Self-driving Car

Taylor Courtney

Edinburgh Napier University, Edinburgh EH10 5DT, United Kingdom
40398643@live.napier.ac.uk

Abstract. This paper specifies the implementation of a self-driving car. The introduction will specify the scope of the problem, followed by a high-level view of the system developed. With the high-level knowledge of the system, the report will then take a closer look at the procedures and functions of the system at a lower level. The lower-level view covers many complexities; therefore, proof of consistency is conducted in the form of sequent calculus upon an important feature. Extensions made within the system are listed, which exist alongside the problem's initial specification. Further extensions that better simulate the complexity of a self-driving car are discussed upon conclusion.

Keywords: Ada, SPARK, Self-driving.

1 Introduction

The problem of a self-driving car entails the numerous scenarios a car can encounter. In every scenario, some conditions must be considered when deciding how to react to situations encountered.

The system implemented is inherently constrained due to the complexity of driving. For a self-driving car to make sensible decisions, there are many variables that it must consider quite often.

Based on the problem, the car will continue to perform actions, such as driving, while the driver can give the car instructions during the journey. The system will allocate the driver's controls a thread to allow asynchronous instructions to the car.

The system implemented allows the user to tell the car when to start driving, and the car will begin the journey. Throughout the said journey, the car constantly monitors multiple variables: battery level, current speed, speed limit, distance to destination, and others external to the system, such as junctions, obstructions, and requests from the driver (more specifically, if the driver requests that the car pulls over).

When the car arrives at its destination, it will automatically park itself for the driver and return to a neutral state. The driver may either exit or give further instructions.

2 Controller Structure

The system has two main threads: the controller and self-drive. The controller thread listens for inputs made by the driver and handles them according to the current state of

the car's self-drive thread. The self-drive thread makes autonomous effects based on scenarios generated by probability. These two threads run parallel in case the driver wants to give the car instructions when it is in motion via self-drive. A third thread also runs the car's diagnostics mode in parallel. However, it only performs any actions when the driver successfully enables the car's Diagnostics Mode.

The threads analyse and control the car's current state and environment via global variables. The system has two major types that the global variables form; more specifically, two records.

2.1 Car Record

The car record consists of variables relevant to the car on a system level.

CarType			
Attributes	**Type**	**Derivation**	**Function**
engineOn	Boolean	N/A	It acts as the car's ignition that the driver can enable/disable.
diagnosticsOn	Boolean	N/A	True when the driver enables diagnostics.
parkRequested	Boolean	N/A	True when the driver tries to change the car's gear to park while it is in motion.
gear	CarGear	Enumeration	The gears available in the system are drive, reverse, and parked.
speed	MilesPerHour	Integer range (-3 .. 20)	Changes depend on the current environment's condition. It becomes negative when the car is in reverse to signify moving backwards.
battery	BatteryLevel	Integer range (0 .. 100)	Decrements when the car is driving.
forceNeedsCharged	Boolean	N/A	True when the battery either reaches a minimum level or falls below a level that is too low to complete the rest of the journey
braking	Boolean	N/A	True when the car needs to slow down: when reaching a turn or destination.

- **engineOn** - the vehicle cannot drive without it being true. The car's battery will charge while false.
- **diagnosticsOn** – can only be enabled when the engine is off. The car becomes static, and all of the driver's controls are disabled until this procedure finishes.
- **parkRequested** - the driver has let the self-driving system know they want it to stop. The self-driving system will slow the car down and park.

- **gear** - can only be changed when the car is immobile, except when its speed is negative (meaning it is moving backwards), then the gear can be changed. This allows the car to switch to drive and move forward if the car is rolling back.
- **speed** - ranged because the car should not exceed the maximum speed limit; maximum 20 for demonstration purposes so it does not take too long to slow down and minimum -3 as the car should not reverse too fast for controllability.
- **battery** – constantly decrements as long as the car is driving or in diagnostics. It can be increased by turning the car (engine) off, essentially allowing the car to charge.
- **forceNeedsCharged** - the car will slow to a stop and immobilise until the battery is charged.

