seen in Table 1. Along with these parameters, there are several inputs and outputs to interface the design with the real-world environment. The inputs take the form of buttons that the driver can use to manage the cruise control system, as well as inputs from the car describing the accelerator pedal, brake pedal, and car speed. The outputs from the cruise control system are used to set the speed of the car, as well as provide feedback to the driver. These interfaces are laid out with the top-level context diagram in Figure 1.

Table 1: Operation parameters

SpeedMin	30.0 km/h	Min speed for cc operation
SpeedMax	150.0 km/h	Max speed for cc operation
SpeedInc	2.5 km/h	Push button speed increase
Kp	8.113	Constants for car speed regulation algorithm and control
Ki	0.5	
ThrottleSatMax	45.0 %	
PedalsMin	3.0 %	Sensitivity of car pedals

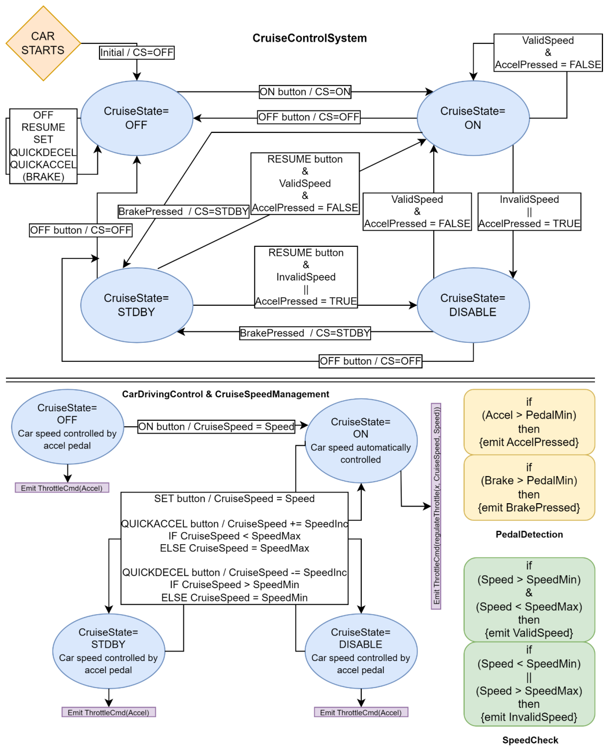
Figure 1: High-level context diagram



3.2. System Design

The operation of our system is defined by the behavioural requirements. When the car starts, the cruise control system should be in the OFF state. In the off state, all inputs from the cruise control buttons should be ignored except for the on button. When the on button is pressed, the cruise state will be set to ON and remain there unless the cruise control operation requirements are broken. For the cruise control system to be managing the speed of the car, the accelerator and brake pedals must not be pressed, and the speed of the car must be within the defined range. If the brake pedal is pressed at any time, the cruise system will enter the standby state and wait to be resumed with the RESUME button. If the accelerator pedal is pressed or the speed ranges are broken, the cruise system will enter the disabled state until these conditions are reversed. Of course, pressing the OFF button during any state will turn off the cruise control system. Our full-state machine diagram can be seen below in Figure 2 or full-sized in the appendix.

Figure 2: System FSM diagram



The driving of the car will also be controlled by our system. When the cruise state is off, disabled, or on standby, our system will pass the accelerator pedal input to the throttle command and return this to the car. This will cause the vehicle to react to the accelerator pedal as expected, with full control given to the driver. When the cruise state is on, the stored CruiseSpeed value will be used with the ThrottleCmd to automatically control the speed of the car.

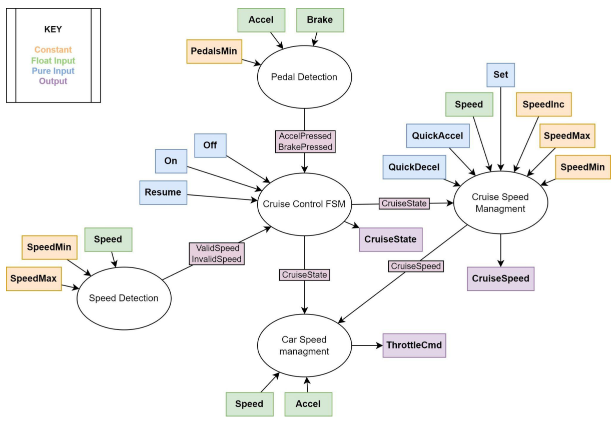
The CruiseSpeed value is altered using the Set, QuickAccel, and QuickDecel buttons. When the cruise control system is in any state except off, pressing the set button will save the current car speed, and pressing QuickAccel or QuickDecel will adjust the saved speed up or down by the constant SpeedInc. The CruiseSpeed value can only be set to a value within the max and min speed range.

4. Design in Esterel

4.1. Design Decisions

Our design implementation contains six modules. We decided it would be beneficial to divide the functionality to separate modular components because it would be beneficial for future refinement, debugging, and testing. We decided to implement separate submodules for speed and pedal detection as this would make the code easier to work with and the end result more robust. All modules excluding the main module, can be viewed in the low-level context diagram in Figure 3.

Figure 3: Low-level context diagram



4.2. System Design

4.2.1. CruiseControlMain

CruiseControlMain is the top-level module that initialises and runs all the other submodules, as well as porting all the signal mappings. This allows the other 5 submodules to be run in parallel. The outputs are CruiseSpeed, ThrottleCmd, and CruiseState, which are directed outputs to the Xeve simulator GUI.

