CRUISE CONTROL IN ESTEREL - DESIGN REPORT

COMPSYS 723 ASSIGNMENT 2, TEAM 12

Hao Lin and Chamith Nanayakkara

Department of Electrical and Computer Engineering
University of Auckland, Auckland, New Zealand

1. Introduction

We are implementing a vehicle cruise control system in the Esterel synchronous programming
Language. A Finite State Machine (FSM) is used to implement the primary control module and C function hosting is used to implement a PI controller..

Version control was done through github: https://github.com/uoa-hlin784/COMSYS723_Assignment_2_Team12_Cruise_Control

2. Introduction

We first decomposed the specifications given in the brief. The information is segregated into Cruise Parameters, Interface Requirements, Behavioural Requirements, Control Properties, Speed management, and Driver Operation Detection. This re-document helped us understand the system and make quick reference to key information during development by placing it in the github Repo's/submitted zip's README. Another benefit of this process is that it ensures that both members are synchronised in knowledge over the design.

2.1. System Context

The decompose specification revealed to us that the inputs are easily classified into distinct groups: System Settings, Driver operation and Speed Sensing. As shown in Figure 1, these signals reflect different demands on the cruise control system. Similar assignments were made for the three outputs *CruiseSpeed*, *CruiseState*, and *ThrottleCmd*.

We've also identified that the cruise control system can be separated into the monitor, control and actuation parts. The monitor part will be a series of conditional checks. The control part will be an FSM. The actuation part will be an PI controller informed by control signals from the control part. Displays are simple re-representation of the *CruiseSpeed*, *CruiseState* signal in the control part, whereas the *ThrottleCmd* is the output of the PI controller.

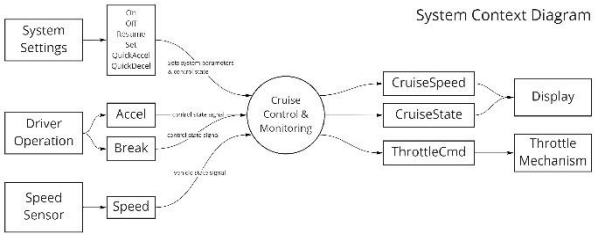

Figure 1. First step - system context decomposition.

2.2. Control-Dataflow Decomposition Diagrams

The system context diagram is further decomposed to reveal the control-dataflow relationships between responsible parts. As shown in Figure 2, input and output signals are assigned to distinct categories. Between the signals are modules of function, that perform some datapath action (control action done by cruise control).

Figure 3 reveals more intermediate signals the control part uses to command the datapath parts (actuation). For PI action, the current signal value, the target signal value and an enable command is required.

The FSM is reactive and responsive to Speed, isAccel, isBrake, On, Off, Resume and outputs/drives signal isGoingOn, CruiseState, CruiseSpeed. Shown in Figure 4, the FSM serves as the primary control unit to instruct the other datapath modules. Aside from idle, overall, the cruise function has four distinct states. In the enabled state and operation state pair, the system performs cruise control, commanding the PI controller to meet target *CruiseSpeed*. In the brake and disabled state, the system is correspondingly on standby or is disabled. Both inhibit cruise control operation with different return to the enabled state.

It is worthy of note that *Set*, *QuickAccel*, *QuickDecel* signals have no influence on state transition, as their primary role is to invoke the function to update *Speed*. Therefore their involvement in the FSM is minimal, limited to self transitions in every state, with the output being invoking and an update function to *CruiseSpeed*.

