Autonomous Driving System Using model checker: Final report

CSEEE6863 F20 Final project

Erilda Danaj, Hengjiu Kang

Abstract

In this project, we simplified the road and traffic problems into two models, an Intuitive model and a complex model. We also proposed two autonomous driving systems(ADS) to control the vehicle, delivering a safe and efficient experience to passengers. In the Intuitive model, if it is possible, the ADS can always avoid the collision, while in the complex model the ADS will not only avoid collisions but can also find the faster routes. In the later section we will discuss the corner cases within each model that ADS cannot solve -- or no one can solve. These two autonomous driving systems and their strategies were verified using model checking methods.

Introduction

Autonomous driving technology helps people to travel faster and safer thanks to the potential it brings to us that vehicles will have better perception on sensing the environment and more accurate control to maneuver. Planning is a hard problem in the autonomous driving industry since driving style is very subjective even though all drivers are *de jure* obeying to the laws and regulations. Autonomous driving system, or more specifically the planning strategy, is highly dependent on the Operational Design Domain (ODD). For example in the low-speed campus scenario with fixed routes, an autonomous vehicle can have a simple planning strategy and operate safely by stopping whenever it sees an obstacle. However in the more complex high-speed free way scenario, an autonomous driving vehicle must have a better perception system that can see vehicles at least hundreds meters ahead and have enough room to brake or change lanes. The most complex scenario is urban operation in busy cities including San Francisco, New York, Tokyo, Shanghai, and etc. Not only too many detected objects overwhelms the perception system but also the complex traffic regulations and those who do not care about the rules, can quite mislead the autonomous driving system.

While always putting safety as the first priority, different companies proposed different Autonomous Driving Systems (Uber) (Waymo) (GM Cruise), and some also proposed formal models for safe driving (Nister 1) (Shalev-shwartz 1). In this paper, we discuss two simplified road models, an intuitive model and a complex model, and then we propose two Autonomous Driving Systems for each one accordingly. Both ADSs are incorporated with model checking tools CBMC, and we will evaluate their performance in different conditions and scenarios, including corner cases.

Autonomous Driving System Intuitive Model

Problem statement

In the intuitive mode, we construct a road with fixed **n** lanes with **m** other vehicles plus **one** ego vehicle.

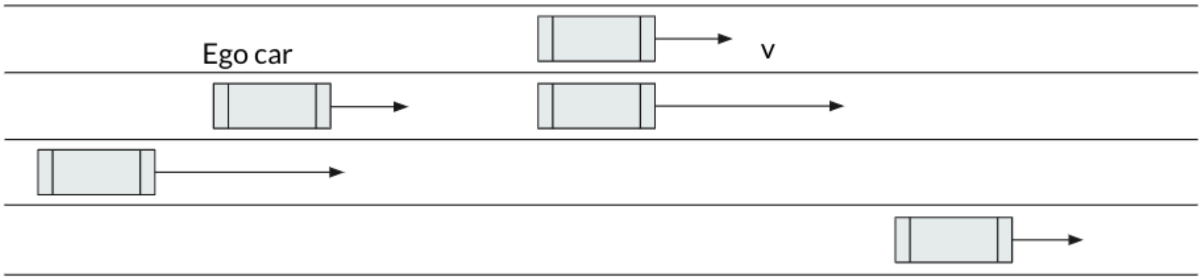


Figure1. Intuitive road setup
All the vehicles have the same velocity direction but different amplitudes(speeds). All **m** other vehicles hold constant speeds and do not change.
According to an assumed road regulation, vehicles have the max speed limit **Vmax.**
Also according to an assumed road regulation, the minimum distance between two vehicles on the same lane is **dmin,** which we called the minimum safe distance.
Ego vehicle has four possible operations:

1. Run at the maximum speed **Vmax**.
2. Stop immediately.
3. Change to the left lane.
4. Change to the right lane.

We check the possible collision by “checking if the distance between ego vehicle and any other vehicle on the same lane, is smaller than the minimum safe distance in the **next** second”

Safety property

There is only one safety property: Do not collide in the next second, which means the next step choice should not result in collision in the next second.

Liveness property

There is no special liveness property specified in the intuitive model.

Initial states

This program generates initial state by creating an array of other vehicles and then creating ego vehicles. By default the values are set to “nondet” but bounded by “assume”, to limit them in a reasonable range.
For specific initial states, we will use different initialization processes and strategies. Please refer to the “Result” section for more details.

Autonomous Driving System Complex Model

Intuitive model provides very limited options to the ego vehicle, thus in many cases that the ego vehicle cannot keep a safe distance from other vehicles which is considered as dangerous. In the complex model, we will set up a more complex road case and we will leverage the s-t graph (LaValle #) to find a better solution for the ADS.

Problem statement

In this model, other vehicles have the same design and characteristics as an intuitive model, but in the complex model we will utilize an S-T planner to optimize the route and we have a liveness property that “ego vehicle should select a route that can go faster”.
In the complex model, instead of having four possible operations, ego vehicle five operations:

1. Run at the maximum speed **Vmax**.
2. Stop immediately.
3. Change to the left lane.
4. Change to the right lane.
5. Run at any speed with integer value between 0 and macro MAX_SPEED

In order to simplify the calculation, we have a minimum vehicle speed limit higher than 0.

Safety property

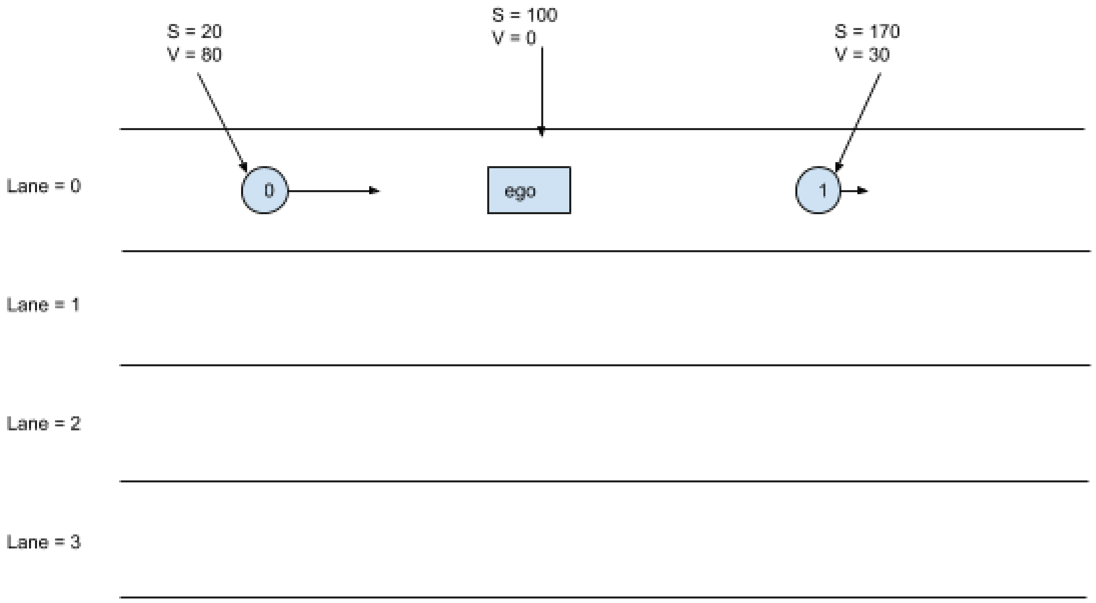
If possible, ego vehicle should not collide with any other vehicle in the next second.

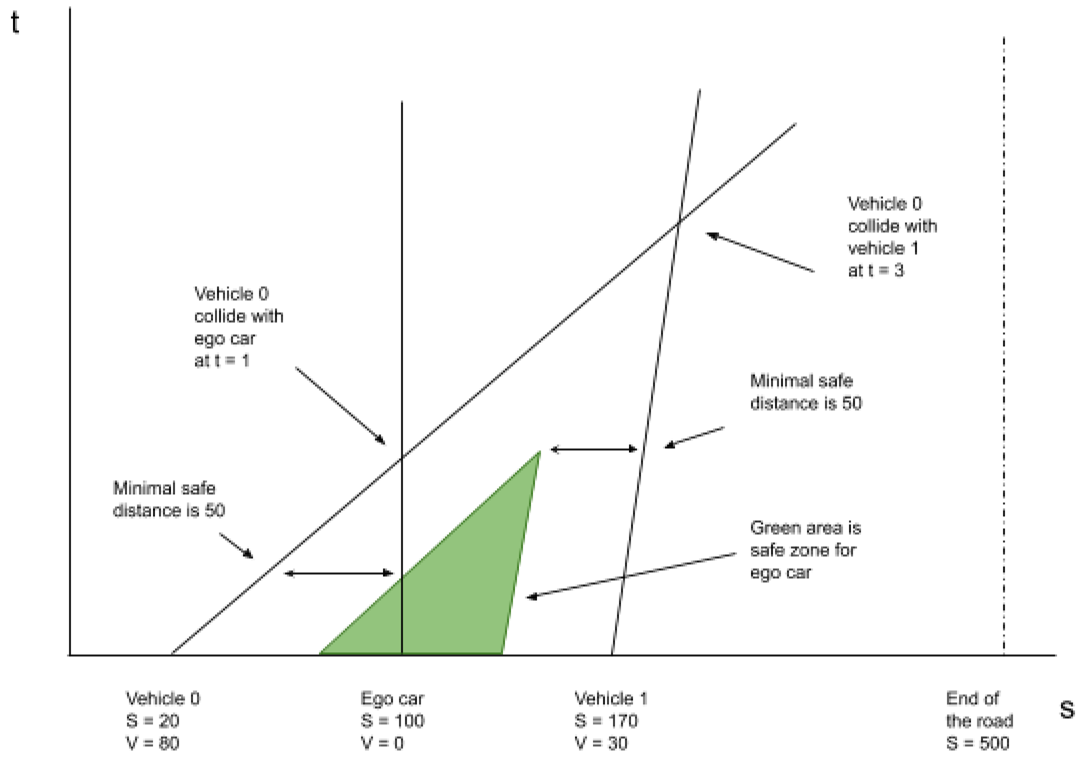
Liveness property

Ego car should choose the operation that can go faster/further.

S-T planning graph

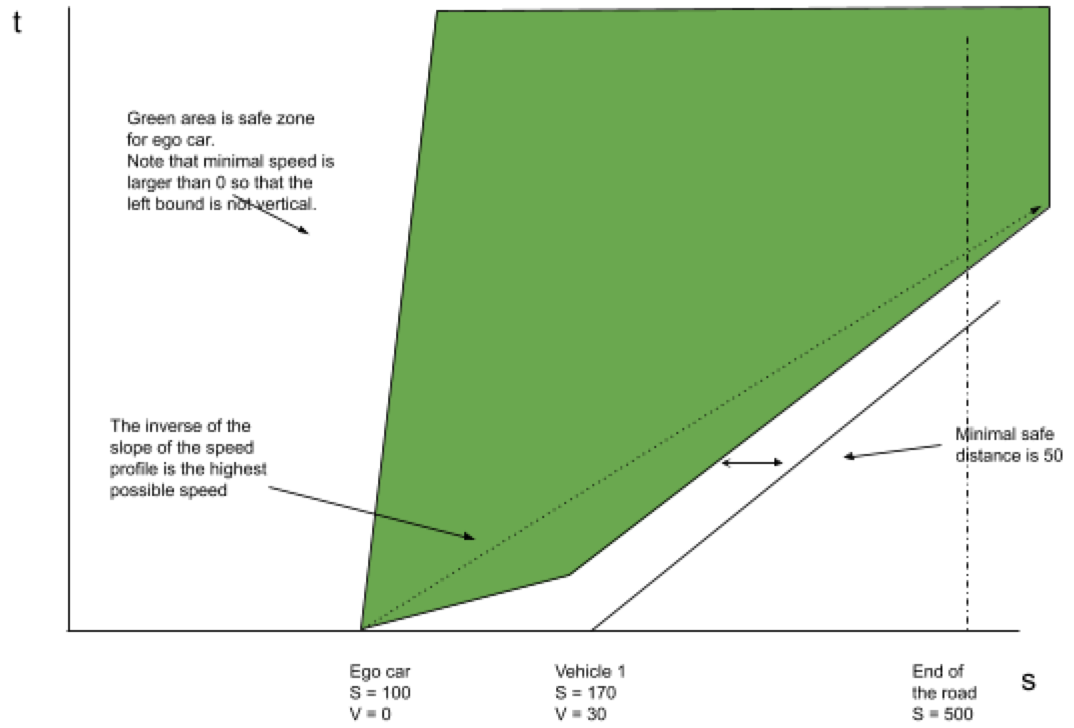
S-T planning graph is a cartesian coordinate system that expresses travel distance on the x-axis(or y-axis) and time on the y-axis(or x-axis). Two vehicles running at constant speed will draw two straight lines on the coordinate system. Whenever two lines have an intersection, two vehicles will collide. For example:



