RoboCar and Maze Solving - AER E 507X Final Project

Saveri Pal

December 11, 2017

---

1 Introduction

This project is divided in two parts. The first part, RoboCar make up the major chunk of this project. The second part, Maze solver is a minor part, attempted mainly to explore the prospect offered by model checking to solve maze puzzles. Section 2 describes the initial project plan, as submitted in the project proposal. Section 3 deals with the final project outcome of RoboCar. The various subsections of section 3 describe how the system is broken in smaller parts, while verifying and validating each part separately. Section 4 deals with the final project outcome of Maze Solver and section 4 contains the concluding remarks with scopes for future work. Section 5 contains the bibliography with all the sources cited.
The additional documents are provided in the final project GitHub repository,
branch '$submission\_finalProject$' and they are mentioned in the text.

2 Initial Project Plan

System to be analyzed: The environment of the system consists of four intersecting two-way roads. There will be two start points situated on opposite sides, from where the cars start. The cars will face obstacles on their journey from their start point to their end point on the opposite side. Obstacles will be presented in the following forms,

- A ditch in the lane
- Nails strewn on the lane
- pedestrian crossing the road
- STOP sign
- Traffic signal (Red, Green, Yellow)

The cars will have to veer through these obstacle on their way to the end, by following traffic rules and without deadlocking with each other. **Specifications to be verified:** Two self-driving car starting at opposite points reaches their end points on the other side, without crashing, deadlocking or breaking any traffic rule.

Terms of success for the project: There should be a model of the cars and all the various paths they can take from the start point to the end point. The model will be validated to check that there are ways a car can crash or drive into a ditch or deadlock with each other. Temporal logic specifications will verify that the cars can reach the end point by successfully avoiding all the obstacles presented. For different specifications the car should take different routes and not stick to the same route all the time.

Demonstration of analysis

- **Benchmark:** The standard against which the results will be compared is 100 % successful travel path (all obstacles avoided,no deadlock).
- **Analysis demonstration in class:** A road map highlighting the path taken by the cars will be shown.
- **Analysis demonstration in report:** the same road map will by attached as a picture in the report.
- **Measure results:** Modeling, verification and validation of the system with one car is 50 % success. The other 50 % is to do the same with two cars. The breakdown for modeling, verification and validation is, $30 + 10 + 10 = 50$.

3 Final Project Outcome: RoboCar

3.1 Initial Approach:

The initial approach was to create a state space of the four possible roads from the starting point to the end point and add a series of obstacles to the paths. Path 1 and Path 2 are parallel. Path 3 and Path 4 are two perpendicular roads that connect Path 1 and 2. Each path is defined by a set of obstacles such as follows,

Path 1: has a ditch in one lane.
Path 2: has a car coming from the opposite side.
Path 3: has a STOP sign.
path 4 and 1: has a Traffic signal intersection.

Obstacle Grouping:I decided to reduce the number of obstacles by grouping them. A ditch in the lane or nails strewn on the lane are similar kind of soft obstacles, which can be ignored or resolved. So keeping any one of them would serve the purpose. Again, the obstacle, pedestrians-crossing-road can be eliminated by the application of a STOP or traffic sign. When the car stops at a red light or a stop sign, it can be assumed that pedestrians cross the road only then. Therefore, this variable can be eliminated.

Challenges Encountered: To implement the initial plan, the initial model consisted of a main module and several sub-modules for avoiding a ditch, dealing with a STOP sign, traffic signal and car from an opposing lane. Making connections between all the modules made debugging difficult. I was not able to verify the model correctly by this approach. Therefore, I took a second approach.

3.2 Compositional Approach:

This approach incorporates breaking a system down into smaller sub-modules and verifying and validating each independently. At the failure of the initial approach, it was deemed suitable to approach the problem from the RoboCar's point of view. The behavior of RoboCar could be verified and validated separately for situations when it encounters a ditch, stop sign, traffic signal. From this it can be asserted that if the car faces such obstacles it would be able to correctly tackle each.
Obstacles: The obstacles considered in this system are listed as follows,

- Lane change - low priority
- Ditch - low priority
- Left or right turn - low priority
- Stop sign - high priority
- Traffic signal - high priority

Description: The system is broken in two levels, the higher level and the lower level. Higher level consists of three models for the main (highest in the hierarchy) and one each for low priority and high priority obstacles. The lower level consists of separate models for each obstacle and their inner workings. A diagram of the hierarchy is provided along with this submission (refer Picture 1).
NuXMV scripts for each of the modules is attached in the project GitHub repository along with this report. The LTL specifications for validation and verification of each of the modules is provided with the module description in the following sections.