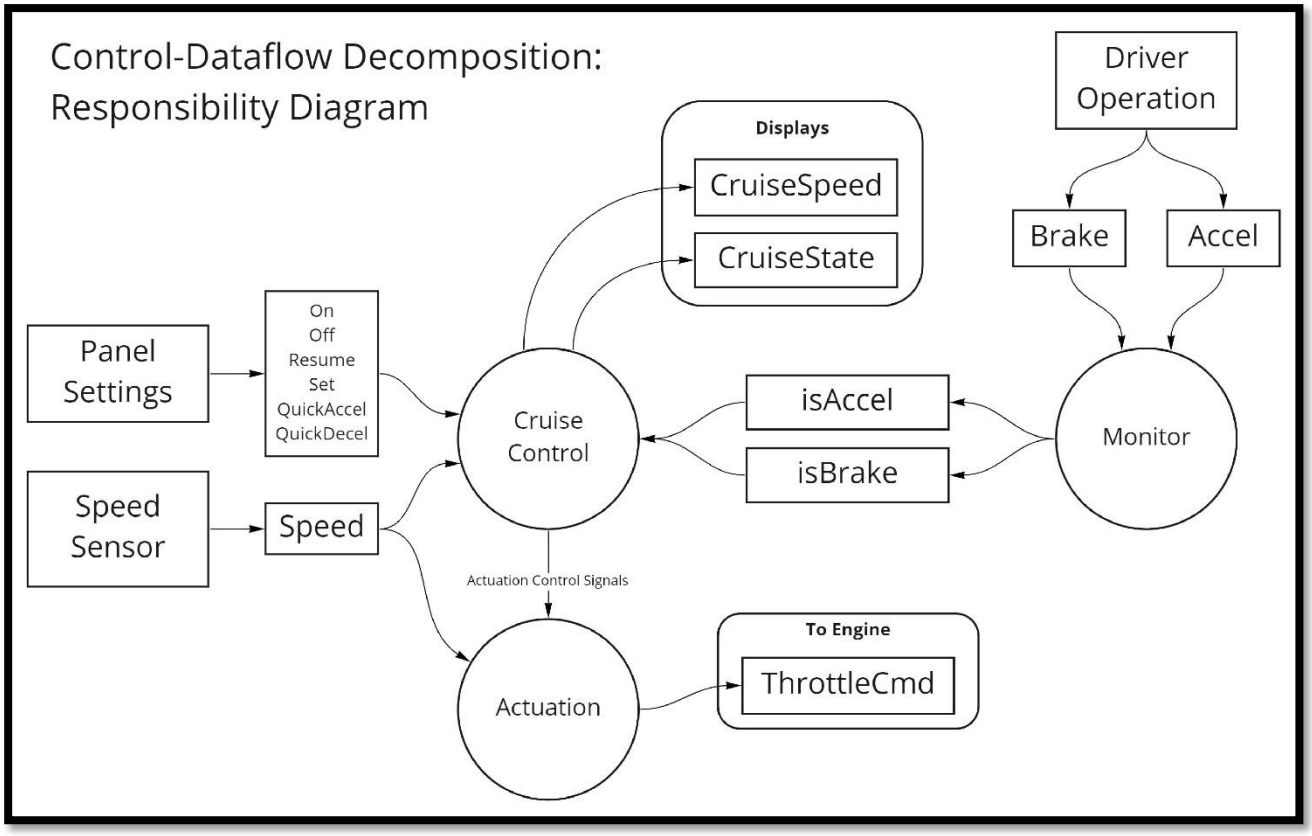

Figure 2. Second step - control-dataflow decomposition into responsibilities.