2.2 Environment Record

The environment record contains variables that describe the car's environment. The driver cannot control anything about the environment.

WorldType			
Attributes	**Type**	**Derivation**	**Function**
curStreetSpeedLimit	MilesPer-Hour	Integer range (-3 .. 20)	Specifies the speed the car should not exceed.
destinationReached	Boolean	N/A	It becomes true when the required number of turns is reached.
obstructionPresent	Boolean	N/A	It has a chance of becoming true (presenting an obstruction) during the journey.
turnIncoming	Boolean	N/A	It also has a random chance of becoming true during the journey.
numTurnsUntilDestnation	WorldTurns	Integer range (0 .. 3)	Simulates the length of a journey. It is ranged for the same reason the **speed** limit is not high: to adequately demonstrate the system.
numTurnsTaken	WorldTurns	Integer range (0 .. 3)	The same type as **numTurnsUntilDestination** as the number of turns taken by the car will never be greater than it.

- **curStreetSpeedLimit** - a random integer that changes every time the car turns onto a different street or initialises a new route. It has the same range of integers as the car's **speed,** as they should both have the same maximum.
- **destination**R**eached** - once this is true, the car is on the same street as the destination and has a random chance of slowing down and parking at the destination.

- **obstructionPresent** – when true, the car will perform an emergency stop, reverse back from the obstruction, drive around it and continue the route.
- **turnIncoming** – when true, the car will slow down to prepare for the turn. Once the car's **speed** reaches 0 (to give way first), the turn will be performed, and the car will accelerate again.
- **numTurnsUntilDestination** - static during routes. Initialised as a random value when the driver begins a route and is only changed when the car reaches a destination and starts a new route. The limit is imposed to show, in good time, the car can park.
- **numTurnsTaken** - incremented after every turn made by the self-driving system. When it is equal to **numTurnsUntilDestination**, the route is complete.

Because the destination arrival, upcoming turn, and obstruction Booleans have some probability of becoming true, an enumerated type specifies this list of scenarios. The type **WorldScenario** contains the enumerated scenarios: arrival at the destination, incoming turn, obstruction, and a value signifying no scenario. One of these scenarios may be returned by a scenario generation function. If anything besides the numerator for no scenario is returned, the self-drive system must accordingly handle the scenario. No scenario has the highest probability of being returned to allow the car to drive more.

The driver should actively receive information about the car's activities. There exists an enumerated type **WorldMessage,** that lists the different messages that can be given by the car. Messages come from the system's state: low battery warning, charge required enforcement, destination arrival, and battery and speed.

2.3 Random Integer Generator

The definition of the random number generator is not large as it does not require a significant definition. The integer generator is a separate package as it requires a type of number and range of numbers to be generated. The specification in the system returns a random integer within the range, type **RandRange,** one to one hundred. The return value can be less than one hundred as the function developed in the system will only return a number less than or equal to the parameter given to the generator.

3 Descriptions of Procedures and Functions

There are three constant values in the scope:

- **MINIMUM_BATTERY** - specifies the minimum battery permitted before enforcing that the car needs to be charged. This is the level checked for by the low battery warnings and the charge enforcement procedure.
- **WARNING_INTERMISSION** - specifies the intermission of each low battery warning; the amount of per cent to go down before issuing another warning.
- **SPEED_LIMIT_STEP_FACTOR** – specifies the multiples of speed limits that are available. For example, the system currently uses ten, so the speed limits generated will be multiples of ten.

3.1 Discharge Battery Procedure (Gold proof)

Decrements the battery level.

- Precondition(s):
  - The car must be on and not be in the parked gear, or
  - Parked and off but in diagnostics mode.
  - It should also not be at 0% battery.
- Postcondition(s): the car's battery level will equal one less than before invocation.

3.2 Warn of Low Battery Function (Gold proof)

Determines whether the driver should be warned of a low battery by checking for the required conditions.

- Precondition(s): N/A because the driver should always be able to be warned of a low battery, even during diagnostics with the engine off.
- Postcondition(s):
  - Will return true if the battery level is:
    - Less than or equal to the minimum,
    - Has a zero modulus of the warning intermission constant (to prevent spamming the driver with warnings), and
    - It does not equal zero (as a unique message is displayed in this instance).
  - Returns false is any of the previously mentioned conditions are not true.