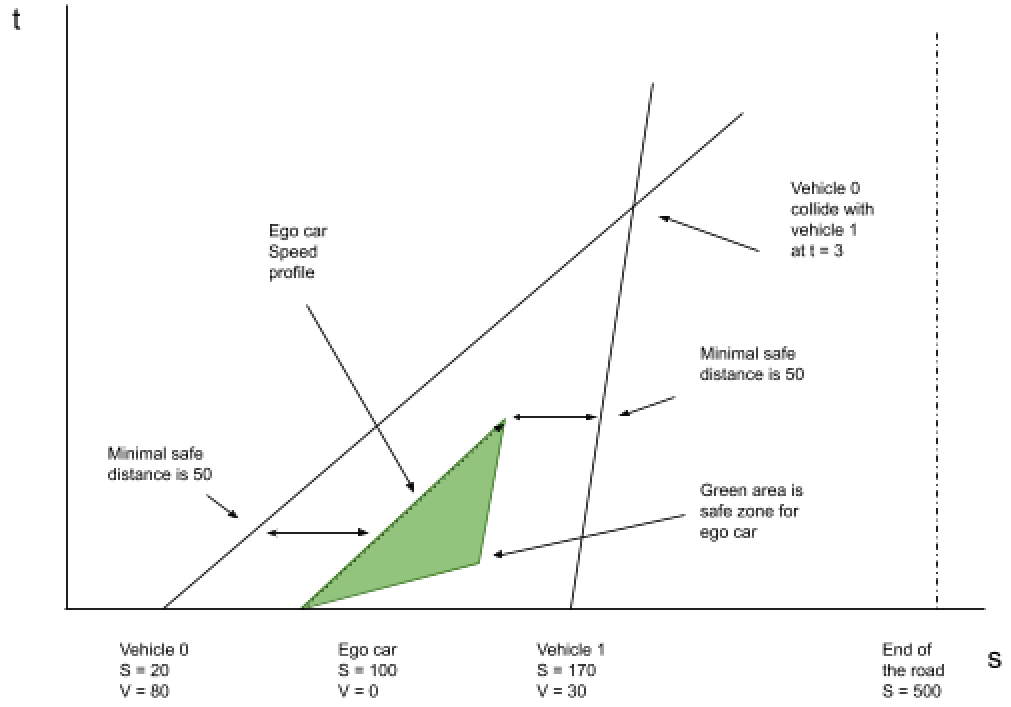


S-T graph planning strategy

In the complex model, when we apply the S-T graph, we also need to define the strategies. According to the liveness property we specified, we will choose the route which has "the highest average speed to reach the furthest point". If the safe zone we showed above is opened to the right side until the end of the road, then we can chose the highest possible speed without colliding with any other vehicle as the solution of this graph, for example:

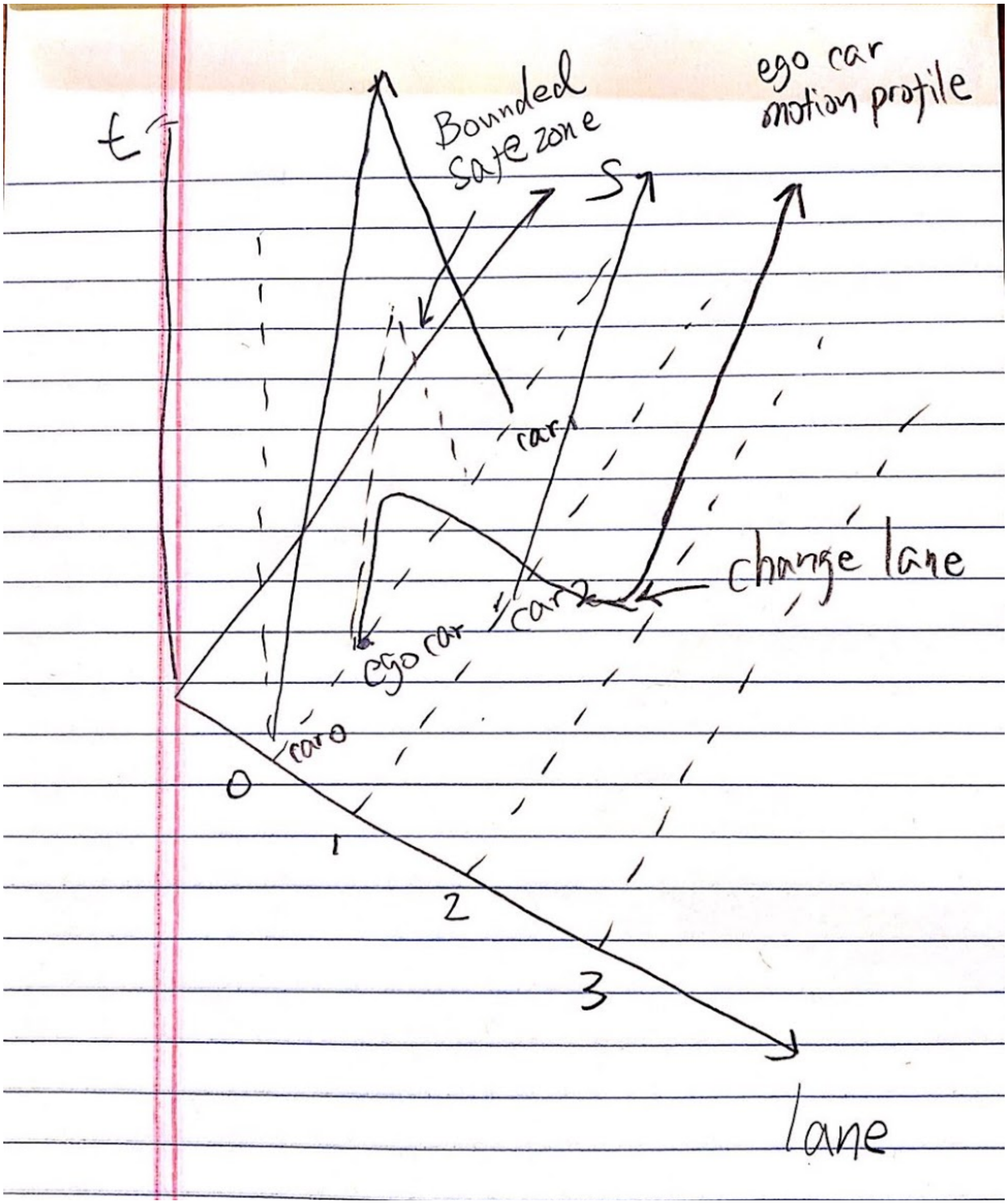

t
Green area is safe zone for ego car.
Note that minimal speed is larger than 0 so that the left bound is not vertical.
The inverse of the slope of the speed profile is the highest possible speed
Minimal safe distance is 50
Ego car
S = 100
V = 0
Vehicle 1
S = 170
V = 30
End of the road
S = 500
s

However if the green zone is bounded, then we choose the speed profile as following:


t
Ego car Speed profile
Vehicle 0 collide with vehicle 1 at t = 3
Minimal safe distance is 50
Minimal safe distance is 50
Green area is safe zone for ego car
Vehicle 0
S = 20
V = 80
Ego car
S = 100
V = 0
Vehicle 1
S = 170
V = 30
End of the road
S = 500
s

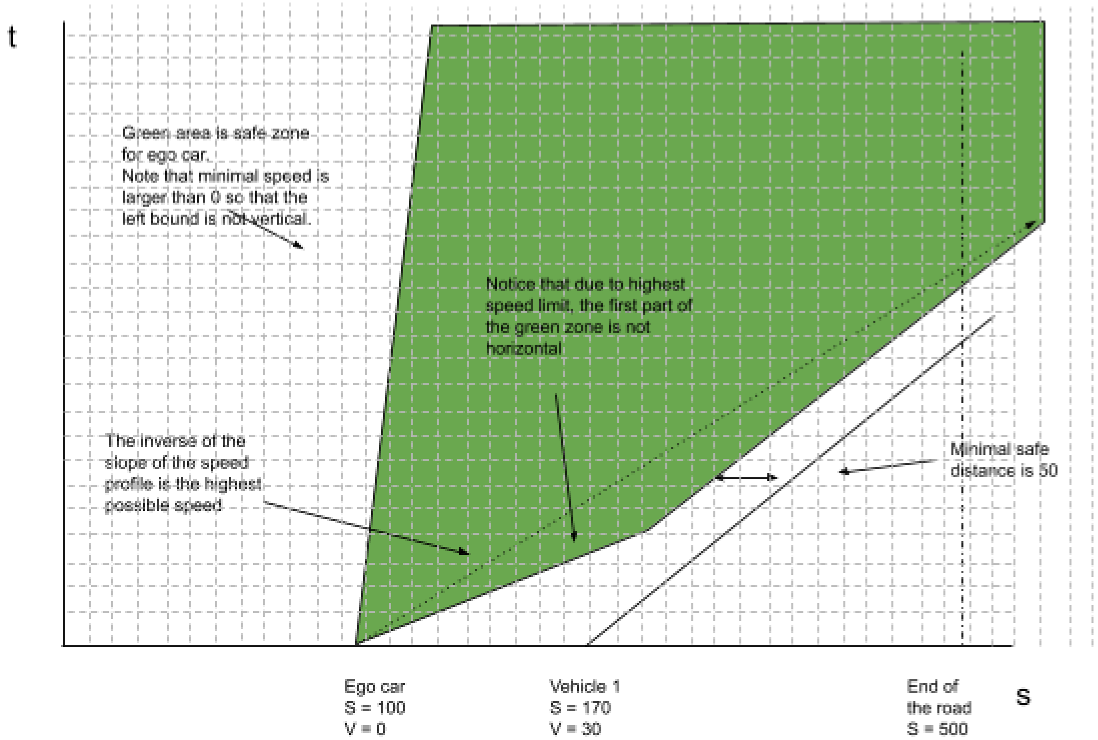
We can divide s-t graph results into different priority categories. For example because we do not do full travel planning here, i.e we do not plan multiple steps in the future, we only care the current selection, so the speed profile that can reach the destination always has higher priority than the profile in the bounded green zone, no matter how fast that speed profile would be.

Because in the latter case, we do not guarantee to be able to reach the destination. However if it is possible we can always change to the adjacent lane in the future, but that will result in 3-dimensional s-t graph which is too complex for our project, i.e:



How to solve the S-T graph

Constructing a polygon and finding optimized speed profile line inside the polygon are not easy. So we will not find the closed-form expression but the approximated result using occupancy map algorithm. We will construct a map with grid size 1-by-1 (1ft on x-axis and 1 second on y-axis).


t
Green area is safe zone for ego car.
Note that minimal speed is larger than 0 so that the left bound is not vertical.
Notice that due to highest speed limit, the first part of the green zone is not horizontal
The inverse of the slope of the speed profile is the highest possible speed
Minimal safe distance is 50
Ego car S = 100 V = 0
Vehicle 1 S = 170 V = 30
End of the road S = 500
s

Initial state

Complex model has similar initial state setups as intuitive model. We will discuss details in the "Result" section.

Result

Intuitive model: the cases where ADS guarantees safety

We first have to define such "safety" as "not collide with other vehicles in the next second", not in the indefinite future. Given that all other vehicles do not change lanes nor change speed, we can approve that as long as the relative distance between ego vehicle and all other vehicles in the same lane is not smaller than the minimum safe distance in the next second, there will be no collision in the next second. For example in the following cases, ADS is safe:

1. Only ego vehicle on the current lane.
2. All other vehicles are either in front of the ego vehicle or in the rear of the ego vehicle.
3. If there is(are) vehicle(s) on the both front and rear side of the ego vehicle, the distance between vehicles on the front and rear side vehicles does not decrease.