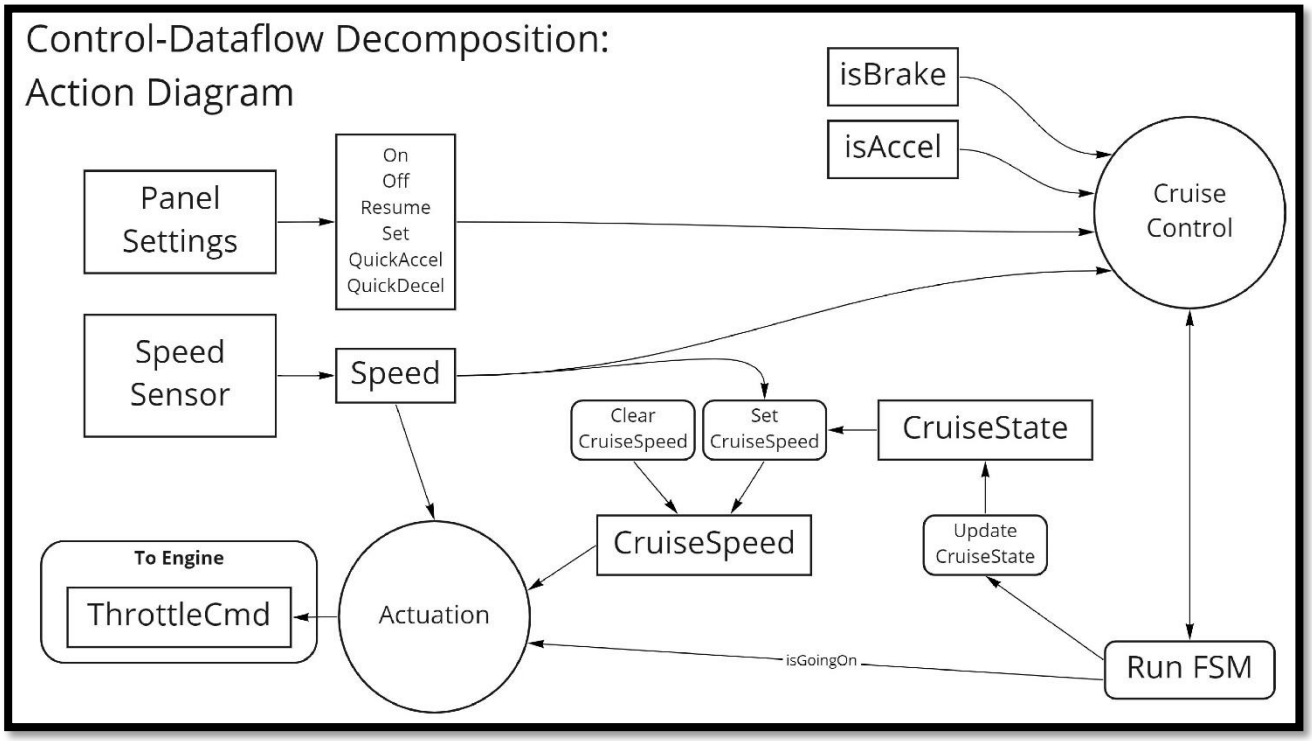

Figure 3. Action diagram identifies shared control variables and the actions done to them.