4.2.2. CruiseControlSystem

CruiseControlSystem is a submodule containing our main FSM implementation and receives inputs from multiple submodules. These inputs include signals from the speed validation and acceleration detection submodules (*ValidSpeed*, *InvalidSpeed*, *AccelPressed*, *BrakePressed*) along with pure inputs: *On*, *Off*, *Resume*, *Set*, *QuickAccel*, and *QuickDecel*. In accordance with our initial FSM system design, as seen in Figure 2, we have four different states (ON, OFF, STDBY, DISABLE), and depending on the input signals, our FSM logic implementation outputs the appropriate *CruiseState*. Traps are used in the implementation as a form of weak preemption in order for our loop to continuously run until the conditions are met. This is preferred over aborts which are strong preemptions and will stop the entire FSM loop structure. Present() functions are used for the input signals within each state case. A local integer variable *state* is initialised with 1 (i.e. OFF state) and is updated depending on the conditions. The output *CruiseState* signal is emitted and is utilised by the other submodules.

4.2.3. CarDrivingControl

CarDrivingControl is a submodule that controls and regulates the throttle of our system, which takes inputs: CruiseState, CruiseSpeed, Speed, Accel, and outputs ThrottleCmd. This submodule determines when the car needs to be driven by either the accelerator pedal (Accel inputs) or be automatically regulated when the cruise controller is on (CruiseState = ON). Because we are utilising integer and float data inputs, an if-else logic was implemented to check the current state (CruiseState) and emit the appropriate value for ThrottleCmd. As specified in the requirements, the regulation for throttle was controlled using a proportional and integral algorithm function with defined Kp and Ki factors - regulateThrottle(). As a loop structure is used, traps are employed similarly as described in our *CruiseControlSystem* submodule.

4.2.4. CruiseSpeedManagement

CruiseSpeedManagement is a submodule that regulates the cruise speed based on pure inputs (Set, QuickAcel, QuickDecel). As the specifications stipulate that the cruise speed will only be managed when the cruise control state is enabled (ON, STDBY, DISABLE), we implemented a simple if-else to check the current state of the cruise control system. If it is OFF, then we simply set the cruise speed to 0 otherwise, we proceed to make adjustments based on inputs. Present() functions were used to detect the *Set*, *QuickAccel*, and *QuickDecel* input signals. A local float variable *newCruiseSpeed* was used to temporarily store the value of the CruiseSpeed during calculations before being emitted as the signal CruiseSpeed. When the QuickAccel signal is detected, the cruise speed is incremented with a constant float value *SpeedInc*. The sum of (*newCruiseSpeed* + *SpeedInc*) is checked to ensure it is below constant SpeedMax. When *QuickDecel* signal is detected, similarly, the cruise speed is decremented by SpeedInc, and the sum of (*newCruiseSpeed* - *SpeedInc*) is checked to ensure the validity of speed. Traps are used as a form of weak preemption similar to *CruiseControlSystem and CarDrivingControl* submodules.

4.2.5. PedalDetection

PedalDetection is a simple submodule to detect if the input floats (Accel and Brake) exceed the PedalMin constant (car pedal sensitivity), emitting either AccelPressed or BrakePressed signals.

4.2.6. SpeedCheck

SpeedCheck is another simple submodule used to check if the speed is within the valid operating limits for cruise control. The input float Speed is checked against the constants SpeedMax and SpeedMin, emitting either ValidSpeed or InvalidSpeed signals.

4.3. Data handling and control flow

To deal with data transfer of signals between each of our submodules, we opted to utilise the *pre(signal)* operator available in Esterel. This allows us to access the previous value of the signal without the need to set up and update a temporary variable. This was useful when determining the transition states - such as from OFF to ON. We also make use of trap statements in several places to achieve weak preemption in our control flow.

5. Testing and Verification

To test our design, we used a Linux environment (Ubuntu 20.04) to run the Esterel makefile and the Xeve simulator.

Steps for testing:

- Clone repository or download file
- Open terminal and navigate to src directory
- *> make CruiseControl.xes*
- *> ./CruiseControl.xes*

We first carried out tests with the provided vector table that contained the expected inputs and outputs.

To ensure our finite state machine was transitioning correctly, we came up with a rigorous testing schema shown in Table 2. Screenshots of our testing results can be viewed in the Appendices of this document.