3.2.1 Higher Level

<u>Main Module</u>
In this level, the model deals with all the traffic directives and obstacles RoboCar can encounter. The directives are grouped into high priority and low priority groups. Ditch, lane change and turn fall into low priority directives. Traffic signal and stop sign are high priority. Low priority directives can be ignored or executed when received, as their execution is not safety critical. However, traffic and stop sign directives need to be executed. Failing to execute these directives causes a car to crash. The variables in this module are,

1. high_priority - denotes whether an incoming command is high priority or not. Scalar variables with possible values of $\{yes, no\}$
2. low_priority - denotes whether an incoming command is low priority or not. Scalar variables with possible values of $\{yes, no\}$
3. resolve - denotes whether a command (high-priority/low-priority) is being executed or not. It is boolean.

(a) Function - This module executes the traffic commands received. A high priority command gets a priority in execution. Low priority commands are executed when there is no existing high priority command. If both high and low priority command is received together, the lower one is ignored and higher one is executed. At any point of time, the resolve variable can execute only one command.

(b) Validation - For validation, the following specifications are used to check the presence of these states in the model,

1. High priority command received: $\Diamond(high = yes \wedge low = no)$
2. Low priority command received: $\Diamond(high = no \wedge low = yes)$
3. Both high and low priority command received: $\Diamond(high = yes \wedge low = yes)$

The validations were true and their never claims produced counter example for occurrence of these states. The model possesses the expected behaviors. Hence, the model is validated.

(c) Verification The system is verified for the following properties,

1. When a high priority command is received, it is resolved in at most next two steps:

$$\Box(high = yes \rightarrow \chi\chi(resolve))$$

2. When both high priority and low priority command is received together, the low priority is ignored in the next state:

$$((high = yes \wedge low = yes) \rightarrow \Box\chi(low = no)))$$

3. If a command is received, eventually it is resolved depending on its priority:

$$((high = yes \vee low = yes) \rightarrow \Diamond resolve)$$

The model verifies all the properties. The never claims of the properties produced the correct path.

(d) NuXmv Code: refer to *high_low_priority.smv*

High Priority and Low Priority Module

The two modules are similar to each other. The only difference in them is that in the low priority module, any number of directives can be requested at a time. And a directive can be ignored if there is an existing directive already accepted. Whereas, in case of high priority (stop, traffic signal), both directives cannot happen together.

A. High Priority: Between Stop sign and Traffic signal, only one can happen at a time. In this model only one directive can be executed at a time. However, these are safety critical directive and none can be ignored. Therefore, it is **assumed** that execution time for a traffic/stop sign directive is small enough to complete its execution before another traffic or stop sign is encountered. There are two scalar variables,

1. command - {stop, traffic, none}
   It is the set of the high priority directives. 'none' denotes when there are no existing directive.
2. state - {ready, busy}
   'ready' denotes that a directive can be accepted now and 'busy' denotes that the state is currently busy.

Validation and verification -

1. Either stop or traffic can occur at a time: $\neg\Diamond(command = stop \wedge command = traffic)$
2. If a command is accepted, it will eventually be executed: $(command = stop) \rightarrow \Diamond(state = busy)$
3. If a state is busy now, it will eventually be ready: $(state = busy) \rightarrow \Diamond(state = ready)$

The specifications are proved true by the model checker and their counter examples show the correct transitions that follow the rules of the model. The model is verified and validated.

NuXmv Code: refer to *high_priority.smv*

B. Low Priority: These directives can be received alone or in combination with others. In such case, any one is accepted and executed, the rest is ignored. Since, they are not safety critical, ignoring them does not cause a problem. The objective of this module is to show how that low priority directives are chosen non-deterministically and that at any instant of time only one directive can be executed. There are two scalar variables in this model taking particular values,

1. command - {ditch, lanechange, turn, none}
   It is the set of all the low priority directives. 'none' denotes when there are no existing directive.
2. state - {ready, busy}
   'ready' denotes that a directive can be accepted now and 'busy' denotes that the state is currently busy and cannot accept further directives until the completion of the current execution.

Validation and verification -

1. More than one command cannot be received at a time: $\neg\Diamond(command = ditch \wedge command = turn)$

2. If a command is accepted, it will eventually be executed: $(command = ditch) \rightarrow \Diamond(state = busy)$

3. If a state is busy now, it will eventually be ready: $(state = busy) \rightarrow \Diamond(state = ready)$

The specifications are proved true by the model checker and their counter examples show the correct transitions that follow the rules of the model. The model is verified and validated.