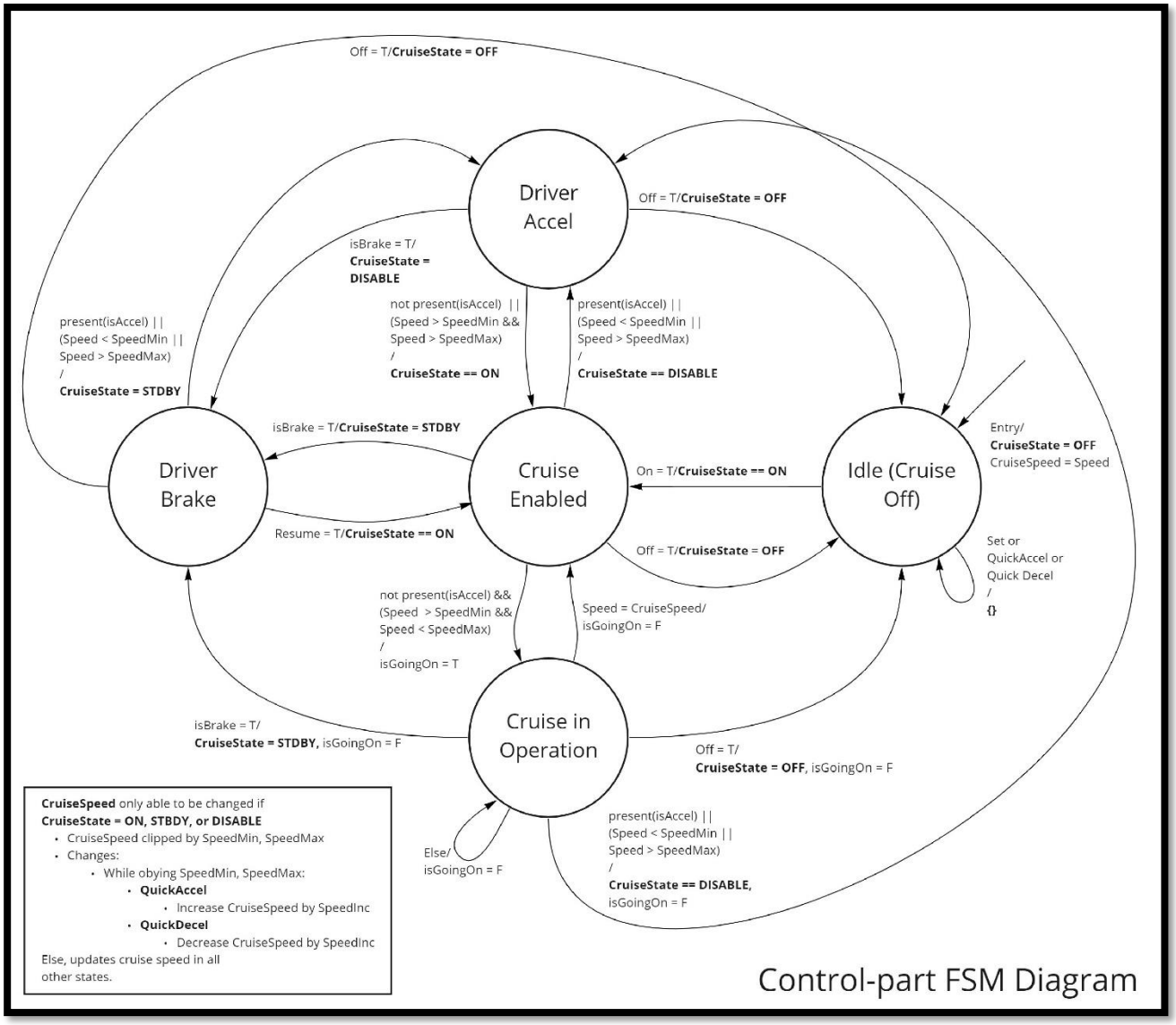

Figure 4. FSM of control part.

Figure 5 details all parameters and modules presented in a conceptual diagram of the entire system. Cruise State Update is the FSM implementation, the Cruise Speed Update is an module aggregated with functions to interact with *CruiseSpeed*, and is directly commanded by the *Set, QuickAccel*, *QuickDecel* signals.

The PI controller is implemented in C and is invoked as a host language in Esterel. All other modules that dictate control are implemented in Esterel software.

3. Design Decision Justification

3.1. Depreciated Input Monitor Modules

There is no dedicated display/output parts/modules. However, during the early stages of development, two modules for monitoring System Settings and Driver Operation were developed and abandoned. Initially, these modules de-coupled responsibility of implementation, allowing the potential to rewire the handling of input signals without modifying the control part.

For example, a reason for having the *settings_monitor* module is to be able to encapsulate internal signal operation. So if the panel settings were to change, only modifications to the *settings_monitor* is required.

This choice was proven to be redundant for a project of this scope and timeframe, as direct wiring and conditional handling within one module were easy and straightforward to implement.