Table 2: Input and output parameters for testing

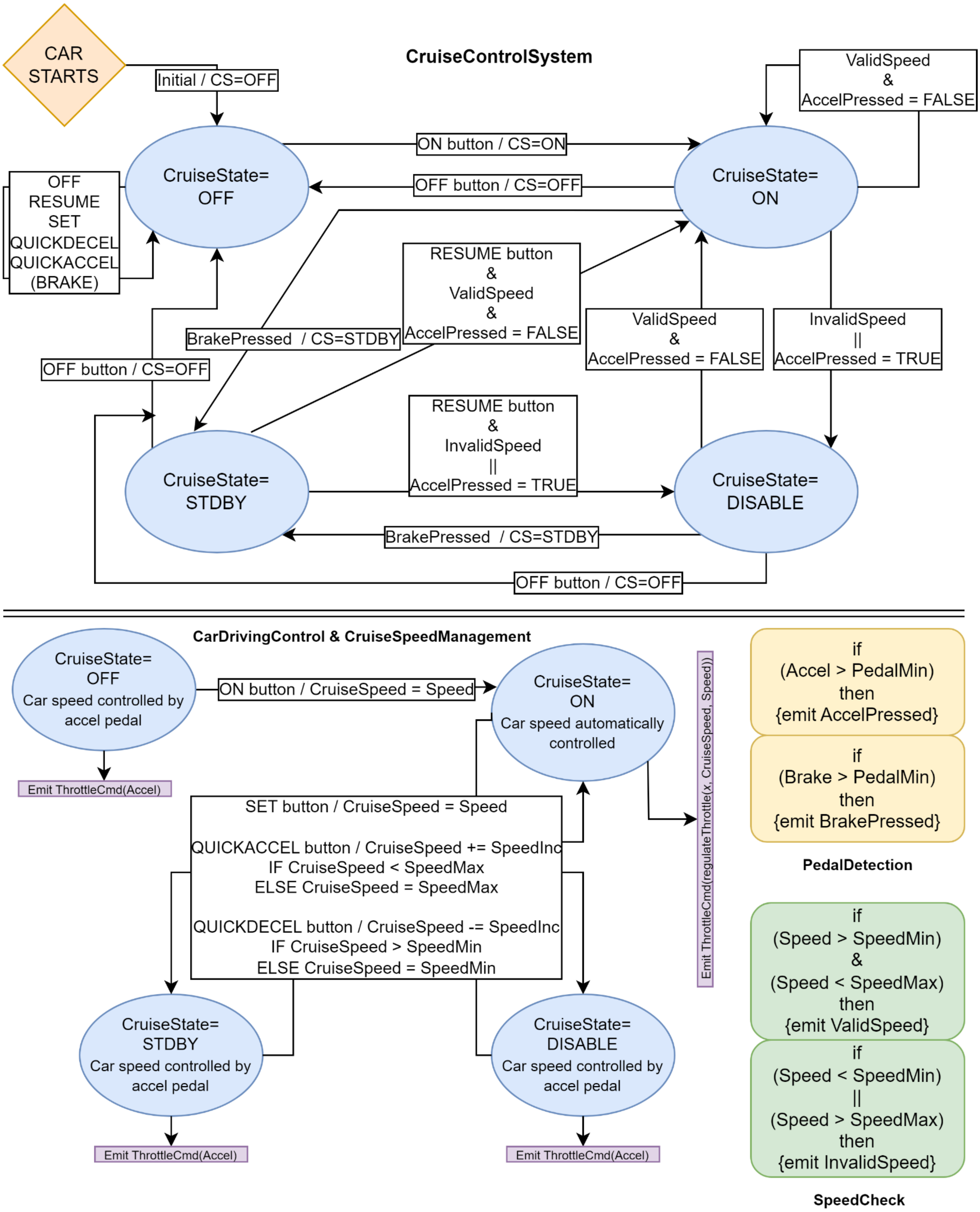
Test case	Initial state	New state	Condition
1	OFF	ON	ON button
2	OFF	OFF/RESUME/SET	OFF/RESUME/SET button
3	ON	OFF	OFF button
4.1	ON	DISABLE	InvalidSpeed or AccelPressed *MinSpeed = 25.0*
4.2	ON	DISABLE	InvalidSpeed or AccelPressed *MaxSpeed = 160.0*
4.3	ON	DISABLE	InvalidSpeed or AccelPressed *AccelPressed = 4.0*
5	ON	ON	ValidSpeed and not AccelPressed
6	DISABLE	ON	ValidSpeed and not AccelPressed
7	DISABLE	STDBY	Brake
8	DISABLE	OFF	OFF button
9	STDBY	ON	Resume & ValidSpeed & not AccelPressed
10.1	STDBY	DISABLE	Resume & (Invalid or AccelPressed) *SpeedMin = 10.0*
10.2	STDBY	DISABLE	Resume & (Invalid or AccelPressed) *SpeedMax = 180.0*
10.3	STDBY	DISABLE	Resume & (Invalid or AccelPressed) *AccelPressed = 15.0*
11	STDBY	OFF	OFF button