NuXmv Code: refer to *low_priority.smv*

3.2.2 Lower Level

The lower level is made up of five separate modules dedicated to specific internal functions. Each of the modules is described as follows,

1. Lane Change: This module deals with changing of lanes by the RoboCar. It consists of a two-lane highway each lane going in opposite direction. This specific model can also be applied to two-lane highways in the same direction. There are three boolean variables, 'lane', 'car' and 'life'.

1. lane - denotes whether RoboCar changes lane or not.

2. car - denotes whether a car is present in the lane RoboCar wishes to change to.

3. life - denotes whether RoboCar is fine or crashed.

(a) Function - The objective of this module is to enable successful lane changes for RoboCar. For this, it checks whether a car is present in the adjacent lane and makes a decision to execute the directive or ignore. If it changes lane while a car is present, it crashes. A successful (or not) execution is depicted by the variable, life. When a crash occurs, it sets to FALSE. A successful lane change is indicated by a lane change with life still set to TRUE.

(b) Validation - To check that the model is correct, it is checked for two LTL specifications,

1. The car can crash: $\Diamond(car \wedge \chi lane)$

2. The car can successfully change lane: $\Diamond(lane \wedge life)$

Both the validations were true and their never claims produced counter example for correct crashing or success. Hence, the model is validated.

(c) Verification - The system has been verified for the following safety and liveliness properties,

1. Safety property: No lane change happens if a car is present in the adjacent lane.

$$\Diamond(car \rightarrow \neg lane)$$

2. Liveliness property: Eventually, successful lane change happens.

$$\Diamond(lane \wedge life)$$

Both the safety and liveliness properties are shown to be true by the model checker. Hence, the model is verified. This means the module is functioning correctly as it is supposed to do.

(d) NuXmv Code: refer to *lanechange.smv*

2. Ditch on lane: This module is called into action when a ditch is detected on the lane. It consists of a two-lane highway each going in opposite direction. There are three boolean variables, 'pos', 'lane', 'obstacle' and one integer variable, 'life'.

1. pos - denotes whether final destination (that is, the point on lane after crossing/avoiding ditch) is reached or not.
2. lane - denotes whether lane changed or not.
3. obstacle - denotes whether an obstacle is detected on lane or not.
4. life (0-10) - denotes whether RoboCar is fine, crashed or drove across the ditch.

(a) Function - The objective of this module is to successfully avoid a ditch. If there is a ditch detected in the lane, the car can drive across it or it can avoid it by changing lane to the adjacent lane and coming back to the original lane. Driving across the ditch is a soft obstacle that does not affect the safety of the RoboCar. It just deducts 5 points from the life. However, in the endeavor to avoid the ditch, if the car fails to come back to the original lane that is considered a hard obstacle. This is safety critical. For a hard obstacle, life points reduce to zero. The 'life' variable is there in the code to show what possible actions are possible in the model.
Assumption - These are the following points incorporated in the model,

- car can change lane even if there isn't any ditch.
- when ditch is detected, it can go in reverse to a position where no ditch is detected.
- it can go back and forth between initial and final position.
- it might not face obstacle.

(a) Validation - To check the validity of the model is checked by LTL specifications,

1. The car can drive through the obstacle: $\Diamond(obstacle \rightarrow (life = 5))$
2. The car can crash: $\Diamond(lane \wedge (life = 0))$

(c) Verification - The system has been verified for the following properties,

1. If the car changes lane, it eventually comes back without crashing.

$$\Diamond(lane \rightarrow (\neg lane \wedge life = 10))$$

2. The car eventually crosses the ditch successfully.

$$\Diamond(obstacle \wedge \Diamond life = 10)$$

Both properties are verified true. Module verified. Since, the ditch is not a hard obstacle, it does not have safety properties for itself, apart from the lane changing part.

(d) NuXmv Code: refer to *ditchonroad.smv*

3. Left/right Turn: This module deals with the turn taken by RoboCar on road. It contains a two-lane highway with exits on right and left sides. There are five boolean variables,

1. lind - denotes left indicator is ON/OFF
2. rind - denotes right indicator is ON/OFF
3. lturn - denotes left turn taken/not
4. rturn - denotes right turn is taken/not

5. life - denotes whether RoboCar is fine/crashed

(a) Function - The objective of this module is to assist in taking correct turns; turn on indicator then take the corresponding turn. For example, Right indicator then right turn. Soft mistakes include Right turn without an indicator and vice versa, they do not cause crashes (let's assume!). Hard mistake are the one where right turn is taken on a left indicator or vice versa. That causes the car to crash.
Assumption: car can turn indicator ON/OFF repeatedly without taking a turn afterwards.

