StateCharts Modeling, simulation, synthesis and testing

Ken Bauwens 20143225
and Baturay Ofluoglu 20174797

December 13, 2017

1 Introduction

In this report, we will first give a small introduction. We will then go over each component and explain how we implemented it. Finally we will draw a short conclusion about statesharts.

1.1 Who did What

Together

- A train is either standing still (either in a station, or outside of it), driving, or cruising at full speed.
- After opening the doors, they must remain open for at least 5 seconds to allow people to unboard and board. Pressing the 'close' button before that will just be ignored.
- When reaching the maximally allowed speed (120 kilometers per hour), further positive acceleration will be ignored.
- Likewise, when standing still, further (negative) acceleration should be ignored. Make sure that in all cases, it is impossible to go backwards!
- The train can go in emergency brake mode when dangerous situations occur. In this mode, the interface no longer listens to the driver, and the train will decelerate as hard as possible (i.e., -1), before turning itself off when speed has reached 0.
- You will have to manually call the update operation to update the current speed and GUI. Do this every 0.02 seconds.

Baturay Ofluoglu

- The train can only open its doors when it is standing still in a station. Pressing the 'open' button at any other time will just be ignored.
- The train can only open its doors once in every station: when the doors have been closed, they cannot be opened again until the next station is reached.
- Standing still does not necessarily imply that you are at a station, so the doors should remain closed if the train is outside of a station.
- When driving in a train station at a speed above 20 kilometers per hour, the train automatically goes into emergency brake mode. This holds at all points in time while in a station.
- When a train has opened its doors, it ignores all acceleration requests until its doors have been closed again. This to prevent the train from leaving with its doors open.

Ken Bauwens

- After an emergency brake, when the speed has reached 0, the interface becomes responsive again after a cooldown period of 5 seconds. That is, speed must remain equal to 0 for 5 seconds before the interface can be used again.
- When passing a red light, the train automatically goes into emergency brake mode.
- When passing a yellow light, the speed must be lower than or equal to 50 kilometers per hour. If it is higher, an emergency brake should be triggered.
- After passing a yellow light, the speed must be limited to 50 kilometers per hour until the next light is reached. Only when a green light is reached, can the train drive at the maximum speed again.
- The driver needs to press the "dead man's button" every 30 seconds. If the button is not pressed within 5 seconds of the prompt, the train goes into emergency brake mode. Pressing the button before the prompt occurs will have no effect.
- Pressing the pause button will stop the simulation completely. Pressing continue afterwards will make it resume.

1.2 Time spent

On the first part we spent about 6h. On the individual parts we spent a total of 5h (Ken: 2h, Baturay: 3h) This brings the total to 11h, which does not include the writing of the report.

1.3 General methodology

To model the train, we split up the functionalities in orthogonal components. These components can be seen in figure 6. Each component has its own meaning, which can be derived from the component name.

2 requirements

2.1 1st Requirement

Requirement: "A train is either standing still (either in a station, or outside of it), driving, or cruising at full speed."

Solution: To accomplish this requirement, we only have to link an accel event to *self.acceleration = a*. To update the position of the train, see requirement 17. The updating of the acceleration can happen in multiple states/transitions, because of the restrictions discussed later.

2.2 2th Requirement

Requirement: "The train can only open its doors when it is standing still in a station. Pressing the 'open' button at any other time will just be ignored."

Solution: This requirement is solved in the *Doors* orthogonal component.
At the initial state the doors are closed which means the doors are at *Closed* state. In order to open the doors, the train needs to be at the station. Therefore, we first observe whether train is at the station by checking *enter* event. Then, the train comes to *InStation* state. In this state it has two options which are either opening door or continue directly. Thus, it can only open the door when the train at *InStation* state.

2.3 3th Requirement

Requirement: "The train can only open its doors once in every station: when the doors have been closed, they cannot be opened again until the next station is reached."

Solution: In each arrival to the train station *enter* event called once. Therefore, referring to requirement 3, a transition from *closed* state to *InStation* would occur only once. Also, *DoorsOpened* state is dependent to *InStation* state. Therefore, the train can open and close its doors only once in the train station.

2.4 4th Requirement

Requirement: "After opening the doors, they must remain open for at least 5 seconds to allow people to unboard and board. Pressing the 'close' button before that will just be ignored."