6. Conclusion

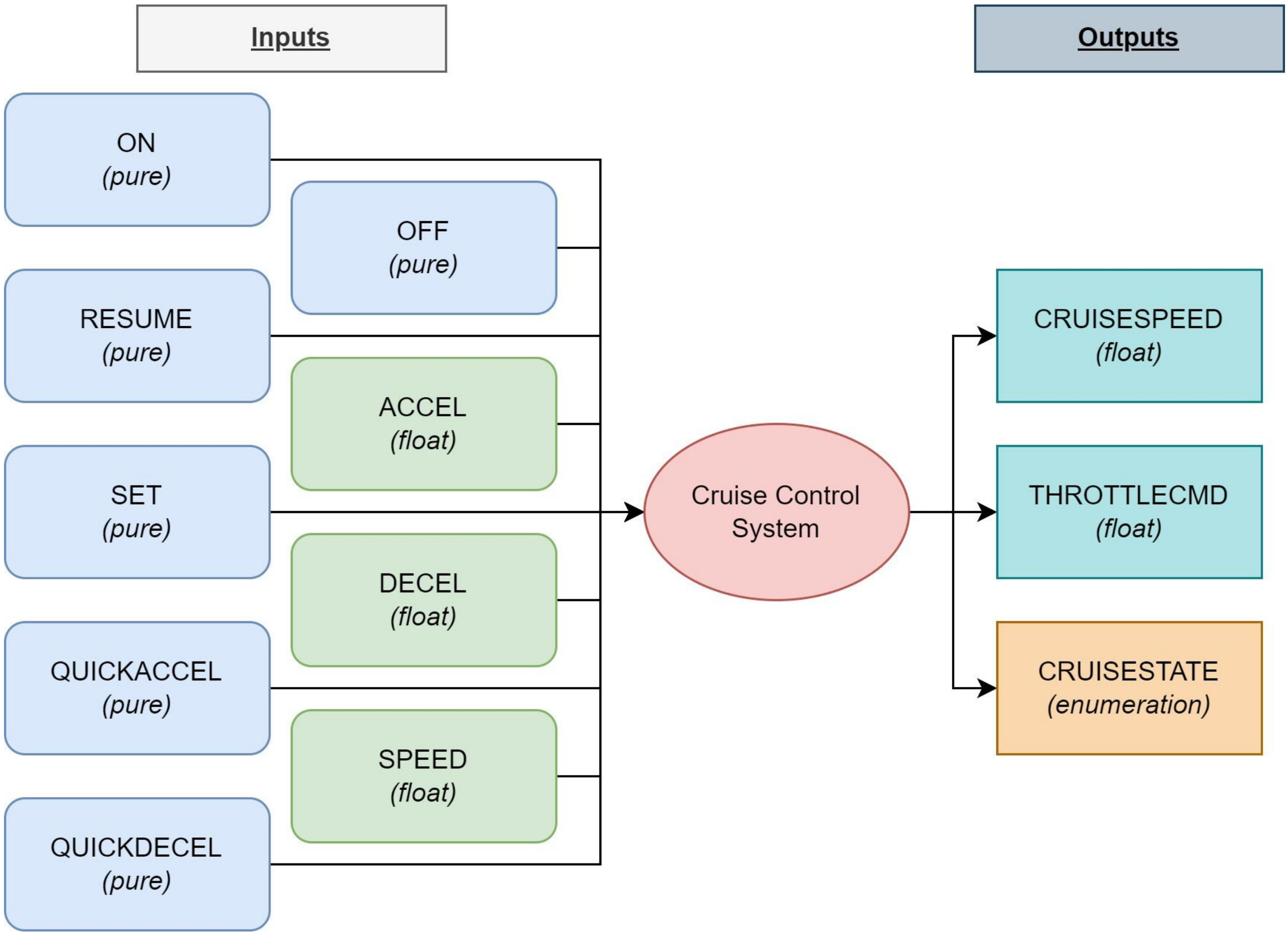
In conclusion, we have successfully designed and implemented a cruise control system following all the specifications provided. We verified our implementation was working through a series of test cases that effectively verified data transfer and state transitions.