3.3 Engine Switch Procedure (Gold proof)

It is invoked when the user tries to toggle the engine on/off.

- Precondition(s): the car must be parked with a minimum charge and not in diagnostics mode.
- Postcondition(s): the value of the engine's Boolean will have changed.

3.4 Change Gear Procedure (Gold proof)

Changes the car's gear to a value from the enumerated gears, given as a parameter.

- Precondition(s):
  - The engine must be on, not in diagnostics.
  - The parameter cannot equal anything other than "reverse" when the environment contains an obstruction.
  - The parameter cannot equal anything other than "parked" when the car requires charging or the driver has requested that the car be parked.
  - The parameter also cannot equal the same gear that the car is currently in.
  - The car's speed must be zero when the gear is changed, except when the user requests that the car park, in which case the speed must be greater than zero.
  - Only one, or none, of the park request, obstruction present, and charge enforcement Booleans can be true.
- Postcondition(s): N/A because the outcome is conditional (see the contract cases).

- Contract Case(s):
  - If the car's speed is greater than zero and the parameter equals parked, the gear will not have changed, and the car's **parkRequested** attribute will be true.
  - If an obstruction is present, the gear will be set to reverse.
  - If the car needs to be charged or the driver requested that the car park, and the car's speed equals zero, then the car's gear will be parked.
  - If any of the three previous postconditions are false, then the car's gear will have changed to the gear given in the parameter.

3.5 Diagnostics Switch Procedure (Gold proof)

Enables/disables diagnostics mode. The driver may cancel diagnostics after it has started.

- Precondition(s): the car must be parked and off with a minimum charge.
- Postcondition(s): the value of the diagnostics mode's Boolean will have changed.

3.6 Modify Speed Procedure (Gold proof)

Increments/decrements the car's speed according to the gear/brake's state.

- Invariant(s): for the car's speed to decrease, the car must: not be braking and have a negative speed and either be reversing or braking with a positive speed. Braking in reverse (at a negative speed) will still bring the car's speed to zero, so it should increase.
- Precondition(s):
  - The engine must be on and not in the parked gear or diagnostics mode.
  - The car's speed will be within the **MilesPerHour** range.
  - The environment's current speed limit will always at least equal the **SPEED_LIMIT_STEP_FACTOR**.
- Postcondition(s):
  - The speed will always be greater than the **MilesPerHour** minimum and less than the environment's current speed limit.
- Contract Case(s):
  - If the car is breaking, then:
    - If the car's speed was greater than zero before invoking this procedure, then the new speed will be greater than or equal to zero and less than the old speed,
    - Else, if the car's old speed was less than zero, then the new speed will be less than or equal to zero and greater than the old speed,
    - Else, the car's speed will remain the same at zero.
  - In any other scenario, the resulting speed will equal:
    - The current street's speed limit, if the old speed was already equal to the limit and the invariant is false, else
    - The **MilesPerHour** minimum, if the old speed was already equal to the minimum and the invariant is true,
    - Else, the old speed is incremented if the invariant is false, or else it will equal the old speed decremented.

3.7 Emergency Stop Procedure (Gold proof)

Drops the speed to zero; invoked when an unexpected obstruction becomes present.

- Precondition(s):
  - The car is driving, in motion, and an obstruction is present.
  - The speed limit will equal at least the **SPEED_LIMIT_STEP_FACTOR**.
  - The car will not be in diagnostics or need to be charged.
- Postcondition(s): the car's speed will equal zero, and the gear will change to reverse.

3.8 Generate Speed Limit Procedure (Gold proof)

Generates a new speed limit during route initialisation (for the street the car was parked on) and after the car turns onto a new street. The car will actively generate new speed limits throughout the journey, which is only required while the car is driving.

- Precondition(s):
  - The car must not be parked or off, but the speed will equal zero.
  - Diagnostics must be off, and the car must not need charging.
- Postcondition(s):
  - The resulting speed limit will always be greater than and a multiple of **SPEED_LIMIT_STEP_FACTOR**, and less than the **MilesPerHour** maximum.
  - All of the environment's other variables will remain unchanged, which is required to prove the route initialisation.