Solution: This requirement is achieved with the timed transition *Doorscan-Close* which will fire after 5.0 seconds. Because this is the only way to proceed to the *Open* state, which is the only state where the doors can be closed again, the doors will remain open for 5.0 seconds.

2.5 5th Requirement

Requirement: "Standing still does not necessarily imply that you are at a station, so the doors should remain closed if the train is outside of a station."

Solution: To satisfy this requirement, *openDoors()* transition only starts from *InStation* state. Train means during other states, the doors cannot be opened. It needs to be at the *InStation* state which symbolize the train enters to the train station.

2.6 6th Requirement

Requirement: "When reaching the maximally allowed speed (120 kilometers per hour), further positive acceleration will be ignored."

Solution: This requirement is satisfied by state *Maxspeed.* Transition *Maxspeed* has a condition *self.velocity* >= *120.* When in state *Maxspeed*, the only action that will result in a transition is accel with acceleration < 0, preventing users from going faster. Notice that because of the segmented nature of time in simulations, it is possible to shortly go faster than 120 km/h. This is why the condition is >= instead of ==. this does not however invalidate the simulation, as even in real life situations it is nearly impossible to achieve an exact max speed.
A consequence of taking the *Maxspeed* transition will be to set velocity to 120 to fix this offset. It wil also set acceleration to 0, for obvious reasons.

2.7 7th Requirement

Requirement: "Likewise, when standing still, further (negative) acceleration should be ignored. Make sure that in all cases, it is impossible to go backwards!"

Solution: Same as **Requirement 6**, but with *Minspeed* and checks for velocity <= 0.

2.8 8th Requirement

Requirement: "The train can go in emergency brake mode when dangerous situations occur. In this mode, the interface no longer listens to the driver, and the train will decelerate as hard as possible (i.e., -1), before turning itself off when speed has reached 0."

Solution: To achieve this, we implemented a transition from the composite group Accelerator to an "error"-state outside of this group. The transition to this state will fire on a *emergencyBrake*-event. To return to the normal execution, an event *resume* has to be fired. We implemented this construction for both the group Accelerator aswell as the group Doors, to completely take away

control from the user during an emergency.
When returning to the Accelerator composite, we return to the correct state by using a *historystate*. This is not necessary for doors because an emergency cannot happen while the doors are opened, meaning the start state is always the correct state to return to. This is a result of the other requirements.
To fire the *emergencyBrake*-event, we have another orthogonal component *emergency*. This component contains the emergency-conditions, and fires the event on emergency. It will also set acceleration to -1 when firing this event.

2.9 9th Requirement

Requirement: "After an emergency brake, when the speed has reached 0, the interface becomes responsive again after a cooldown period of 5 seconds. That is, speed must remain equal to 0 for 5 seconds before the interface can be used again."

Solution: Using the *emergency*-component from requirement 8, we extend this with a *slowdown* state and a *cooldown* state. The *slowdown* state will check the velocity every "eventloop". As soon as velocity 0 is reached, it will transition to the *cooldown* state. In this state, the transition *resumeNormal* will fire after 5.0 seconds. This transition will also fire the *resume*-event to return to the regular execution.

2.10 10th Requirement

Requirement: "When driving in a train station at a speed above 20 kilometers per hour, the train automatically goes into emergency brake mode. This holds at all points in time while in a station."

Solution: For this requirement, we also used Emergency break solution. When the train is at *enter* state, the velocity should be lower than 20 km/h. We add *self.velocity >20* constraint and when it passes the limit speed then *emergencyBreak* event will be raised.

2.11 11th Requirement

Requirement: "When passing a red light, the train automatically goes into emergency brake mode."

Solution: Using the previously explained Emergency brake solutions, we can easily add this requirement by creating a conditional transition in the *emergency* component. This transition will fire on a *red_light* event, entering the emergencymode.

2.12 12th Requirement

Requirement: "When passing a yellow light, the speed must be lower than or equal to 50 kilometers per hour. If it is higher, an emergency brake should be triggered."

Solution: Same as Requirement 11, except with a *yellow_light* event and a condition *self.velocity >50*.

2.13 13th Requirement

Requirement: "After passing a yellow light, the speed must be limited to 50 kilometers per hour until the next light is reached. Only when a green light is reached, can the train drive at the maximum speed again."