7. Appendix

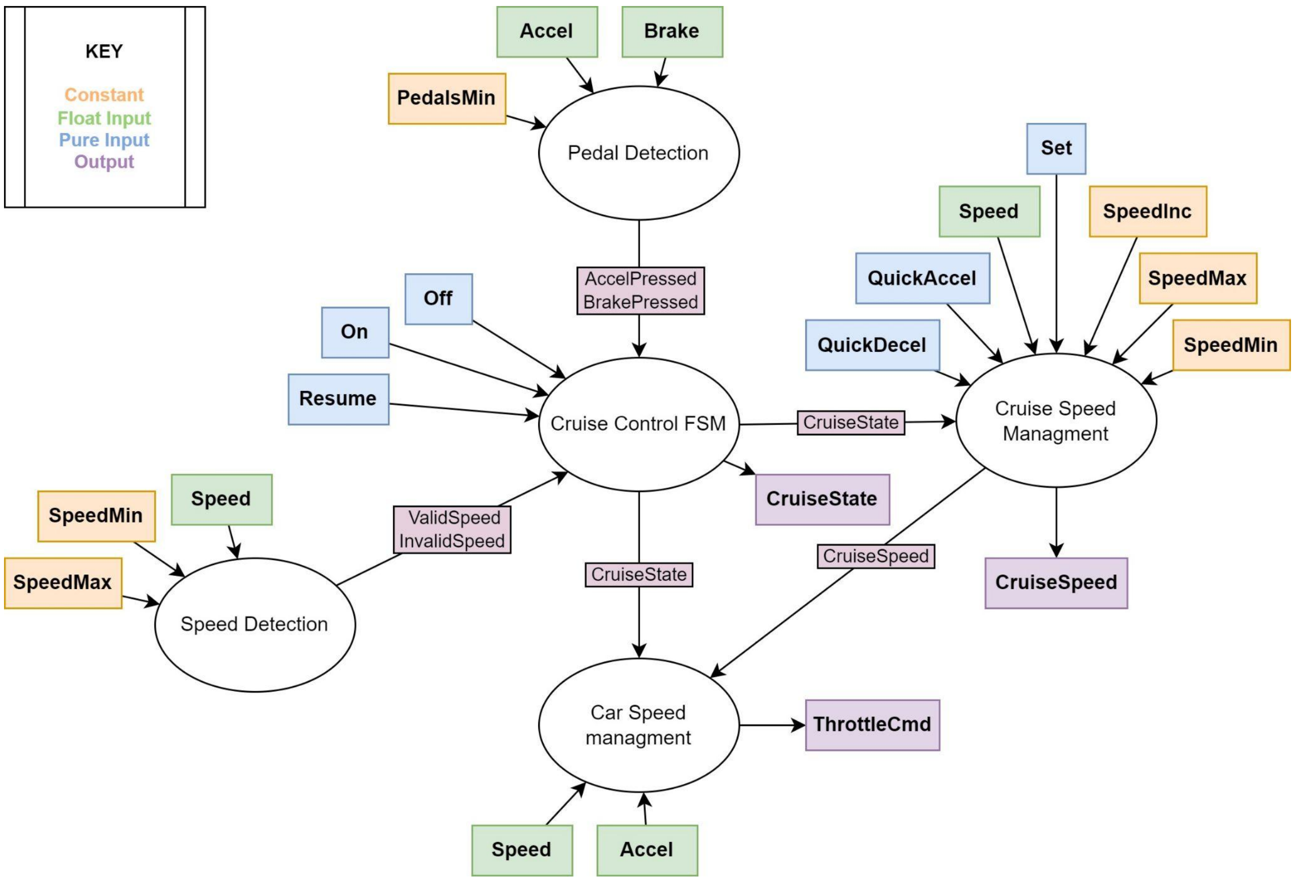
7.1. Full-size FSM diagram



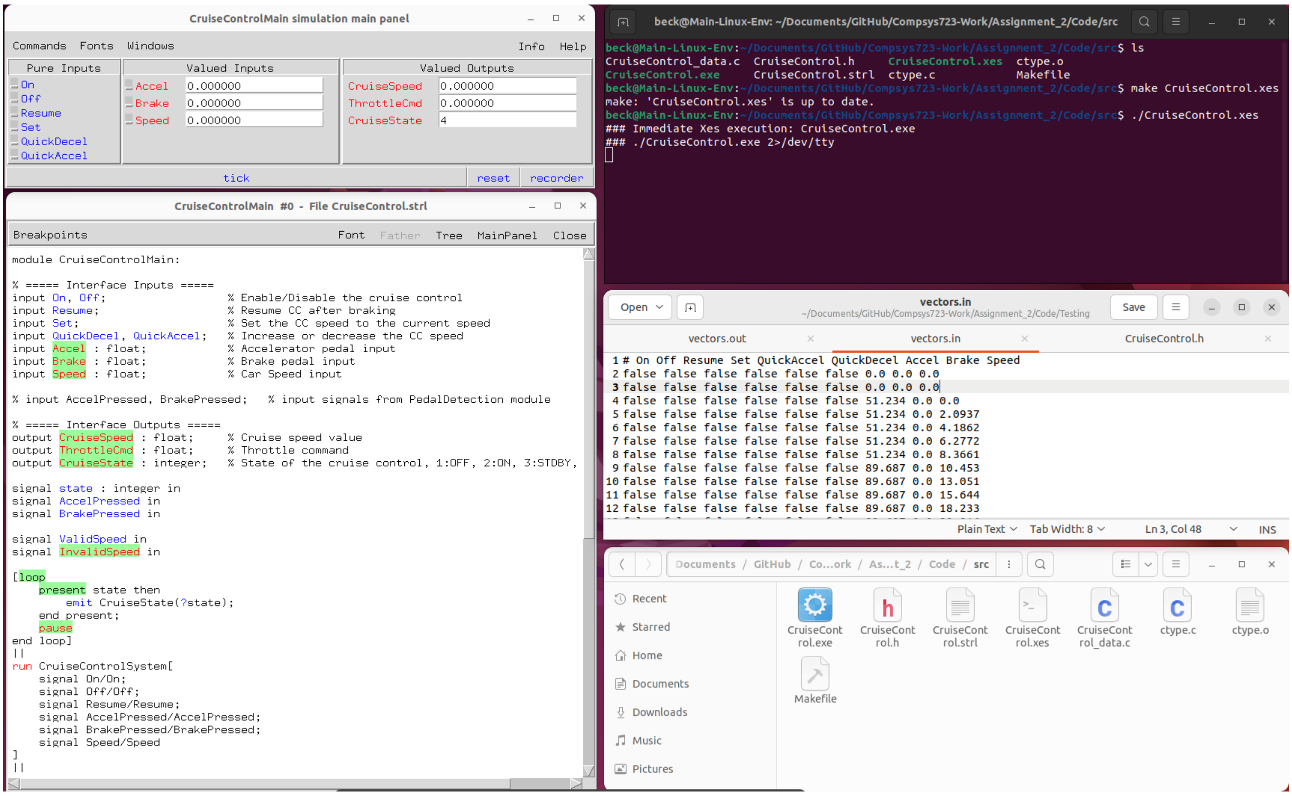
7.2. Full-size top-level context diagrams