3.9 Initialise Route Procedure (Gold proof)

Route initialisation occurs when a route is started, but only when the car reached the previous destination. It initialises the number of turns required before the destination is reached and the first street's speed limit.

- Precondition(s):
  - The car must be on, charged, stationary and in the drive gear.
  - There must be no obstruction, and the car must not be in diagnostics mode.
  - If the previous destination was not reached, then the variables for initialisation will already be assigned values.
- Postcondition(s):
  - The speed limit, number of turns until the destination, and number of turns taken are initialised. All except the number of turns taken will be greater than zero.
  - The Boolean determining that the previous destination was reached will be false.

3.10 Car Turn Procedure (Gold proof)

Simulates progress in the route by incrementing the number of turns taken. It is invoked twice for every turn: first, to let the system know a turn is incoming, and second, to let it know the turn has been made. A new, random speed limit is generated after a turn.

- Precondition(s):
  - The car will be driving and has slowed down to zero in case it must give way.

  - The car must be charged, not in diagnostics mode, avoiding an obstruction, or has reached its destination.
  - There must be remaining turns to make: the number of turns taken will be less than the number of turns until the destination is reached.
- Postcondition(s): the incoming turn Boolean will be inverted, and the car's brake will activate if a turn is incoming.
- Contract Case(s):
  - If the invocation lets the system know that the turn was dealt with, the number of turns taken will increment.
  - If the invocation lets it know a turn is incoming, the number of turns will not have changes as the car is preparing to turn, not performing a turn.

3.11 Generate Scenario Function (Gold proof)

Returns a scenario based on probability.
- Precondition(s): the car is in drive and charged, not parking, in diagnostics, parking at the destination, turning, or dealing with an obstruction, in other words, not already dealing with a scenario.
- Postcondition(s): N/A as the function has no effect; it only returns a scenario.

3.12 Car Condition Check Function (Gold proof)

Returns the car's condition while it is active. The conditions are the notices that the driver can be given about the car.
- Precondition(s): only required while the car is active. Therefore, the car must be driving and charged, not diagnostics.
- Postcondition(s): N/A as the function has no effect; it only returns a condition.

3.13 Random Number Generation Function (Gold proof)

Returns a random integer between one and the parameter given.

- Precondition(s): the parameter given will be less than or equal to the parameter.
- Postcondition(s): the number returned will be between one and the parameter.

4 Proof of Consistency

The car's speed modification system has considerable complexity and is an important part. The car must be mindful of every constraint on its speed. The speed must not exceed the current street's speed limit or reverse too fast to maintain controllability. When braking, the car must not increase the speed when driving (as the car will be at a positive speed) or decrease it when reversing (as the car will be at a negative speed).

The proof examines whether, when breaking, the current system's expected functionality of decreasing the speed in a forward motion will encounter any runtime errors.

- s = the car's speed,
- Where S is the set (range) of integers of which the car's speed can be a member, $F = S'First$ and $E = S'Last$ (End),
- L = the environment's current speed limit,
- B = the Boolean determining whether the car's brake is active.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \Rightarrow s - 1 > F}\text{Ax} \quad \dfrac{}{\Gamma \Rightarrow s - 1 < L}\text{Ax}}{\Gamma \Rightarrow s - 1 > F \wedge s - 1 < L}\text{R}\wedge \quad \dfrac{}{\Gamma \Rightarrow s - 1 \geq 0}\text{Ax}}{\Gamma \Rightarrow s - 1 \geq 0 \wedge s - 1 > F \wedge s - 1 < L}\text{R}\wedge$$

$$\dfrac{}{B \wedge F < 0 \wedge E > 0 \wedge L \leq E \wedge s \geq 1 \wedge s \geq F \wedge s \leq L \Rightarrow s - 1 \geq 0 \wedge s - 1 > F \wedge s - 1 < L}\text{L}\wedge^{*}$$

$$\dfrac{}{\Rightarrow (B \wedge F < 0 \wedge E > 0 \wedge L \leq E \wedge s \geq 1 \wedge s \geq F \wedge s \leq L) \supset (s - 1 \geq 0 \wedge s - 1 > F \wedge s - 1 < L)}\text{R}\supset$$

Fig. 1. modifySpeed Proof of Consistency (SVG).