Solution: To fulfill this requirement, we mirror our accelerate setup to a limitedAccelerate setup. When in state *accelerate*, a *yellow_light* will transition to the *limitedSpeed* state, which does the same as the regular setup except with an upperbound on self.velocity of 50. We return to the *accelerate* state when receiving a *green_light* event in states *limitedSpeed* and *limitedMaxSpeed*.

2.14 14th Requirement

Requirement: "The driver needs to press the "dead man's button" every 30 seconds. If the button is not pressed within 5 seconds of the prompt, the train goes into emergency brake mode. Pressing the button before the prompt occurs will have no effect."

Solution: This requirement is modeled in the *Deadman* component. The *Timetopoll* transition fires every 30.0 seconds. A *alive* event fires the *awake* transition, indicating everything is fine. After 5.0 seconds in the *Polling* state, transition *driverSleeps* will fire and it will raise a "DeadDriver" event, which will start the emergency mode as described earlier.

2.15 15th Requirement

Requirement: "When a train has opened its doors, it ignores all acceleration requests until its doors have been closed again. This to prevent the train from leaving with its doors open."

Solution: To satisfy this requirement, we raised *allowedToOpen* event during *openDoors()* transition because this transition is only active when the doors are allowed to open based on other requirements. We also raised *closeAllowed* event during *closeDoors()* transition for the same reason. So, when a train stops at the station, it can be either at state *MinSpeed* or *LimitedMinSpeed* depending on the last light it get (green or yellow). To deactivate acceleration system, when the doors are opened *openDoors* transition is activated and it reaches *DoorsOpen* state. In this state, the train cannot accelerate. If the doors are closed then *DoorsClosed* transition activate again *MinSpeed* or *LimitedMinSpeed* state to enable acceleration.

2.16 16th Requirement

Requirement: "Pressing the pause button will stop the simulation completely. Pressing continue afterwards will make it resume."

Solution: This is easily implemented with the *pause*-transition exiting the "TrainSystem" component. This transition fires on the *pause* event. To return, the transition *continue* fires on a *continue* event. We use a history state to remember the setup on pausing.

2.17 17th Requirement

Requirement: "You will have to manually call the update operation to update the current speed and GUI. Do this every 0.02 seconds."

Solution: This requirement is modeled with the component *update*, which calls self.updateState() every 0.02 seconds using a timed transition.

3 Conclusion

We noticed that statecharts are useful to model concurrent systems.
The decomposition of the problem in components also results in a fairly object oriented breakdown, which can easily be translated in actual code.