We can prove the above cases by setting up different initial states.

Environment variables

There are two major environment variables:

- MAX_CARS: how many other vehicles
- MAX_LANE_WIDTH: how many lanes

Each vehicle has three member variables:

- lane: which lane the vehicle is currently on. Only the ego vehicle can change the lane.
- speed: the speed of the current vehicle. Only the ego vehicle can change the speed.
- position: the position of the current vehicle on the current lane.

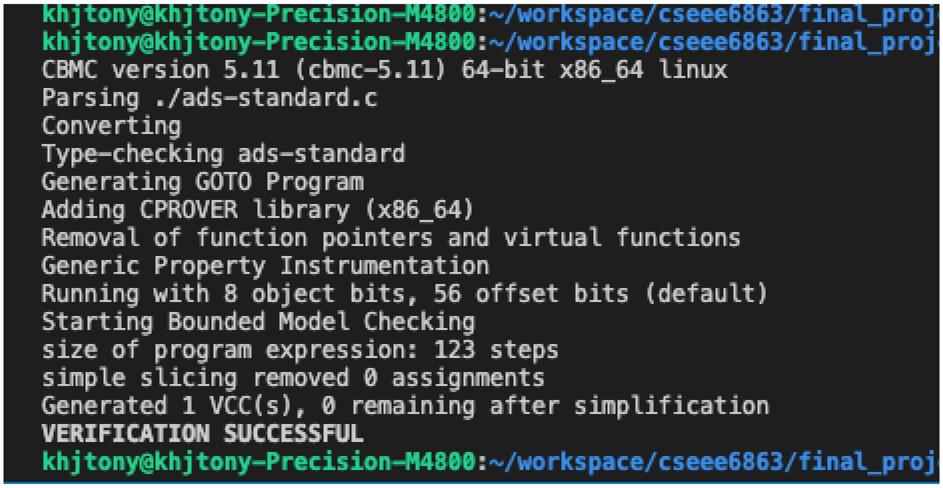
CBMC Command to check the model

cbmc ./ads-standard.c --stop-on-fail --compact-trace --unwind 10

First case: Only ego vehicle on the road

In this case, we can set variable **#define MAX_CARS = 0**, in which case only the ego vehicle is in the world.

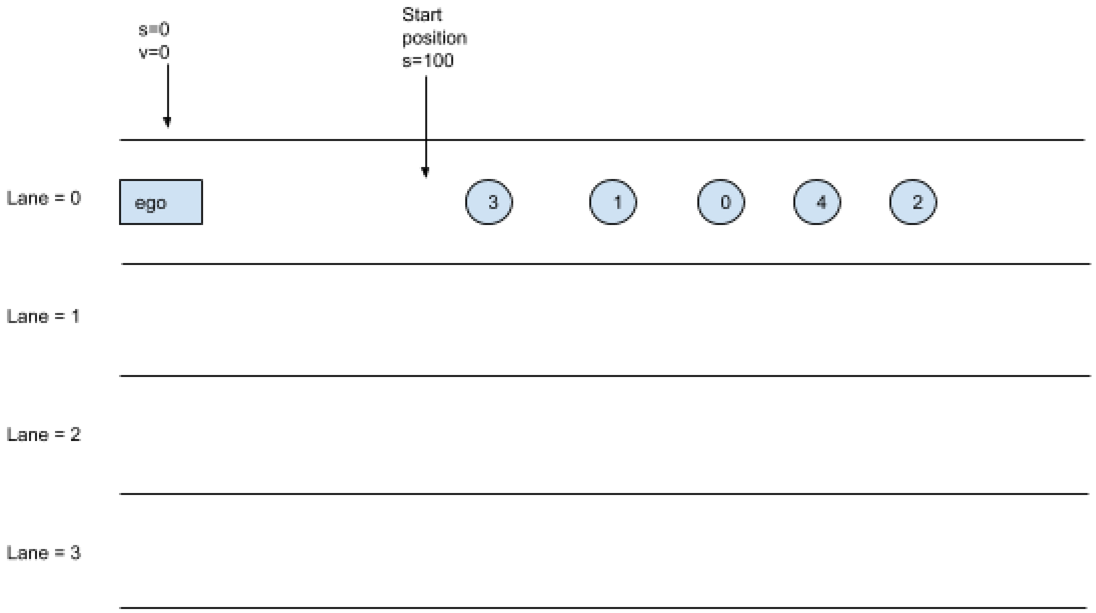
The model checking result is as following:



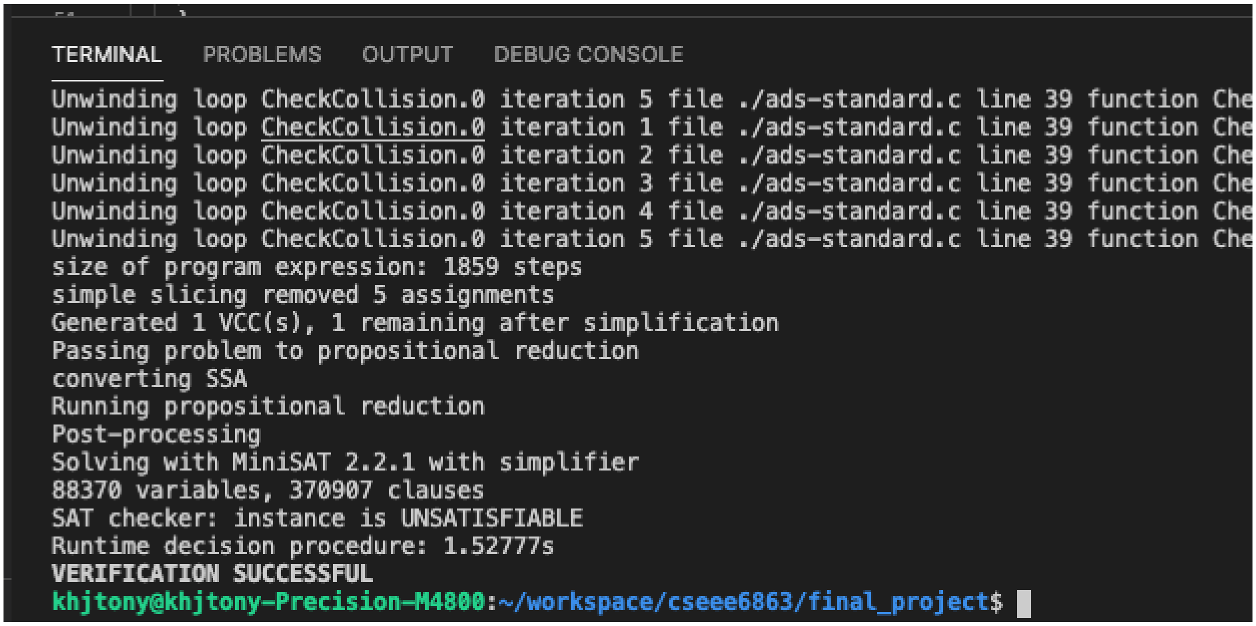
```
khjtony@khjtony-Precision-M4800:~/workspace/cseee6863/final_proj
khjtony@khjtony-Precision-M4800:~/workspace/cseee6863/final_proj
CBMC version 5.11 (cbmc-5.11) 64-bit x86_64 linux
Parsing ./ads-standard.c
Converting
Type-checking ads-standard
Generating GOTO Program
Adding CPROVER library (x86_64)
Removal of function pointers and virtual functions
Generic Property Instrumentation
Running with 8 object bits, 56 offset bits (default)
Starting Bounded Model Checking
size of program expression: 123 steps
simple slicing removed 0 assignments
Generated 1 VCC(s), 0 remaining after simplification
VERIFICATION SUCCESSFUL
khjtony@khjtony-Precision-M4800:~/workspace/cseee6863/final_proj
```

Second case: All other vehicles are on the one side

In the second case, we can set ego vehicles's initial position to 0 instead of an undetermined value, so all the other vehicles are in front of the ego vehicle and all of them have position >= 100:



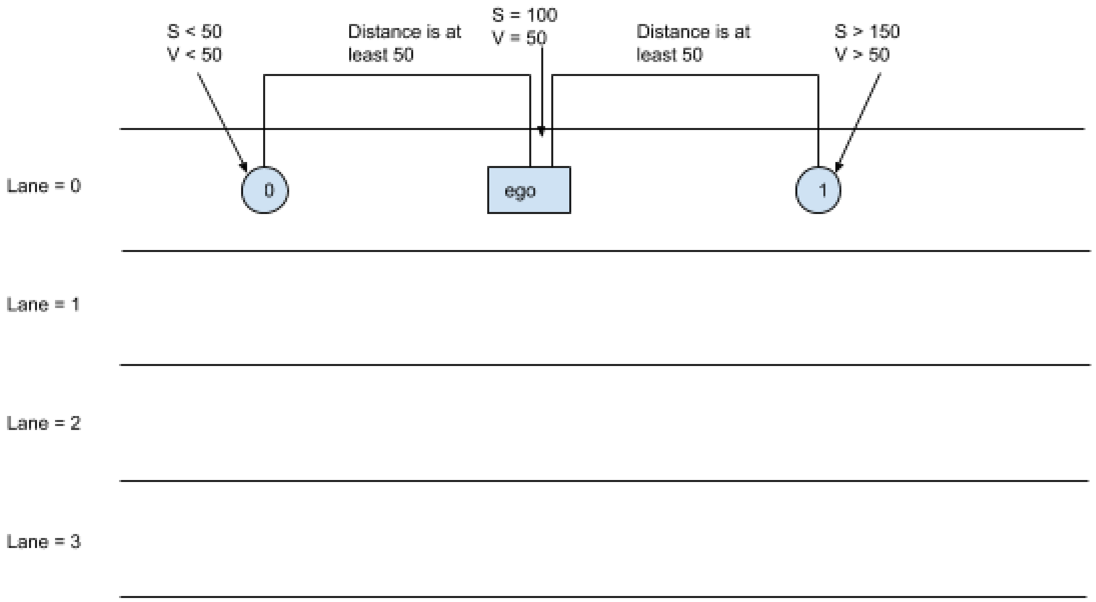
And the model checking result is shown as following:



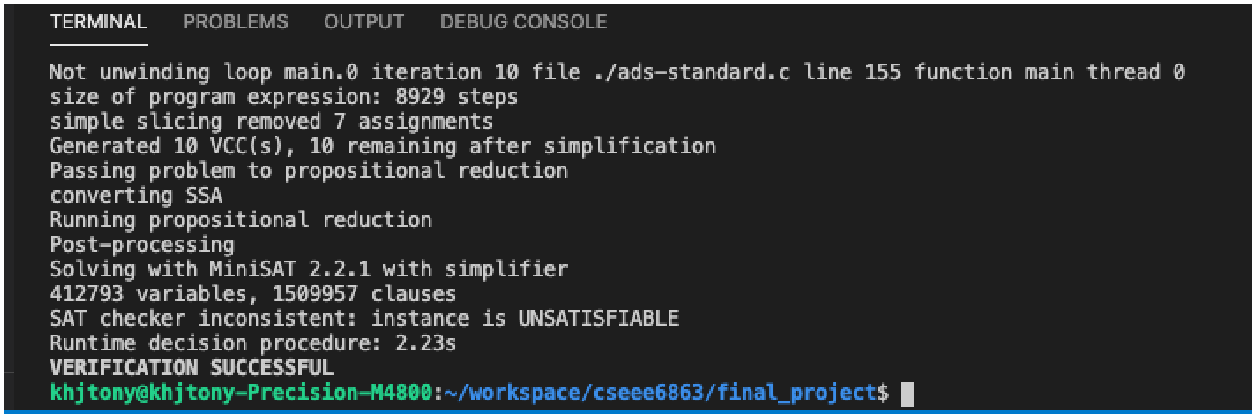
```
Unwinding loop CheckCollision.0 iteration 5 file ./ads-standard.c line 39 function Che
Unwinding loop CheckCollision.0 iteration 1 file ./ads-standard.c line 39 function Che
Unwinding loop CheckCollision.0 iteration 2 file ./ads-standard.c line 39 function Che
Unwinding loop CheckCollision.0 iteration 3 file ./ads-standard.c line 39 function Che
Unwinding loop CheckCollision.0 iteration 4 file ./ads-standard.c line 39 function Che
Unwinding loop CheckCollision.0 iteration 5 file ./ads-standard.c line 39 function Che
size of program expression: 1859 steps
simple slicing removed 5 assignments
Generated 1 VCC(s), 1 remaining after simplification
Passing problem to propositional reduction
converting SSA
Running propositional reduction
Post-processing
Solving with MiniSAT 2.2.1 with simplifier
88370 variables, 370907 clauses
SAT checker: instance is UNSATISFIABLE
Runtime decision procedure: 1.52777s
VERIFICATION SUCCESSFUL
khjtony@khjtony-Precision-M4800:~/workspace/cseee6863/final_project$
```

Third case: Other vehicles do not move closer

In this case, we need to manually set other vehicles initial conditions and give them different speeds.