(b) Validation - To check the validity of the system, the following specifications is checked,

1. The car can crash: $\Diamond(lind \wedge rturn)$

(c) Verification - The system has been checked for the following properties,

1. The car can successfully take a turn:

$$\Diamond(lind \wedge lturn)$$

This property is verified true. This module fulfills its objective.

(d) NuXmv Code: refer to $leftrightturn.smv$

4. Traffic Signal: This module assists in correctly resolving a traffic signal, when encountered. It consists of a basic traffic signal with three lights, red, green and yellow. There are two scalar variables each taking particular symbolic values.

1. go - denotes whether the car is moving or not. Possible values are {yes, no}
2. light - denotes the possible color lights encountered at a traffic signal, {red, green, yellow}

(a) Function - At a signal light there is no control over what color might be ON. In such a situation, it was found that modeling just the behavior of the car at each possible light color (red, green, yellow) is easier to handle and design. The scalar variable 'light' is assigned a random value initially. Based on that, the next value of 'go' is determined. At a green light, the car keeps going. At yellow, it may stop or may keep going. At red, it stops at the next time instant.

(b) Validation - To check the validity of the system, the value of 'light' was initially assigned to red, green and yellow each and the following specifications were checked,

1. for init(light) := yellow, $\Diamond((light = yellow) \rightarrow \chi(go = yes | go = no))$
2. for init(light) := red, $\Diamond((light = red) \rightarrow \chi(go = no))$
3. for init(light) := green, $\Diamond((light = green) \rightarrow \chi(go = yes))$

All the specifications are proved true. The behavior of the system is as it is supposed to be. The model is valid.

(c) Verification - The system has been verified for the safety following property, with initial value of 'light' set to random.

1. Safety property: When red light is detected, the car always stops at the next time instant

$$\Box((light = red) \rightarrow \chi(go = no))$$

The specification is verified true.

(d) NuXmv Code: refer to $trafficsignal2.smv$

5. Stop sign: This module assists in correctly resolving a stop sign, when encountered. It consists a four-way stop. However, this model is for valid single STOP signs too. The variables in this model are,

1. go - denotes whether RoboCar is moving or not.
2. stop - STOP sign detected or not.
3. n_obs - number of cars reaching the intersection before this car
4. life - denotes whether RoboCar is fine or Crashed.

(a) Function - When a STOP sign is detected, Robocar counts the number of cars that reach an intersection before it. It stops and counts down till all of them pass the intersection. Only then the stop sign turns FALSE and RoboCar starts crossing the intersection. Therefore, the variable stop is mainly dependent on n_obs.
Rule: car that reaches intersection first, passes first.
Assumption: All cars correctly follows the above rule.

(b) Validation - To check the validity of the system, the following specification is checked,

1. The car counts down till 0 sequentially, if there are cars reaching the intersection before it:
   $\square(n\_obs = 3 \rightarrow \chi n\_obs = 0$
2. When the number of obstacles counts down to zero, the car starts from stop:
   $\Diamond(n\_obs = 1 \rightarrow \chi(n\_obs = 0 \wedge go))$

The validation is proved true by the model checker. Their never claims produces a counterexample of the correct count down and start condition. Hence, model is verified.

(c) Verification - The system has been verified for the following properties,

1. If there is a stop sign, the car will stop at the next instant:

$$\square(stop \rightarrow \chi\neg go)$$

2. The car will only start from stop position when stop is FALSE (which means when $n\_obs = 0$):

$$\square(\neg stop \rightarrow \chi go)$$

The above safety properties are true. This module is verified.

(d) NuXmv Code: refer to $stopmodule.smv$

4 Maze Solver

4.1 Motivation and Challenges:

The initial plan of this project, as outline in section 2, consisted of a motion planning problem where a car starts from an original position and reaches a final destinations while tackling obstacles on the way. This problem can be reduced to a maze solving problem. The similarities being, the presence of a start point and an end point, and obstacles on the way.
The initial approach was to replicate a small part of the motion planning problem solved in, 'Evaluation and Benchmarking for Robot Motion Planning Problems Using TuLiP', by N. Spooner et al. In this paper,the Temporal Logic Planning (TuLiP) toolbox has been used to generate benchmark grids that are then solved in NuSMV. However, the problem faced while working on this was that the NuSMV wrapper used in this work is for an older version of TuLiP, which could not be installed. Finally, for this problem, a Python based package Gr(1)Py has been used to solve two mazes.