7.3. Full-size low-level context diagram



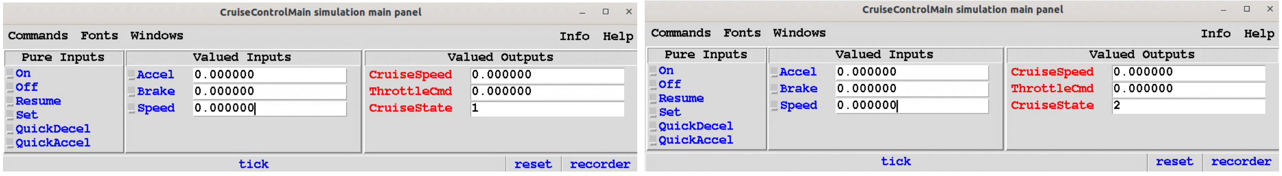
7.4. Linux testing environment



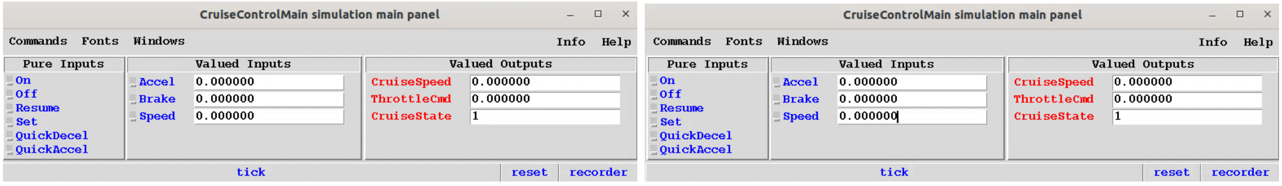
7.5. Custom testing inputs and results

Test case	Initial state	New state	Condition
1	OFF	ON	ON button
2	OFF	OFF/RESUME/SET	OFF/RESUME/SET button
3	ON	OFF	OFF button
4.1	ON	DISABLE	InvalidSpeed or AccelPressed *MinSpeed = 25.0*
4.2	ON	DISABLE	InvalidSpeed or AccelPressed *MaxSpeed = 160.0*
4.3	ON	DISABLE	InvalidSpeed or AccelPressed *AccelPressed = 4.0*
5	ON	ON	ValidSpeed and not AccelPressed
6	DISABLE	ON	ValidSpeed and not AccelPressed
7	DISABLE	STDBY	Brake
8	DISABLE	OFF	OFF button
9	STDBY	ON	Resume & ValidSpeed & not AccelPressed
10.1	STDBY	DISABLE	Resume & (Invalid or AccelPressed) *SpeedMin = 10.0*
10.2	STDBY	DISABLE	Resume & (Invalid or AccelPressed) *SpeedMax = 180.0*
10.3	STDBY	DISABLE	Resume & (Invalid or AccelPressed) *AccelPressed = 15.0*
11	STDBY	OFF	OFF button