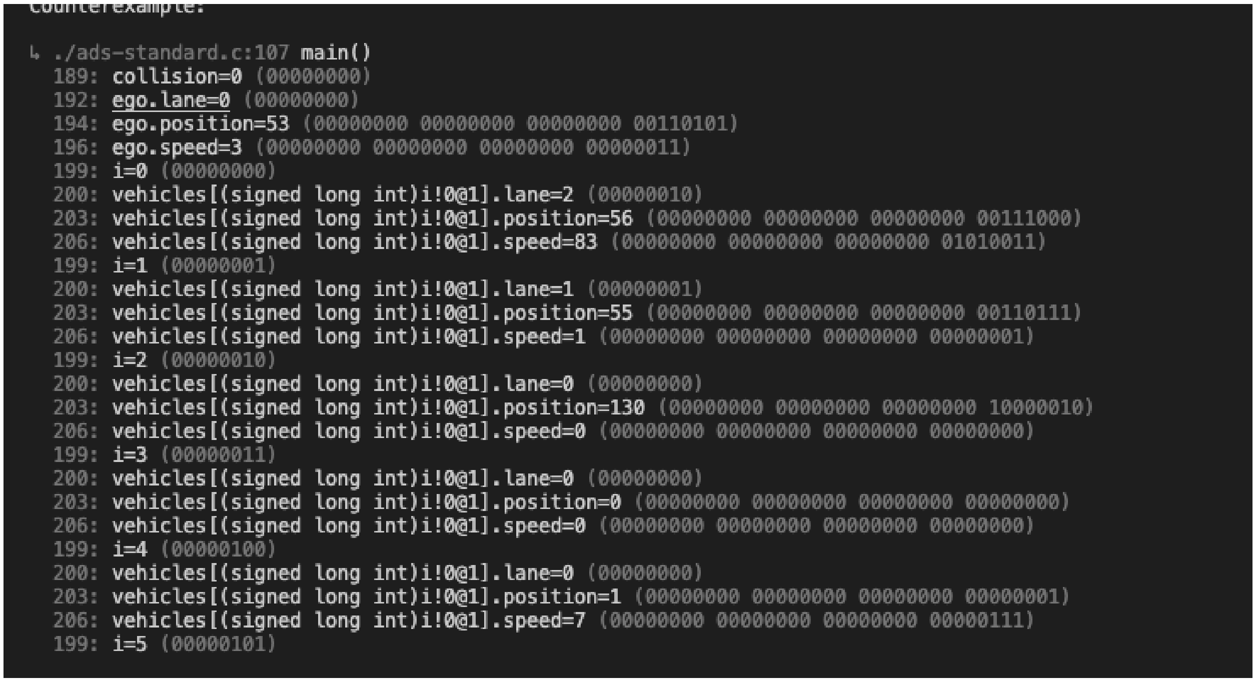
And then the result is following



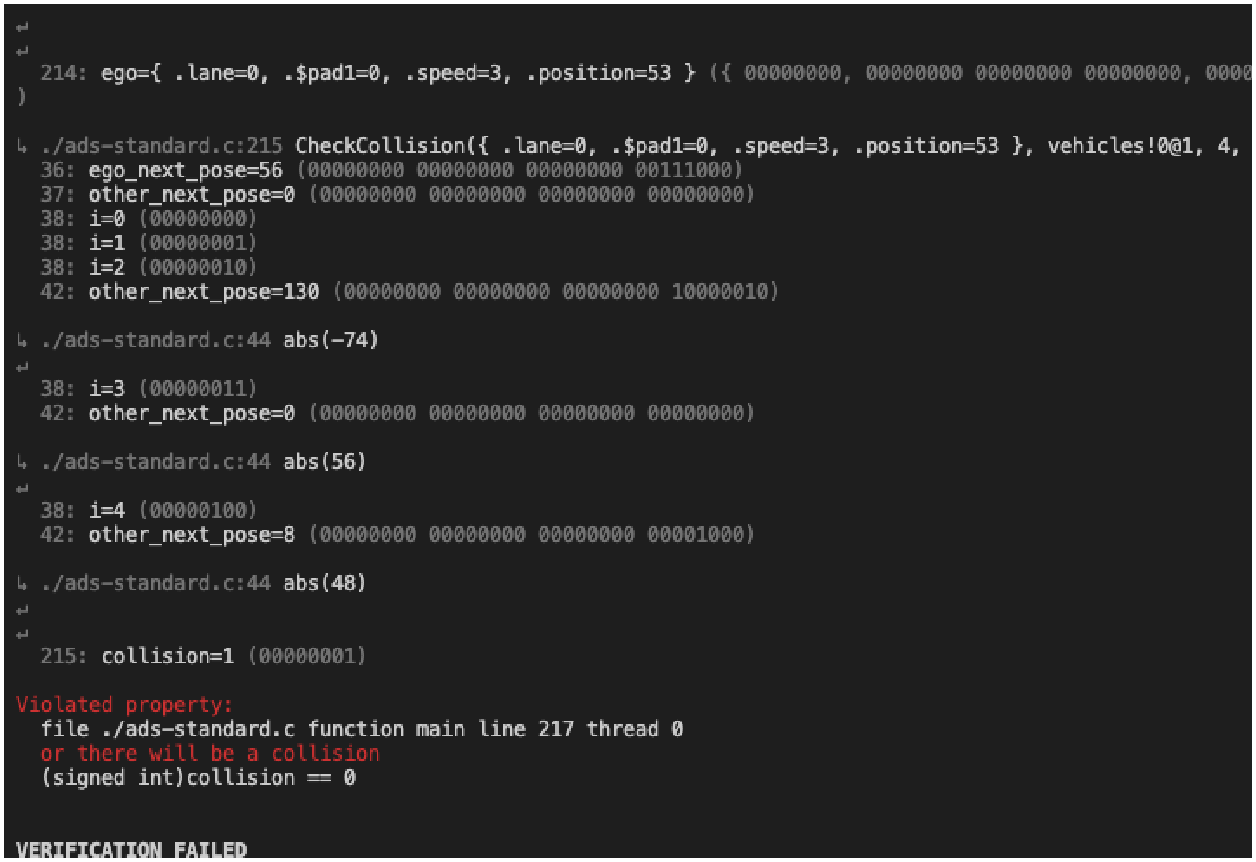
```
Not unwinding loop main.0 iteration 10 file ./ads-standard.c line 155 function main thread 0
size of program expression: 8929 steps
simple slicing removed 7 assignments
Generated 10 VCC(s), 10 remaining after simplification
Passing problem to propositional reduction
converting SSA
Running propositional reduction
Post-processing
Solving with MiniSAT 2.2.1 with simplifier
412793 variables, 1509957 clauses
SAT checker inconsistent: instance is UNSATISFIABLE
Runtime decision procedure: 2.23s
VERIFICATION SUCCESSFUL
khjtony@khjtony-Precision-M4800:~/workspace/cseee6863/final_project$
```

Intuitive model: The cases where ADS does not guarantee safety

There are a lot of cases where safety cannot be guaranteed for example if all the vehicles are just around ego vehicle:



```
Counterexample:

↳ ./ads-standard.c:107 main()
  189: collision=0 (00000000)
  192: ego.lane=0 (00000000)
  194: ego.position=53 (00000000 00000000 00000000 00110101)
  196: ego.speed=3 (00000000 00000000 00000000 00000011)
  199: i=0 (00000000)
  200: vehicles[(signed long int)i!0@1].lane=2 (00000010)
  203: vehicles[(signed long int)i!0@1].position=56 (00000000 00000000 00000000 00111000)
  206: vehicles[(signed long int)i!0@1].speed=83 (00000000 00000000 00000000 01010011)
  199: i=1 (00000001)
  200: vehicles[(signed long int)i!0@1].lane=1 (00000001)
  203: vehicles[(signed long int)i!0@1].position=55 (00000000 00000000 00000000 00110111)
  206: vehicles[(signed long int)i!0@1].speed=1 (00000000 00000000 00000000 00000001)
  199: i=2 (00000010)
  200: vehicles[(signed long int)i!0@1].lane=0 (00000000)
  203: vehicles[(signed long int)i!0@1].position=130 (00000000 00000000 00000000 10000010)
  206: vehicles[(signed long int)i!0@1].speed=0 (00000000 00000000 00000000 00000000)
  199: i=3 (00000011)
  200: vehicles[(signed long int)i!0@1].lane=0 (00000000)
  203: vehicles[(signed long int)i!0@1].position=0 (00000000 00000000 00000000 00000000)
  206: vehicles[(signed long int)i!0@1].speed=0 (00000000 00000000 00000000 00000000)
  199: i=4 (00000100)
  200: vehicles[(signed long int)i!0@1].lane=0 (00000000)
  203: vehicles[(signed long int)i!0@1].position=1 (00000000 00000000 00000000 00000001)
  206: vehicles[(signed long int)i!0@1].speed=7 (00000000 00000000 00000000 00000111)
  199: i=5 (00000101)
```

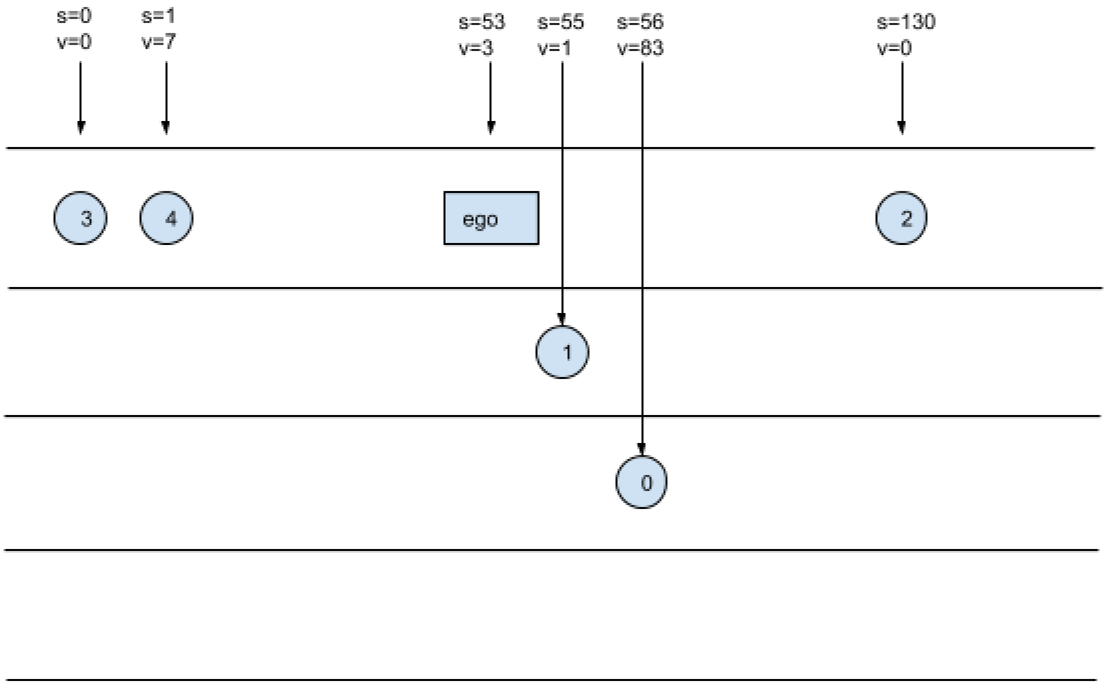


```
↵
↵
  214: ego={ .lane=0, .$pad1=0, .speed=3, .position=53 } ({ 00000000, 00000000 00000000 00000000, 0000
)

↳ ./ads-standard.c:215 CheckCollision({ .lane=0, .$pad1=0, .speed=3, .position=53 }, vehicles!0@1, 4,
  36: ego_next_pose=56 (00000000 00000000 00000000 00111000)
  37: other_next_pose=0 (00000000 00000000 00000000 00000000)
  38: i=0 (00000000)
  38: i=1 (00000001)
  38: i=2 (00000010)
  42: other_next_pose=130 (00000000 00000000 00000000 10000010)

↳ ./ads-standard.c:44 abs(-74)
↵
  38: i=3 (00000011)
  42: other_next_pose=0 (00000000 00000000 00000000 00000000)

↳ ./ads-standard.c:44 abs(56)
↵
  38: i=4 (00000100)
  42: other_next_pose=8 (00000000 00000000 00000000 00001000)

↳ ./ads-standard.c:44 abs(48)
↵
↵
  215: collision=1 (00000001)

Violated property:
  file ./ads-standard.c function main line 217 thread 0
  or there will be a collision
  (signed int)collision == 0


VERIFICATION FAILED
```