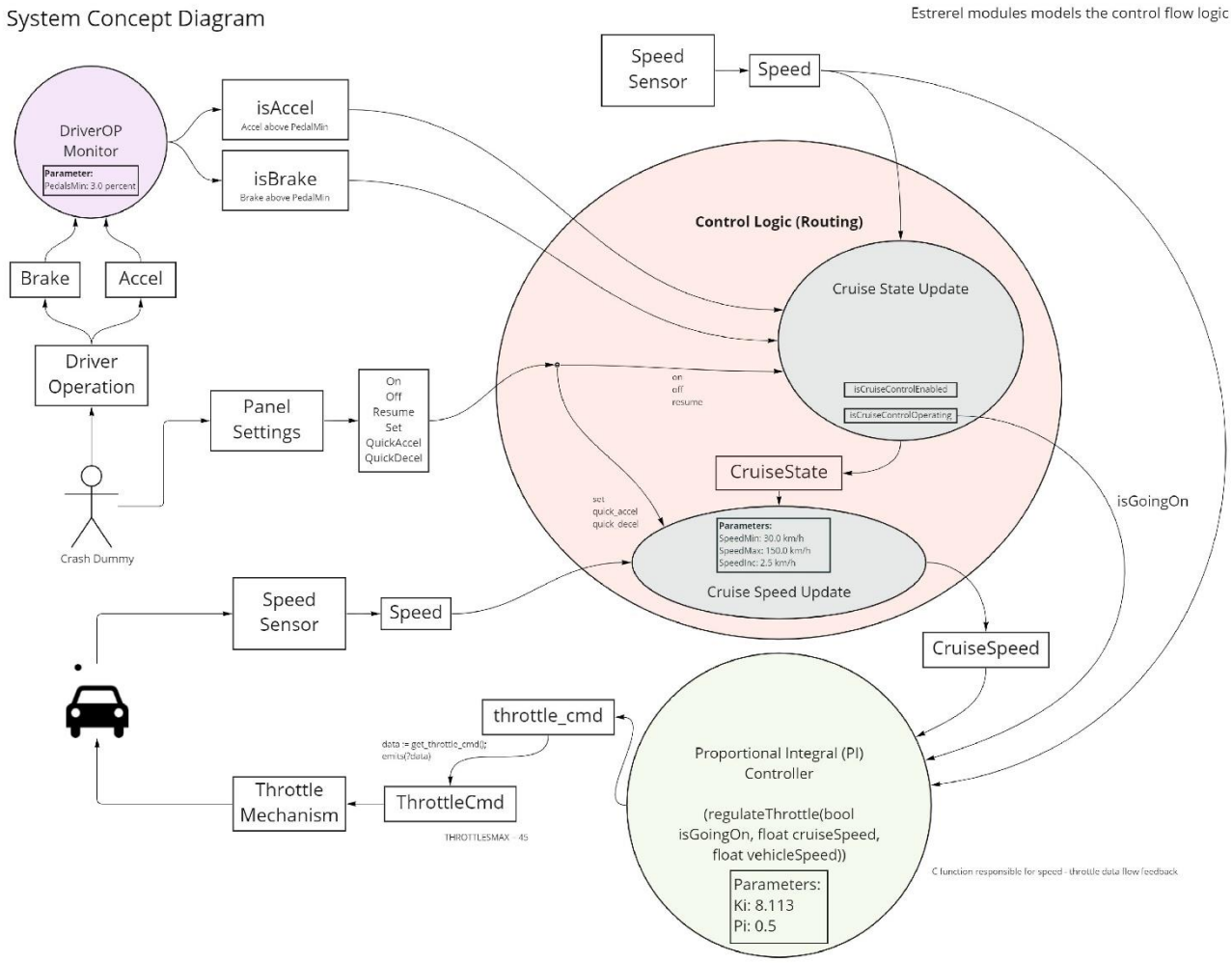

Figure 5. Final system concept diagram of cruise control program showing distinct responsibilities.

3.2. Software Structure

The Esterel software consists of three modules: *cruise_state_updater*, *cruise_speed_updater*, and *PI_controller* that runs in parallel to each other, as per the separation of responsibilities shown in Figure 5. The PI controller functions in *cruiseregulation.c* is wrapped in the *PI_controller* module, which handles the passed signal values.

Each module essentially presents a *cruise_state* check and a series of presence checks on global signal emission to control the behaviour of each module and its signal emission.

3.2.1. Signal Assumptions

We've interpreted the signals *Brake*, *Off* as hard aborts. Both signals should appropriately take precedence: Braking is a safety critical manoeuvre that demands immediate changes to vehicle motion. Turning the cruise control off is not safety critical but is unanimous in all states. We've refrained from a statechart representation and opted for a standard FSM.

Figure 7 gives a demonstration over the software's operation.

4. Organisation and Timeline

4.1. Work Partition

Work partitioning is done by partitioning the system responsibilities into modules shown from Figure 5, where modules are partitioned to be completed by each member.