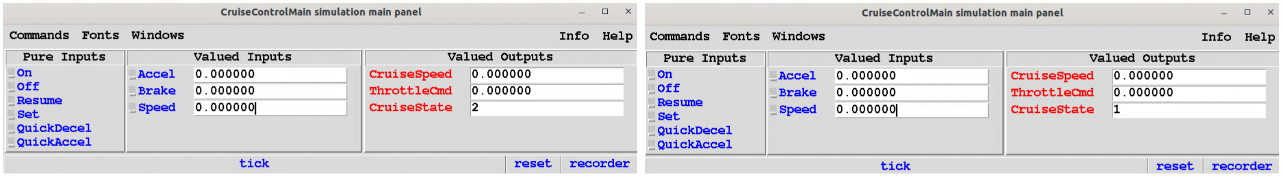
Test 1



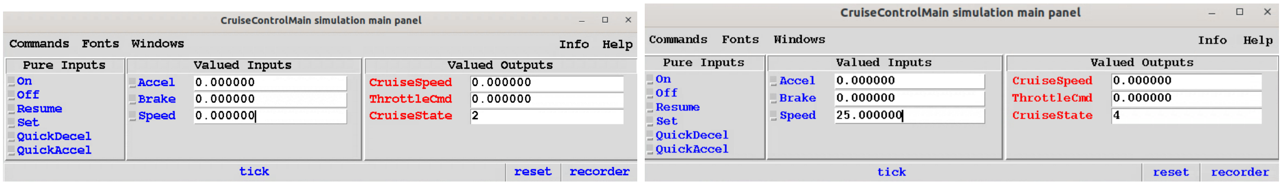
Test 2



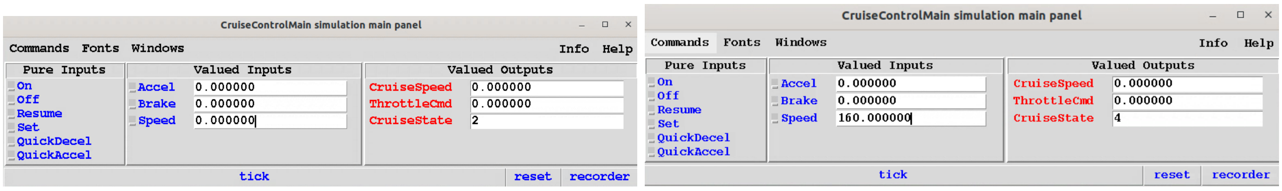
Test 3



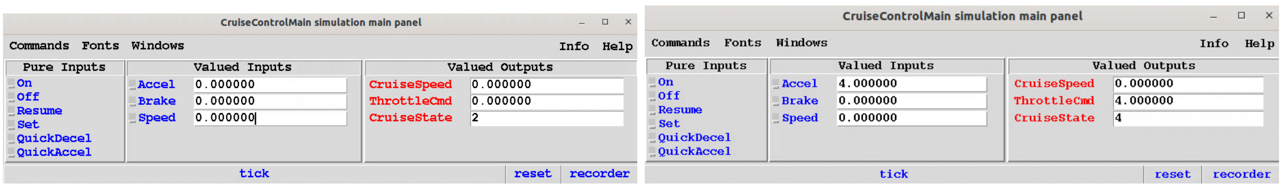
Test 4.1



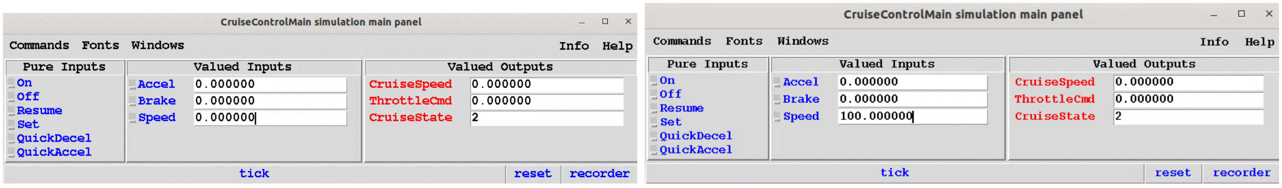
Test 4.2



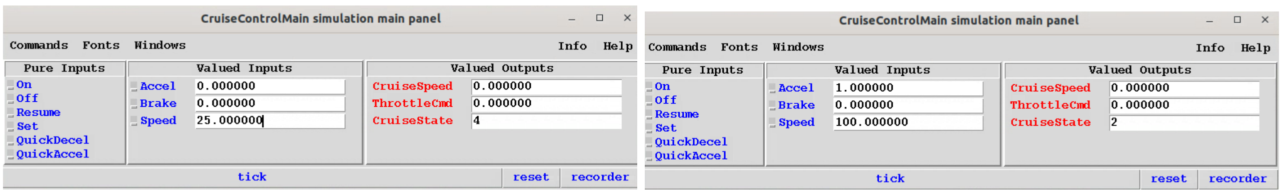
Test 4.3



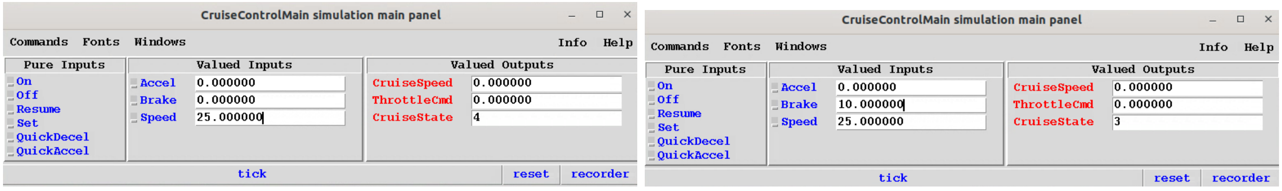
Test 5



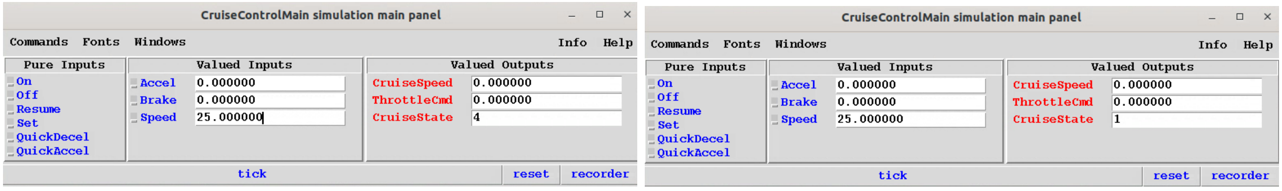
Test 6



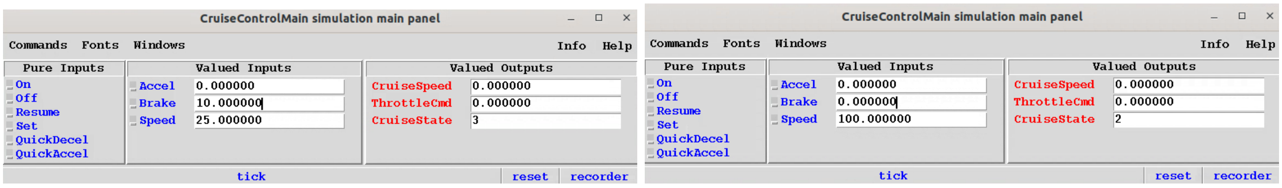
Test 7



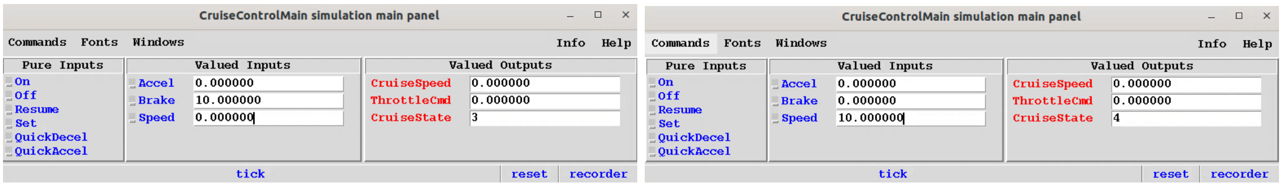
Test 8



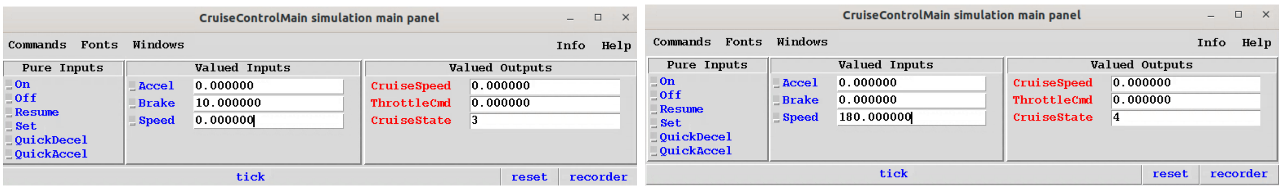
Test 9



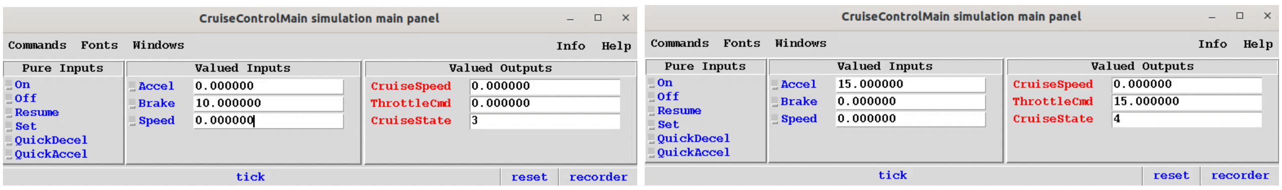
Test 10.1



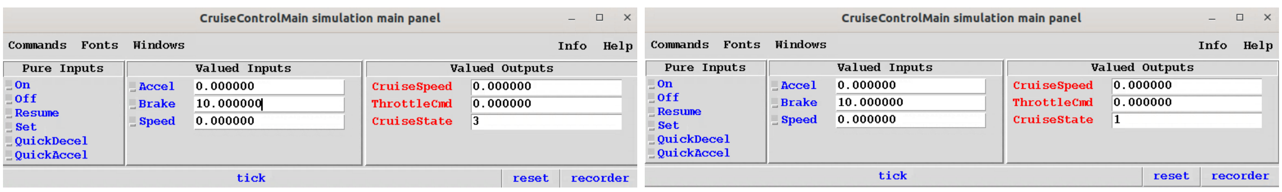
Test 10.2



Test 10.3



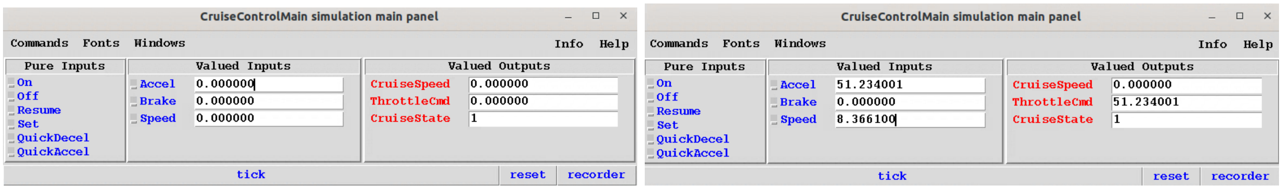
Test 11