The cbmc provides a counterexample:

Vehicle	Lane	position	speed
ego	0	53	3
0	2	56	83
1	1	55	1
2	0	130	0
3	0	0	0
4	0	1	7

And we have the following road map:

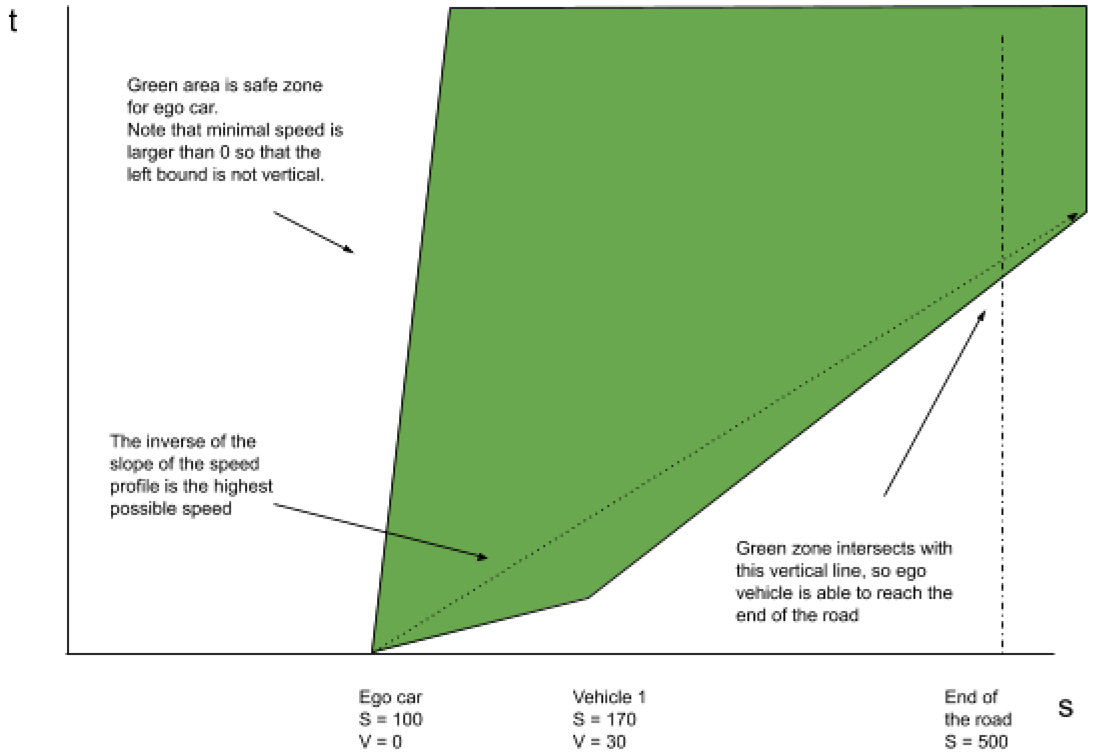


We can see that, ego car cannot change to left nor right, and if ego car stop and position equals to 53, then the next second vehicle No.4 will enter the minimal safety distance. If ego car move at the max speed, then the next second ego car will enter the minimal safety distance of vehicle No.2.

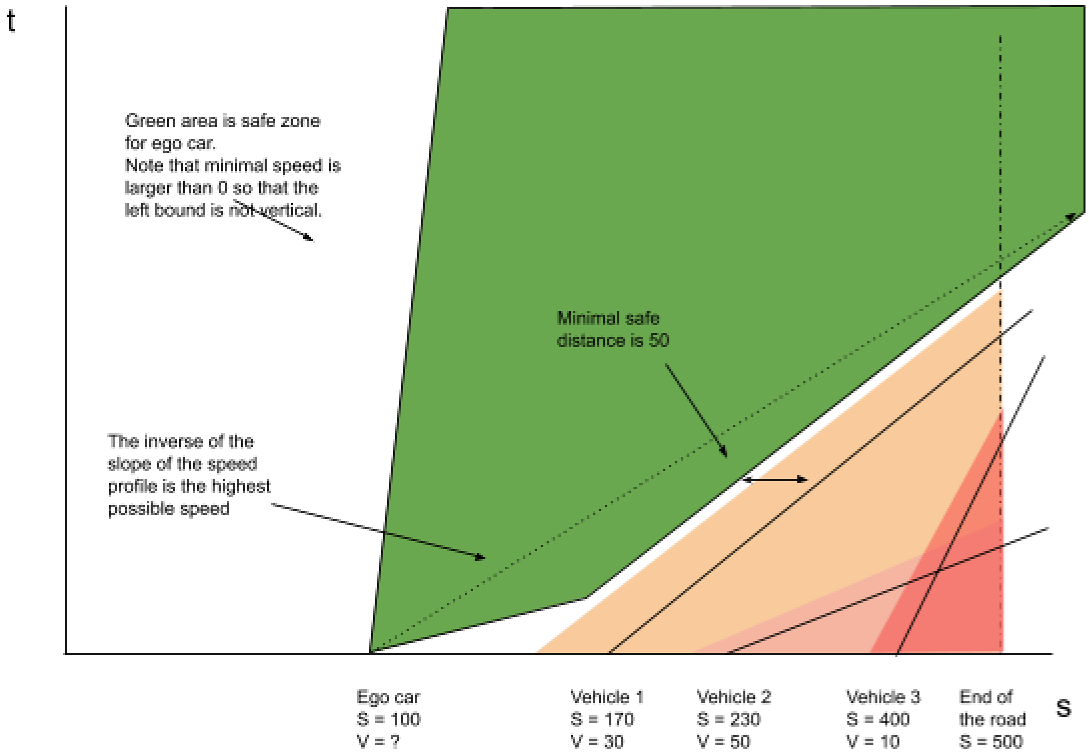
Complex Model: Cases where ADS guarantees safety

Because we did not change the problem setup dramatically, the cases where ADS guarantees safety in the Intuitive model will still be valid in the ADS Complex model, which we can analytically approve them using ST graph.
However due to the complexity of the software or maybe the wrong way of using CBMC tool, I was not able to verify this model. I will discuss more details in the Discussion section.

First case: Only ego vehicle is on the road


t
Green area is safe zone for ego car.
Note that minimal speed is larger than 0 so that the left bound is not vertical.
The inverse of the slope of the speed profile is the highest possible speed
Green zone intersects with this vertical line, so ego vehicle is able to reach the end of the road
Ego car
S = 100
V = 0
Vehicle 1
S = 170
V = 30
End of the road
S = 500
s

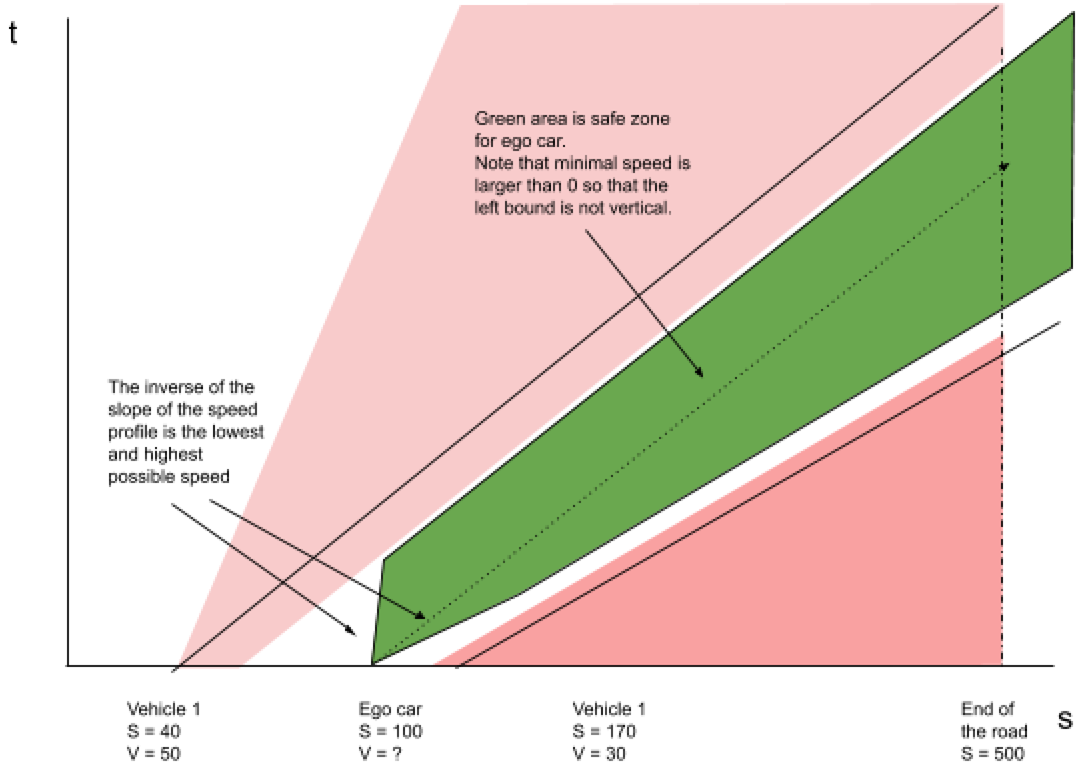
Second case: Only ego vehicle is on the road



t
Green area is safe zone for ego car.
Note that minimal speed is larger than 0 so that the left bound is not vertical.
Minimal safe distance is 50
The inverse of the slope of the speed profile is the highest possible speed
Ego car
S = 100
V = ?
Vehicle 1
S = 170
V = 30
Vehicle 2
S = 230
V = 50
Vehicle 3
S = 400
V = 10
End of the road
S = 500
S

In this graph, green is safe zone for ego car, and various red polygons are the zones occupied by Vehicle 1, vehicle 2, vehicle 3 respectively. Notice how the slopes changed relating to their speed.

Third case: Other vehicles do not move closer


t
Green area is safe zone for ego car. Note that minimal speed is larger than 0 so that the left bound is not vertical.
The inverse of the slope of the speed profile is the lowest and highest possible speed
Vehicle 1
S = 40
V = 50
Ego car
S = 100
V = ?
Vehicle 1
S = 170
V = 30
End of the road
S = 500
s

In this case, two red zones will never have an intersection above the s-axis, which is t > 0.

Complex Model: Cases where ADS does not guarantee safety

In the complex model, ego vehicle has freedom to arbitrarily choose speed as the next motion, so ego car has more flexibility to maneuver on the road. However since other parts of the problem setup did not change, the boundary of the green zone shown in the ST Graph does not change, but just the speed profile we can change. In this context, cases where ADS does not guarantee safety in the Intuitive model will also apply to the ADS complex model.