4.2 Background:

Reactive Synthesis: Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specifications. It offers an approach where correct implementation of a system is automatically obtained from a given specification. Thus, the need to manually design a system and then verify it by model checking is eliminated. However, one of the biggest challenge of reactive synthesis is complexity for LTL is double exponential to the length of the formula.
General Reactivity of Rank 1: To address the above challenge, General Reactivity of Rank 1 or GR(1) was introduced by Piterman et al. It has an efficient polynomial time symbolic synthesis algorithm. The class of GR(1) formulas is sufficiently expressive to provide a complete specification of many designs. GR(1) is a class of LTL formula of the form,

$$(\square\lozenge P_1 \wedge .... \wedge \square\lozenge P_m) \rightarrow (\square\lozenge Q_1 \wedge .... \wedge \square\lozenge Q_n)$$

Here $P_i and Q_i$ are boolean combination of atomic propositions.
GR1Py: It is an enumerative reactive synthesis tool for GR(1) fragment of LTL. It is a Python based application package. In this package the maze problem can be designed in the form of a game between the system and the environment. Static and dynamic obstacles can be implemented and the goal of the system is to satisfy the specification regardless of the actions of the environment.

4.3 Problem Setup and Results:

Problem Setup: For this work, two 2-dimensional three by three grids is considered. Each cell is adjacent to four or less other cells. Two cells are considered to be adjacent if they share one edge. The pointer must always occupy exactly one cell at a time. Particular cells are marked as static obstacles, as shown in Picture 2 (fig 1 and fig 2) as black squares. An obstacle cell is forbidden. Diagonal movements are not allowed. The cell marked 'I' denotes the initial position, and the cell marked 'G' is the goal cell.
Obstacle Benchmarks: Two types of complexity metric is considered. In the first grid, the grid has narrow one-cell width corridors with defined dead ends (refer Picture 2, figure 1). There is only one correct path to reach the goal cell. The second grid has open corridors and well-defined shorter and longer route to reach the goal cell from the initial cell (refer Attachment 2, figure 2).
Results: The system was designed in Python Gr1Py. For a particular system goal, a counter example of the correct path from initial to goal cell is generated. For Grid 1 (Picture 2, figure 1), the counter example contained the correct path with never encountering a dead end. For Grid 2 (Attachment 2, figure 2), the counter example contained the shortest path to the goal cell. Hence, the system goal is correctly verified and defined by the system.

Gr1py odes: The codes for the two grids are attached in the GitHub repository.
for Grid 1, refer *threeby*3_*gw*.*spc*
for Grid 2, refer *threeby*3_1.*spc*

5 Conclusion and Future Work

The first and major part of this project, verifying and validating the behavior of an automated car, named RoboCar, has been successfully executed. The system was broken down into various modules and sub-modules, each of which has been validated and verified. With all the modules verified, it can be said that when faced with traffic obstacles, the car will be able to tackle them efficiently. An extension of this work could be to connect the software modules to hardware parts and verify and validate the correct functioning of that system.
The second and minor part consisting of a maze solver has been employed to solve two, three by three mazes. The results show good prospects as the mazes were solved for shortest routes. It opened up the scope to apply the tool to analyze larger and more complex mazes in future.

6 Bibliography:

1. T. Wongpiromsarn and R. Murray, Formal Verification of an Autonomous Vehicle System, Conference of Decision and Control (2008)
2. K. Y. Rozier , Linear Temporal Logic Symbolic Model Checking, Computer Science Review (2010), doi: 10.1016/j.cosrev.2010.06.002
3. N. Spooner et al, Evaluation and Benchmarking for Robot Motion Planning Problems Using TuLiP, CDS Technical Report (2012)
4. R. Cavada, A. Cimatti, et al, NuSMV 2.6 Tutorial, Italy
5. Cimatti, A., Clarke, E., Giunchiglia, F. et al, NuSMV: a New Symbolic Model Verifier, STTT(2000)2:410
6. Temporal Logic Planning Toolbox, https://github.com/tulip-control/tulip-control.git
7. N. Piterman, A. Pnueli and Y. Sa'ar, Synthesis of Reactive(1) Designs (2012), https://doi.org/10.1016/j.jcss.2011.08.007
8. S. Maoz and J. Ringert, GR(1) Synthesis for LTL SPecification Patterns, ESEC/SIGSOFT FSE 2015. DOI10.1145/2786805.2786824
9. gr1py https://github.com/slivingston/gr1py