Appendices

Figure 1: Accelerator component

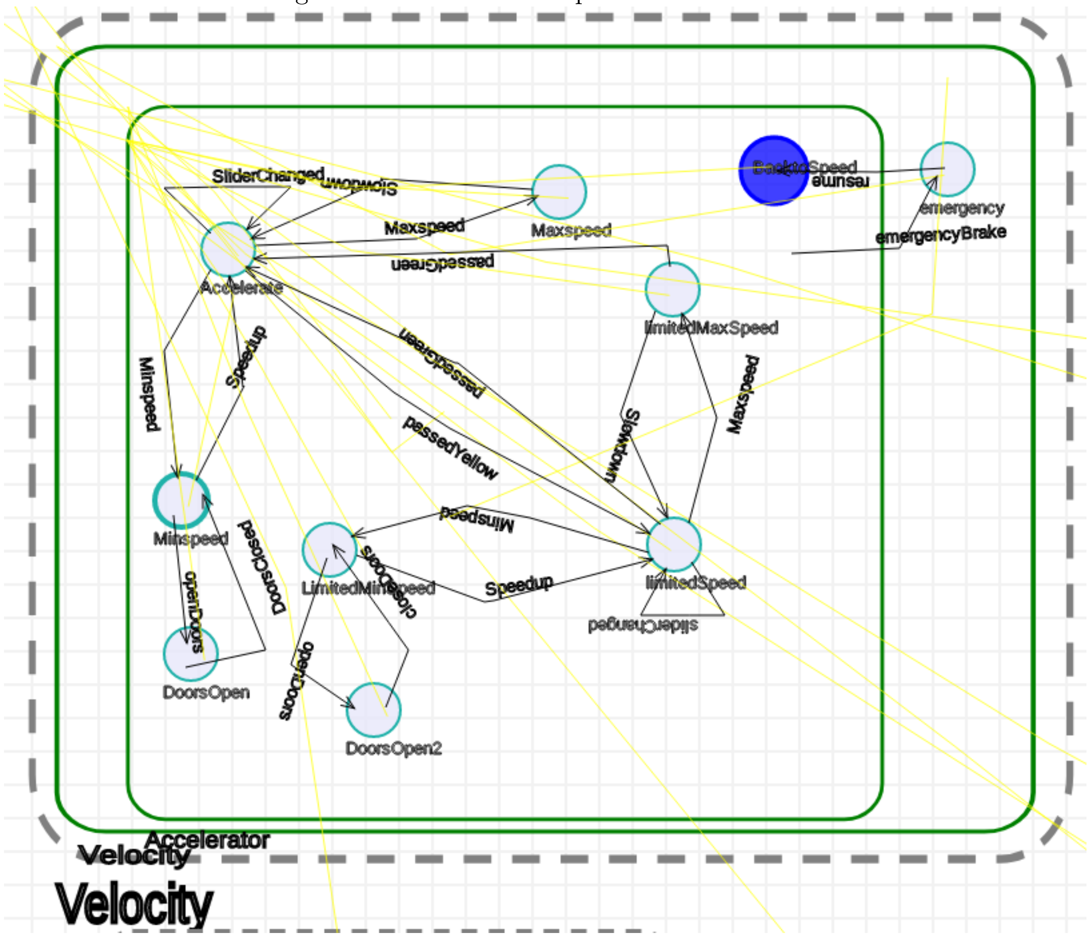


Figure 2: Doors component

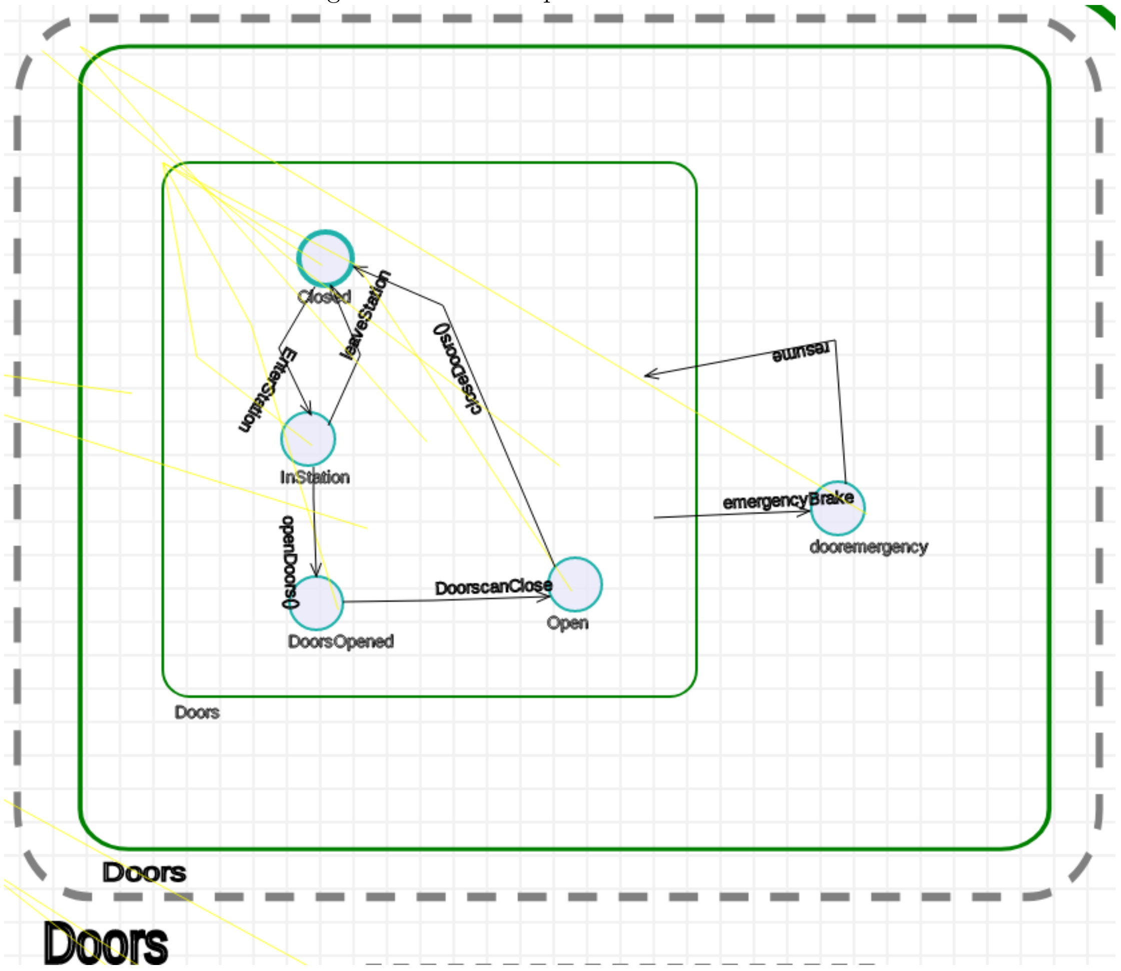


Figure 3: Emergency component

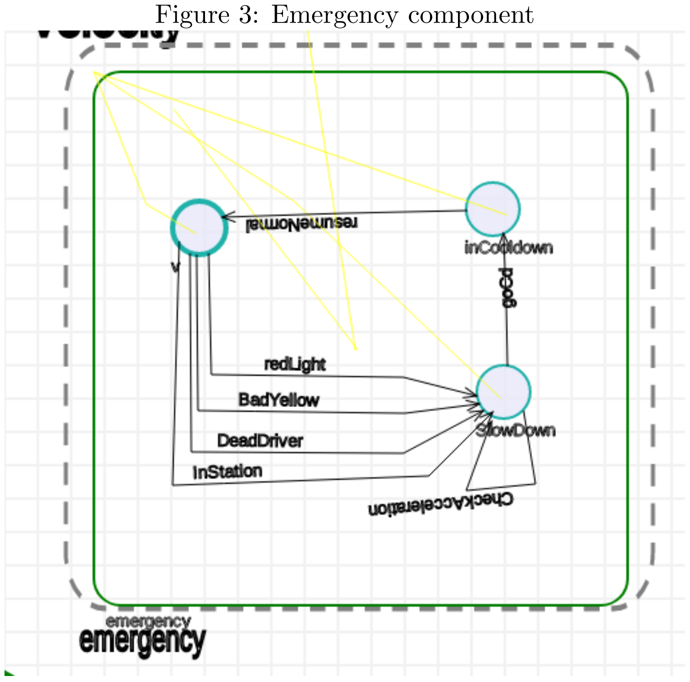


Figure 4: Deadman component

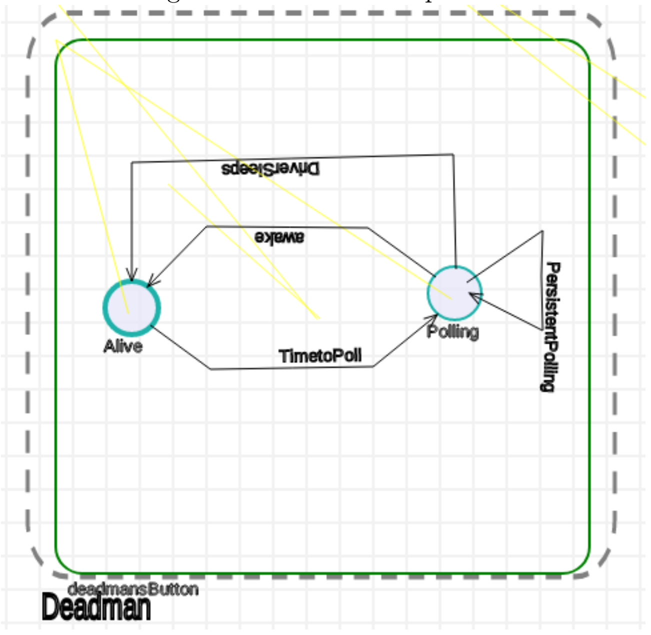


Figure 5: Update component