Task Contribution (0 to 5)					
	Monitor (depreciated)	PI Controller	Speed Update	State Update	Report
Hao	5	1	0	3	5
Chamith	0	5	5	5	3

Figure 6. Project Contribution chart (for 0 to 5 for each task)

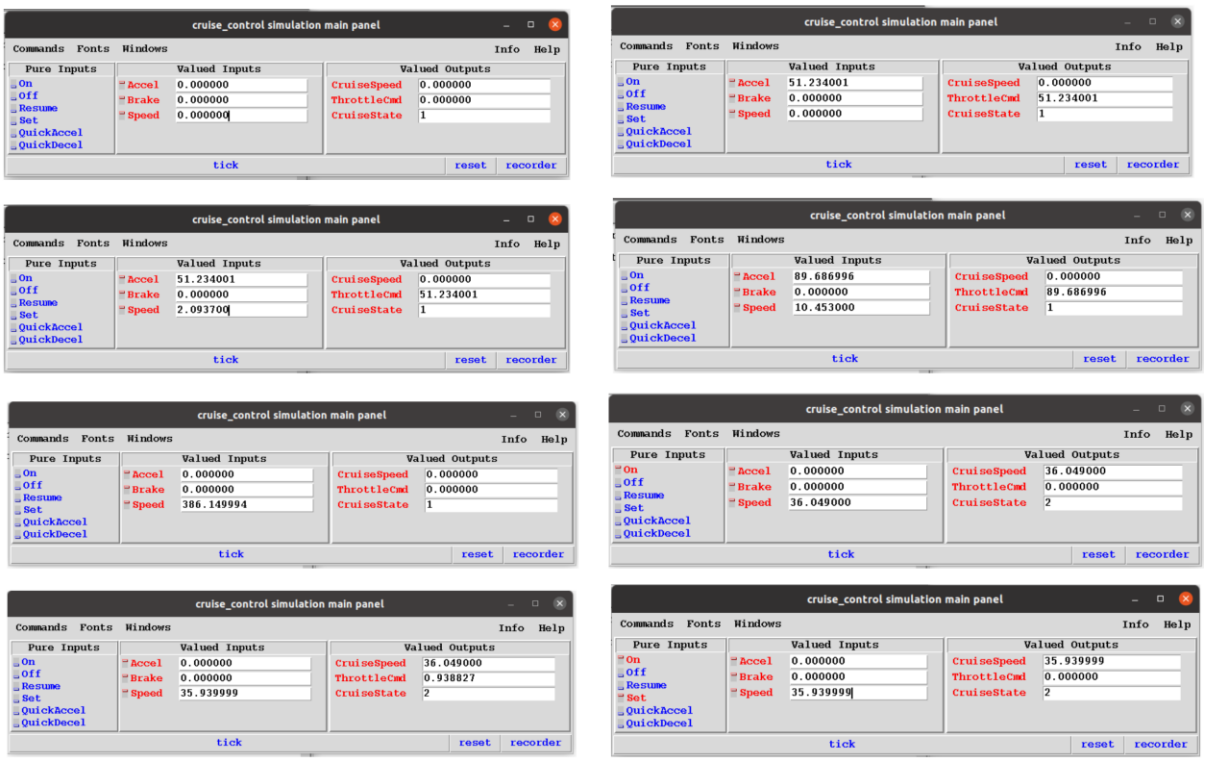

Figure 7. Sequence of results checked in simulation, where each screenshot is a tick transition.

4.2. Gantt

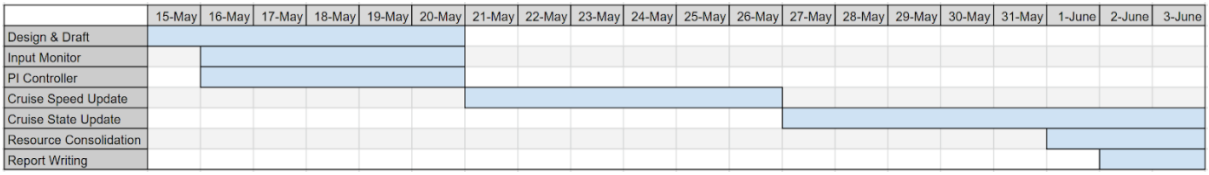

Figure 8. The timeline allocated for each task in this project.

5. Conclusion

To build a cruise control system, we solidified our scope with a context diagram to identify the relationships between external entities and the software system. We performed control-dataflow decomposition to formally segregate tasks, function and responsibility of the software system. Dividing the tasks among two members we created a concurrent, synchronously cruise control system with safety guaranteed by the Estereal language.