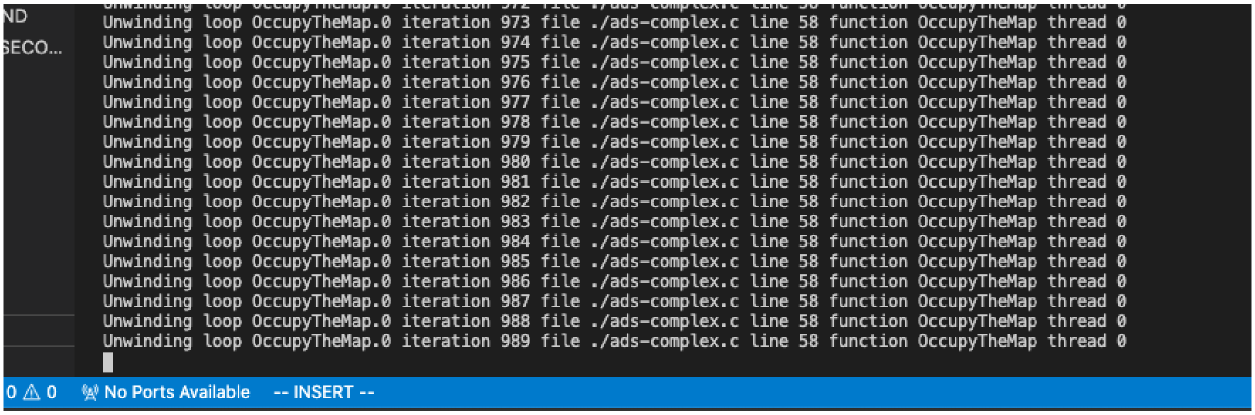
Discussion

Arrays dramatically increased complexity for CBMC

A function that CBMC is not able to check in a limited time.
ADS-complex model is relatively more complex, which has search algorithm, so I used CBMC to check the function “OccupyTheMap” using the following command:

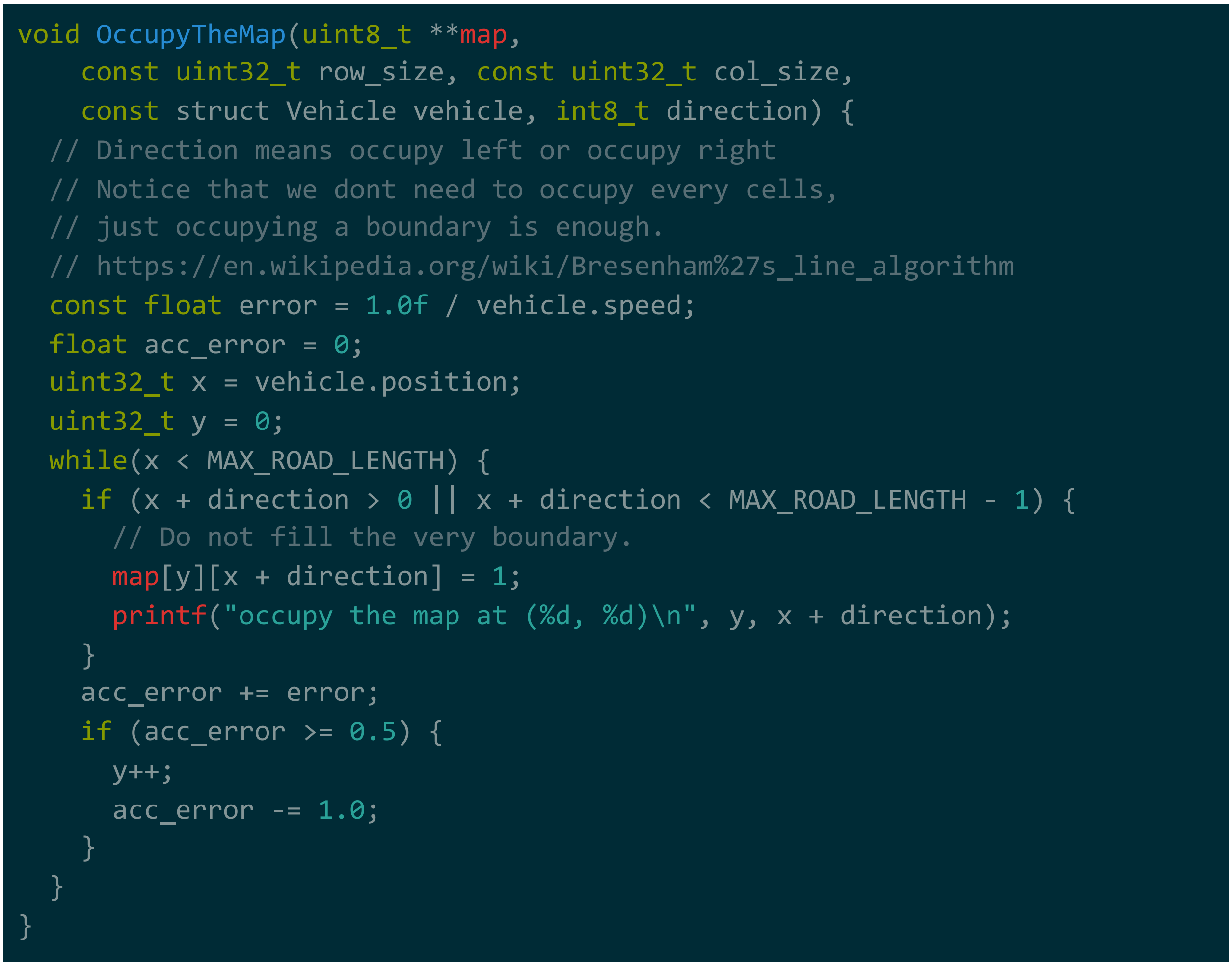
```
cbmc ./ads-complex.c --bounds-check --pointer-check --compact-trace --stop-on-fail --function OccupyTheMap
```

However it takes forever to run:



```
Unwinding loop OccupyTheMap.0 iteration 972 file ./ads-complex.c line 58 function OccupyTheMap thread 0
Unwinding loop OccupyTheMap.0 iteration 973 file ./ads-complex.c line 58 function OccupyTheMap thread 0
Unwinding loop OccupyTheMap.0 iteration 974 file ./ads-complex.c line 58 function OccupyTheMap thread 0
Unwinding loop OccupyTheMap.0 iteration 975 file ./ads-complex.c line 58 function OccupyTheMap thread 0
Unwinding loop OccupyTheMap.0 iteration 976 file ./ads-complex.c line 58 function OccupyTheMap thread 0
Unwinding loop OccupyTheMap.0 iteration 977 file ./ads-complex.c line 58 function OccupyTheMap thread 0
Unwinding loop OccupyTheMap.0 iteration 978 file ./ads-complex.c line 58 function OccupyTheMap thread 0
Unwinding loop OccupyTheMap.0 iteration 979 file ./ads-complex.c line 58 function OccupyTheMap thread 0
Unwinding loop OccupyTheMap.0 iteration 980 file ./ads-complex.c line 58 function OccupyTheMap thread 0
Unwinding loop OccupyTheMap.0 iteration 981 file ./ads-complex.c line 58 function OccupyTheMap thread 0
Unwinding loop OccupyTheMap.0 iteration 982 file ./ads-complex.c line 58 function OccupyTheMap thread 0
Unwinding loop OccupyTheMap.0 iteration 983 file ./ads-complex.c line 58 function OccupyTheMap thread 0
Unwinding loop OccupyTheMap.0 iteration 984 file ./ads-complex.c line 58 function OccupyTheMap thread 0
Unwinding loop OccupyTheMap.0 iteration 985 file ./ads-complex.c line 58 function OccupyTheMap thread 0
Unwinding loop OccupyTheMap.0 iteration 986 file ./ads-complex.c line 58 function OccupyTheMap thread 0
Unwinding loop OccupyTheMap.0 iteration 987 file ./ads-complex.c line 58 function OccupyTheMap thread 0
Unwinding loop OccupyTheMap.0 iteration 988 file ./ads-complex.c line 58 function OccupyTheMap thread 0
Unwinding loop OccupyTheMap.0 iteration 989 file ./ads-complex.c line 58 function OccupyTheMap thread 0
```

Go back and check the function itself, we found that this function does not have the knowledge of pointer unless the input parameters provide that (row_size, col_size):



```c
void OccupyTheMap(uint8_t **map,
    const uint32_t row_size, const uint32_t col_size,
    const struct Vehicle vehicle, int8_t direction) {
  // Direction means occupy left or occupy right
  // Notice that we dont need to occupy every cells,
  // just occupying a boundary is enough.
  // https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm
  const float error = 1.0f / vehicle.speed;
  float acc_error = 0;
  uint32_t x = vehicle.position;
  uint32_t y = 0;
  while(x < MAX_ROAD_LENGTH) {
    if (x + direction > 0 || x + direction < MAX_ROAD_LENGTH - 1) {
      // Do not fill the very boundary.
      map[y][x + direction] = 1;
      printf("occupy the map at (%d, %d)\n", y, x + direction);
    }
    acc_error += error;
    if (acc_error >= 0.5) {
      y++;
      acc_error -= 1.0;
    }
  }
}
```

So it is understandable that this function is hard to check. The “printf” is added to help me find an actual segmentation fault issue, which was not caught by CBMC during run time and CBMC still gives “Successful” results.

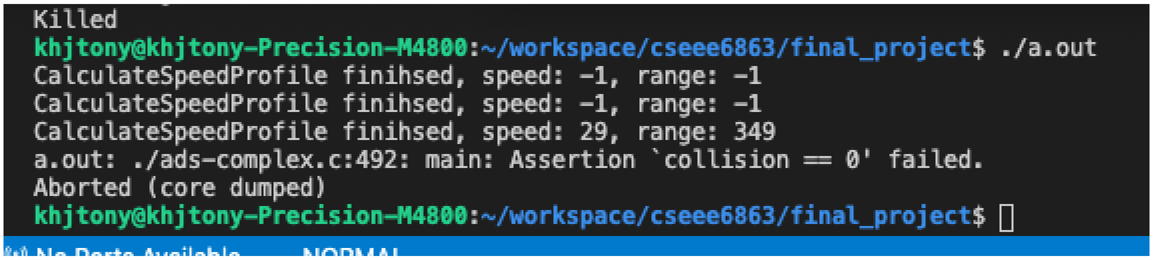
After using maybe best debugging method -- “printf”, I corrected this function to the following:

```
void OccupyTheMap(uint8_t **map,
    const uint32_t row_size, const uint32_t col_size,
    const struct Vehicle vehicle, int8_t direction) {
  // Direction means occupy left or occupy right
  // Notice that we dont need to occupy every cells,
  // just occupying a boundary is enough.
  // https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm
  const float error = 1.0f / vehicle.speed;
  float acc_error = 0;
  uint32_t s = vehicle.position;
  uint32_t t = 0;
  for (uint32_t s = vehicle.position + direction; s < col_size; s++) {
    if (s >= MAX_ROAD_LENGTH - 1) {
      break;
    } else {
      map[row_size - 1 - t][s] = 1;
      // printf("[%d] occupy the map at (%d, %d)\n", t, row_size - 1 - t,
s);
    }
    acc_error += error;
    if (acc_error >= 0.5) {
      t++;
      acc_error -= 1.0;
    }
  }
}
```

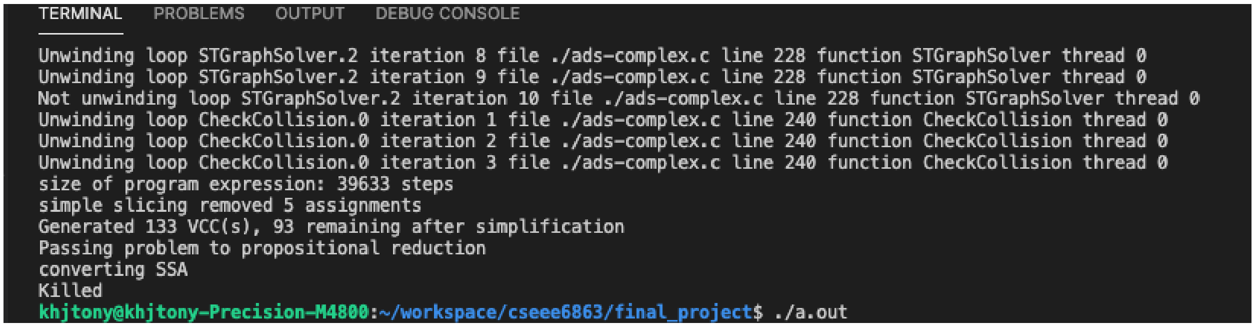
But cbmc still cannot provide a conclusion in the reasonable time.