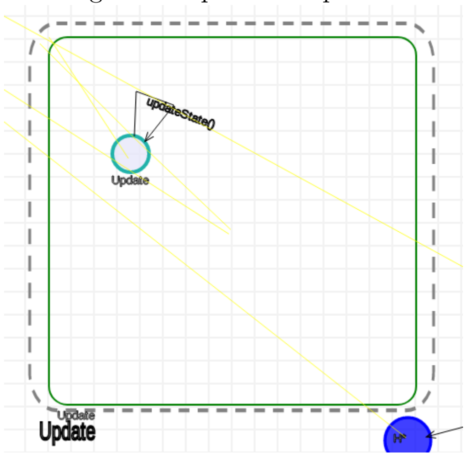
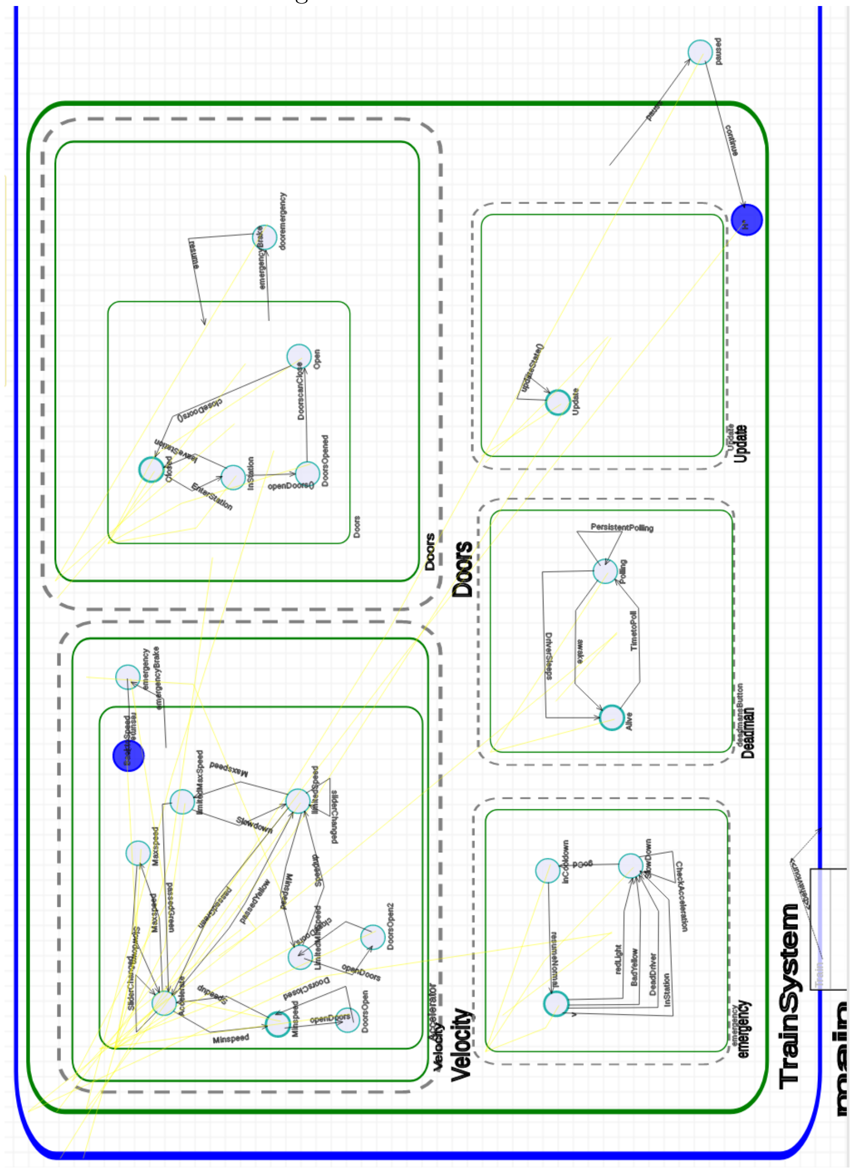


Figure 6: Entire statechart


main
TrainSystem
Velocity
Doors
Update
Deadman
emergency
paused
continue
Accelerator
Emergency
emergencyBrake
LimitedMaxSpeed
MaxSpeed
LimitedSpeed
Minspeed
Slowdown
sliderChanged
SpeedUp
passedYellow
passedGreen
openDoors
DoorsOpen
DoorsOpen2
DoorsClosed
Closed
InStation
DoorsOpened
DoorscanClose
Open
EnterStation
leaveStation
openDoors()
closeDoors()
emergencyBrake
dooremergency
resume
Update
updateState()
Polling
PersistentPolling
Alive
awake
DriverSleeps
TimetoPoll
deadmanButton
InCooldown
SlowDown
CheckAcceleration
resumeNormal
redLight
BadYellow
DeadDriver
InStation