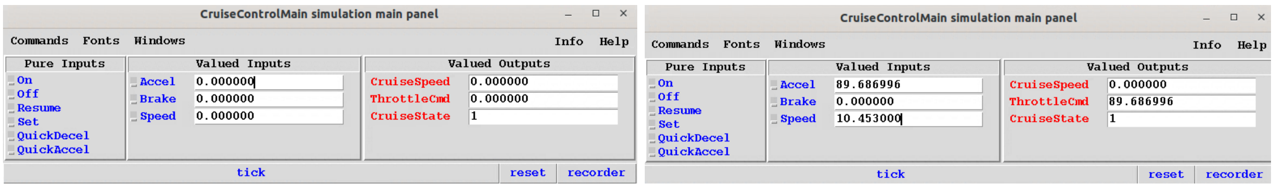
7.6. Example test cases from provided vector table

Line	On Off Resume Set QuickAccel QuickDecel Accel Brake Speed	CruiseSpeed ThrottleCmd CruiseState
8	false false false false false false 89.687 0.0 10.453	0.000000 51.234001 1
9	false false false false false false 89.687 0.0 10.453	0.000000 89.686996 1
14	false false false false false false 89.687 0.0 23.393	0.000000 89.686996 1
15	false false false false false false 89.687 0.0 25.962	0.000000 89.686996 1
20	true false false false false false 0.0 0.0 36.049	36.049000 0.000000 2

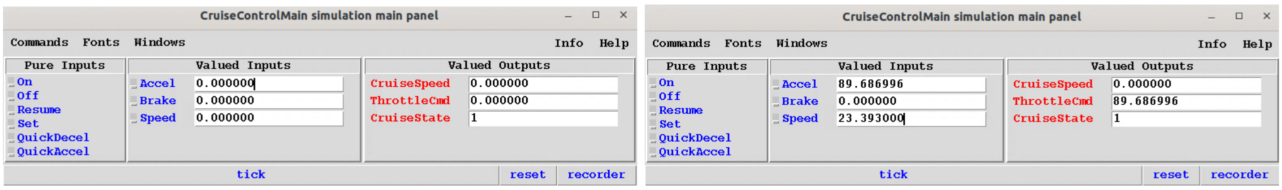
Line 8



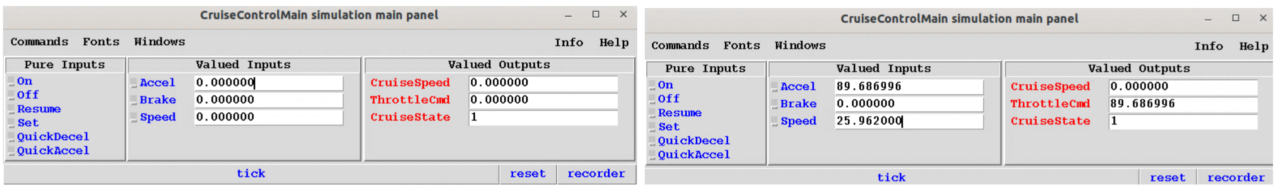
Line 9



Line 14



Line 15



Line 20 (Requires 2 ticks to complete: On → Set)