Unwind option matters a lot

In a more complex case when I want to use CBMC to check the whole program after can already run it with some sample initial setups and gives looked correct results, CMBC was eventually terminated by “Killed”. I think that it maybe because of too many status:



```
Killed
khjtony@khjtony-Precision-M4800:~/workspace/cseee6863/final_project$ ./a.out
CalculateSpeedProfile finihsed, speed: -1, range: -1
CalculateSpeedProfile finihsed, speed: -1, range: -1
CalculateSpeedProfile finihsed, speed: 29, range: 349
a.out: ./ads-complex.c:492: main: Assertion `collision == 0' failed.
Aborted (core dumped)
khjtony@khjtony-Precision-M4800:~/workspace/cseee6863/final_project$ 
```

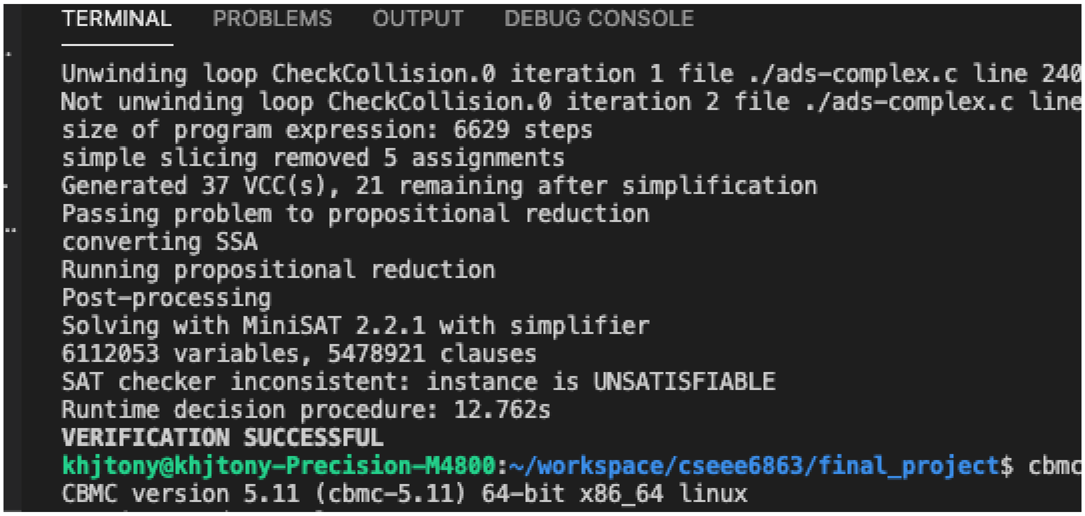


```
Unwinding loop STGraphSolver.2 iteration 8 file ./ads-complex.c line 228 function STGraphSolver thread 0
Unwinding loop STGraphSolver.2 iteration 9 file ./ads-complex.c line 228 function STGraphSolver thread 0
Not unwinding loop STGraphSolver.2 iteration 10 file ./ads-complex.c line 228 function STGraphSolver thread 0
Unwinding loop CheckCollision.0 iteration 1 file ./ads-complex.c line 240 function CheckCollision thread 0
Unwinding loop CheckCollision.0 iteration 2 file ./ads-complex.c line 240 function CheckCollision thread 0
Unwinding loop CheckCollision.0 iteration 3 file ./ads-complex.c line 240 function CheckCollision thread 0
size of program expression: 39633 steps
simple slicing removed 5 assignments
Generated 133 VCC(s), 93 remaining after simplification
Passing problem to propositional reduction
converting SSA
Killed
khjtony@khjtony-Precision-M4800:~/workspace/cseee6863/final_project$ ./a.out
```

The first image shows that the program is runnable.

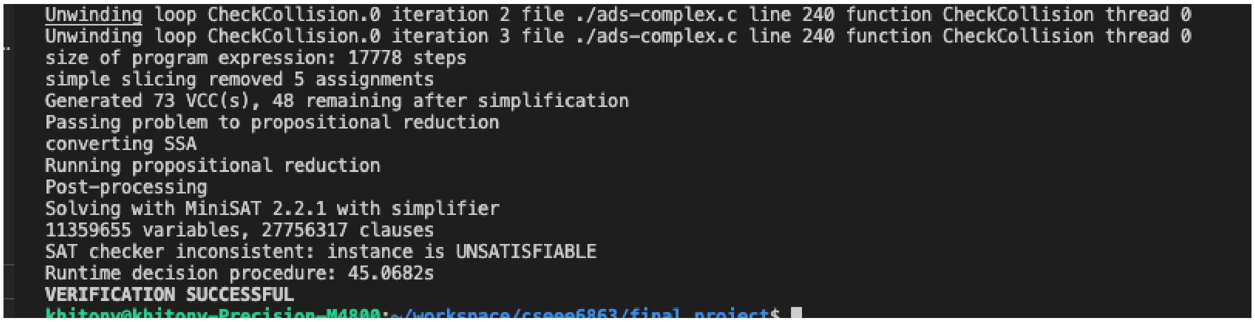
The second image shows that the CMBC was killed when I used option “--unwind 10”

If I use “--unwind 2”, it seems that CBMC was not able to search deep enough to find a designed collision.



```
Unwinding loop CheckCollision.0 iteration 1 file ./ads-complex.c line 240
Not unwinding loop CheckCollision.0 iteration 2 file ./ads-complex.c line
size of program expression: 6629 steps
simple slicing removed 5 assignments
Generated 37 VCC(s), 21 remaining after simplification
Passing problem to propositional reduction
converting SSA
Running propositional reduction
Post-processing
Solving with MiniSAT 2.2.1 with simplifier
6112053 variables, 5478921 clauses
SAT checker inconsistent: instance is UNSATISFIABLE
Runtime decision procedure: 12.762s
VERIFICATION SUCCESSFUL
khjtony@khjtony-Precision-M4800:~/workspace/cseee6863/final_project$ cbmc
CBMC version 5.11 (cbmc-5.11) 64-bit x86_64 linux
```

If I use “--unwind 5”, it seems not enough as well:



```
Unwinding loop CheckCollision.0 iteration 2 file ./ads-complex.c line 240 function CheckCollision thread 0
Unwinding loop CheckCollision.0 iteration 3 file ./ads-complex.c line 240 function CheckCollision thread 0
size of program expression: 17778 steps
simple slicing removed 5 assignments
Generated 73 VCC(s), 48 remaining after simplification
Passing problem to propositional reduction
converting SSA
Running propositional reduction
Post-processing
Solving with MiniSAT 2.2.1 with simplifier
11359655 variables, 27756317 clauses
SAT checker inconsistent: instance is UNSATISFIABLE
Runtime decision procedure: 45.0682s
VERIFICATION SUCCESSFUL
khjtony@khjtony-Precision-M4800:~/workspace/cseee6863/final_project$
```

Appendix I: ADS Intuitive model code

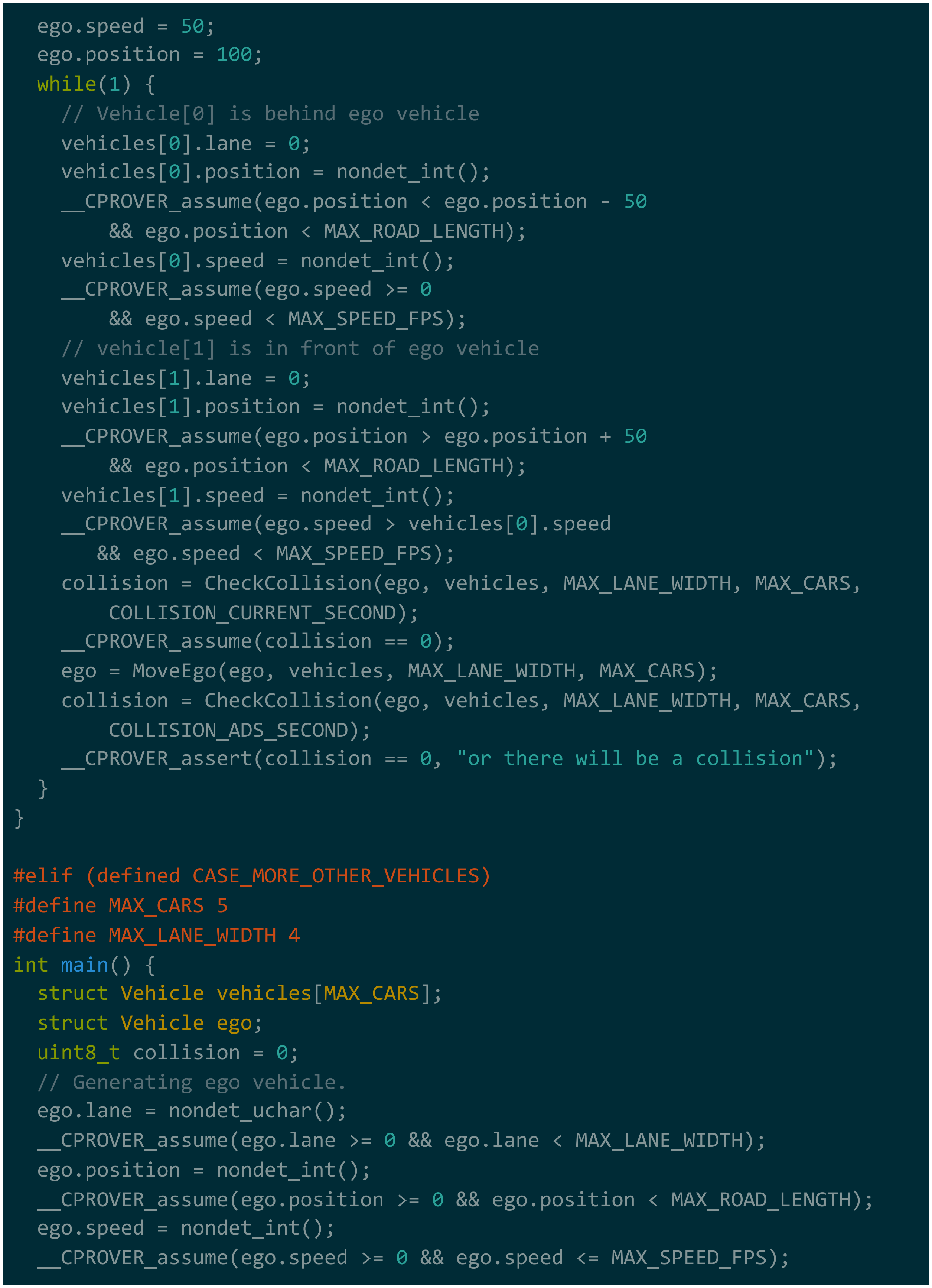
```
#include <assert.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
```

```c
#include <stdlib.h>

#define MIN_SPEED 0
// Max speed is in ft/s
#define MAX_SPEED_FPS 88
// Minimum safe distance is 50ft
#define MIN_DISTANCE_FT 50
// Max road length is 500 ft
#define MAX_ROAD_LENGTH 500
// System check collision in next second
#define COLLISION_CURRENT_SECOND 0
// ADS check sollision after two seconds
#define COLLISION_ADS_SECOND 1

#define CASE_ONLY_ME
// #define CASE_ONE_SIDE
// #define CASE_BOTH_SIDE
// #define CASE_MORE_OTHER_VEHICLES
// #define CASE_TEST_COLLISION

int nondet_int();
uint8_t nondet_uchar();

struct Vehicle {
  uint8_t lane;
  int32_t speed;
  int32_t position;
};

uint8_t CheckCollision(const struct Vehicle ego,
    const struct Vehicle* vehicles, const uint8_t max_lanes,
    const uint8_t max_cars, const int32_t seconds) {
  int32_t ego_next_pose = ego.position + ego.speed * seconds;
  int32_t other_next_pose = 0;
  for (uint8_t i = 0; i < max_cars; i++) {
    if (ego.lane != vehicles[i].lane) {
      continue;
    }
    other_next_pose = vehicles[i].position
      + vehicles[i].speed * seconds;
    if (abs(ego_next_pose - other_next_pose) >= MIN_DISTANCE_FT &&
        (ego.position - vehicles[i].position) *
        (ego_next_pose - other_next_pose) > 0) {
```