The proof in **Fig. 1** shows that the car's speed will always remain within the required bounds when decreasing while in drive and braking. The second derivation, L∧, is applied four times in one derivation. In the $s - 1 > F$ axiom branch, the result is an axiom as $F < 0$ (in (Γ) gamma) and $s \geq 1$, so $s - 1$ will never equal anything < 0.

Proving **modifySpeed** when the car is reversing can be done via the inverse of the proof in **Fig. 1** would be applicable and prove the formula is also true. The inverse being that the precondition contains $s \leq -1$ instead of $s \geq 1$, and the postcondition is instead $s + 1 \leq 0 \wedge s + 1 > F \wedge s + 1 < L$. The branch $s + 1 < L$ would be satisfiable for the same reason $s - 1 > F$ is satisfiable when the car is driving: in reverse, $s + 1 \leq 0$, and $L > 0$.

5 Extensions

5.1 Formal System Extensions

Parallelism – the driver's controls, the self-driving system, and the diagnostics mode are all contained within separate threads to allow the self-driving system to accept/reject the driver's controls based on the car's current state.

Records – the CarType and WorldType records act as classes because they are composite data types containing multiple attributes. The records are used for the same reason classes are used: to group information that belongs to the same object.

Contract Cases – some functions require conditional postconditions. For example, when trying to change the car's gear, the outcome where the user requested that the car pulls over to park will only be true *if* the gear requested is the parked gear and the car is in a natural motion (not dealing with obstructions). Contracts ensure that for a postcondition to be true, some state of preconditions must be true. Considering the previous

gear-change example, the preconditions for the park request contract case would specify that the discussed conditions need to be true for the request to be successful.

Random Number Generator – the random number generator cannot be directly accessed in Ada SPARK. It needs to be explicitly given a range of numbers to generate another number within and a type of number to generate; natural, real, or negative.

5.2 Problem Specification Extensions

Route Simulation – when the driver first puts the car into drive, the route is initialised with a random number of turns and a random speed limit of the current street. The car continuously accelerates toward the generated speed limit until an upcoming turn arrives (with a 10% chance of arriving every half a second). The car will slow down to zero, make the turn, generate a speed limit of the new street, and then accelerate toward the new speed limit again. This repeats until the number of turns taken equals the initially generated number of turns. When this occurs, the car then continues to accelerate to the speed limit a final time, with a 10% chance of arriving at the destination at any moment, which it will slow down and park.

Park Requests – when in motion if the driver wishes for the car to pull over, the driver can try changing the gear to park. A request for the car to park will be submitted to the system, and it will focus on slowing down and parking.

Obstruction Handler / Emergency Stops – the scenario generation function has a 5% chance to signify to the self-driving system's thread while driving that an obstruction has become present. In this scenario, the car must slow to zero MPH, reverse back until there is adequate space to divert the obstruction, brake to zero MPH again, change gear to drive, and then resume the journey. When dealing with an obstruction, the car does not accept any of the driver's requests to park until it is clear of the obstruction.

Breaking – when the car needs to slow down, a brake is activated. The brake can decrease/increase the speed to zero to immobilise the car. The break's functionality can also increase the car's speed from a negative speed (while reversing) to zero. It is often used when performing turns, parking the car, and dealing with emergency stops.

6 Conclusion

There are many constraints that a self-driving car may face. Introducing further complexity to the self-driving system would be interesting on top of the existing extensions. For example, cars must respond to traffic lights and traffic-dense routes. The system could be extended further to respond to these constraints, requiring more complex pre- and postconditions than existing ones.

The system could also load a dataset of routes instead of generating them based on probability. Doing so could also introduce the discussed complexities of traffic lights and traffic-dense routes and give the car a real destination to reach. Doing so would also remove unnecessary complexity of the current system, such as the route initialisation and the scenario generator.