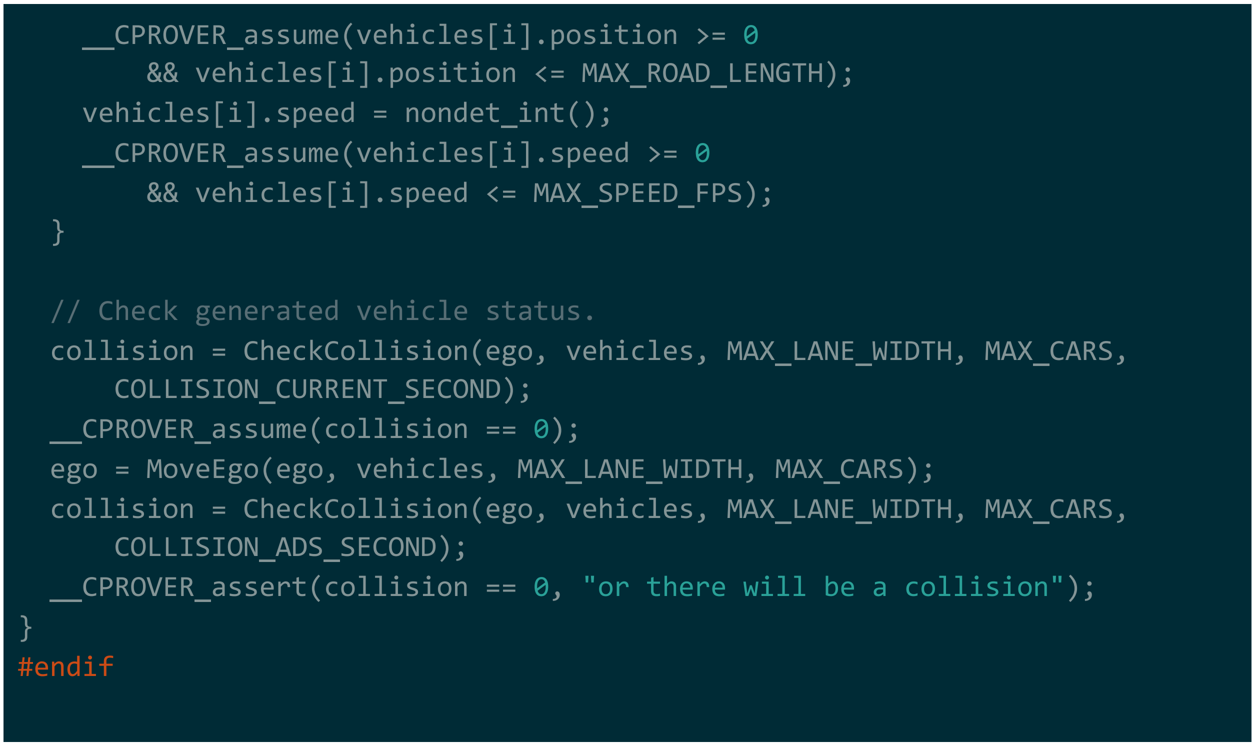
```
      continue;
    } else {
      return 1;
    }
  }
  return 0;
}

struct Vehicle EgoMaxSpeed(const struct Vehicle ego) {
  struct Vehicle local_copy = ego;
  local_copy.speed = MAX_SPEED_FPS;
  return local_copy;
}

struct Vehicle EgoStop(const struct Vehicle ego) {
  struct Vehicle local_copy = ego;
  local_copy.speed = 0;
  return local_copy;
}

struct Vehicle ChangeEgoLaneLeft(const struct Vehicle ego) {
  struct Vehicle local_copy = ego;
  if (local_copy.lane > 0) {
    local_copy.lane--;
  }
  return local_copy;
}

struct Vehicle ChangeEgoLaneRight(const struct Vehicle ego,
    const uint8_t max_lanes) {
  struct Vehicle local_copy = ego;
  if (local_copy.lane < max_lanes - 1) {
    local_copy.lane++;
  }
  return local_copy;
}

struct Vehicle MoveEgo(
    const struct Vehicle ego, const struct Vehicle* vehicles,
    const uint8_t max_lanes, const uint8_t max_cars) {
  if (CheckCollision(
      EgoMaxSpeed(ego), vehicles, max_lanes, max_cars,
      COLLISION_ADS_SECOND) == 0) {
```

```
    return EgoMaxSpeed(ego);
  } else if (CheckCollision(
      EgoStop(ego), vehicles, max_lanes, max_cars,
      COLLISION_ADS_SECOND) == 0) {
    return EgoStop(ego);
  } else if (CheckCollision(
      ChangeEgoLaneLeft(ego), vehicles, max_lanes, max_cars,
      COLLISION_ADS_SECOND) == 0) {
    return ChangeEgoLaneLeft(ego);
  } else if (CheckCollision(
      ChangeEgoLaneRight(ego, max_lanes), vehicles, max_lanes, max_cars,
      COLLISION_ADS_SECOND) == 0) {
    return ChangeEgoLaneRight(ego, max_lanes);
  }
  return ego;
}

#if defined CASE_ONLY_ME
#define MAX_CARS 0
#define MAX_LANE_WIDTH 4
int main() {
  struct Vehicle vehicles[MAX_CARS];
  struct Vehicle ego;
  uint8_t collision = 0;

  // Generating ego vehicle.
  ego.lane = nondet_uchar();
  __CPROVER_assume(ego.lane >= 0 && ego.lane < MAX_LANE_WIDTH);
  ego.position = 0;
  ego.speed = nondet_int();
  __CPROVER_assume(ego.speed >= 0
      && ego.speed <= MAX_SPEED_FPS);
  // Check generated vehicle status.
  collision = CheckCollision(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS,
      COLLISION_CURRENT_SECOND);
  __CPROVER_assume(collision == 0);
  ego = MoveEgo(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS);
  collision = CheckCollision(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS,
      COLLISION_ADS_SECOND);
  __CPROVER_assert(collision == 0, "or there will be a collision");
}

#elif (defined CASE_ONE_SIDE)
```

```
#define OTHER_MIN_POSITION 100
#define MAX_CARS 5
#define MAX_LANE_WIDTH 4
int main() {
  struct Vehicle vehicles[MAX_CARS];
  struct Vehicle ego;
  uint8_t collision = 0;

  // Generating ego vehicle.
  ego.lane = nondet_uchar();
  __CPROVER_assume(ego.lane >= 0 && ego.lane < MAX_LANE_WIDTH);
  ego.position = 0;
  ego.speed = nondet_int();
  __CPROVER_assume(ego.speed >= 0
      && ego.speed <= MAX_SPEED_FPS);
  // Generating other vehicles.
  for (uint8_t i = 0; i < MAX_CARS; i++) {
    vehicles[i].lane = ego.lane;
    vehicles[i].position = nondet_int();
    __CPROVER_assume(vehicles[i].position >= OTHER_MIN_POSITION
        && vehicles[i].position <= MAX_ROAD_LENGTH);
    vehicles[i].speed = nondet_int();
    __CPROVER_assume(vehicles[i].speed >= 0
        && vehicles[i].speed <= MAX_SPEED_FPS);
  }
  // Check generated vehicle status.
  collision = CheckCollision(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS,
      COLLISION_CURRENT_SECOND);
  __CPROVER_assume(collision == 0);
  ego = MoveEgo(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS);
  collision = CheckCollision(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS,
      COLLISION_ADS_SECOND);
  __CPROVER_assert(collision == 0, "or there will be a collision");
}

#elif (defined CASE_BOTH_SIDE)
#define MAX_CARS 2
#define MAX_LANE_WIDTH 4
int main() {
  struct Vehicle vehicles[MAX_CARS];
  struct Vehicle ego;
  uint8_t collision = 0;
  ego.lane = 0;
```

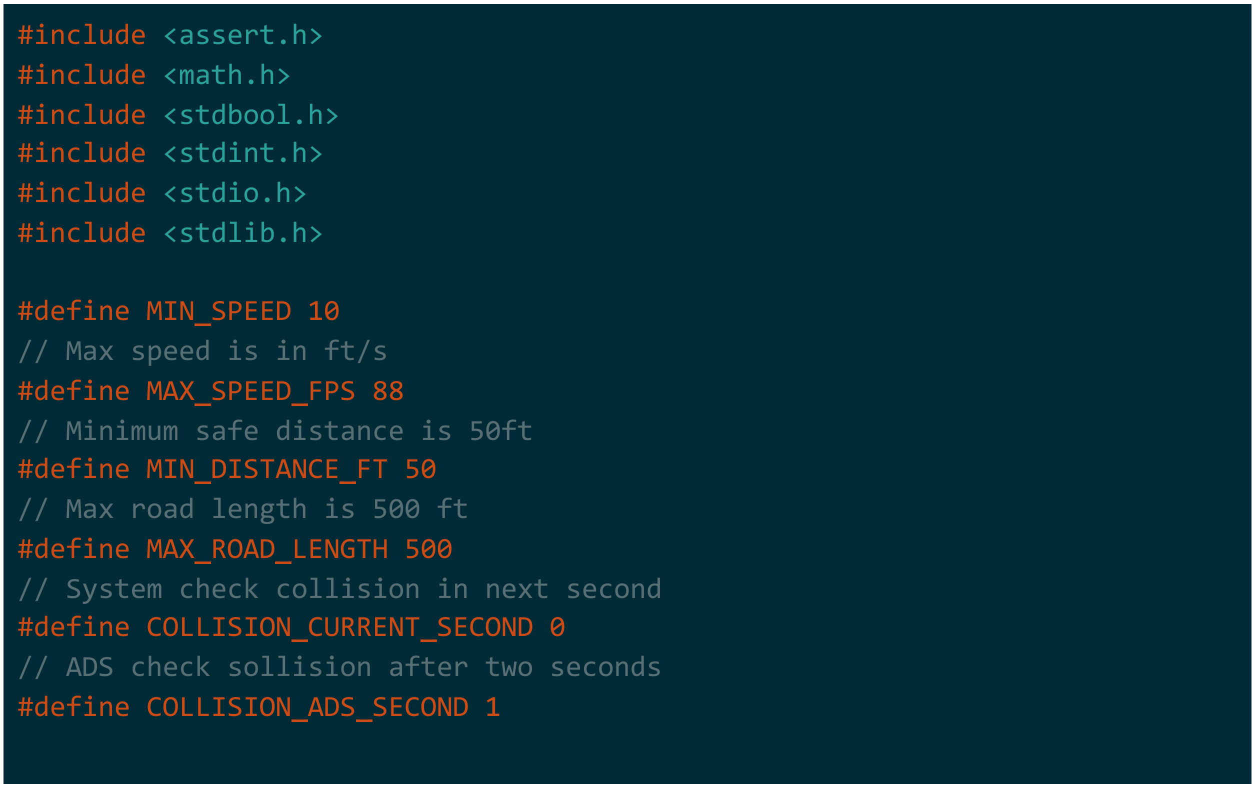


```c
  ego.speed = 50;
  ego.position = 100;
  while(1) {
    // Vehicle[0] is behind ego vehicle
    vehicles[0].lane = 0;
    vehicles[0].position = nondet_int();
    __CPROVER_assume(ego.position < ego.position - 50
        && ego.position < MAX_ROAD_LENGTH);
    vehicles[0].speed = nondet_int();
    __CPROVER_assume(ego.speed >= 0
        && ego.speed < MAX_SPEED_FPS);
    // vehicle[1] is in front of ego vehicle
    vehicles[1].lane = 0;
    vehicles[1].position = nondet_int();
    __CPROVER_assume(ego.position > ego.position + 50
        && ego.position < MAX_ROAD_LENGTH);
    vehicles[1].speed = nondet_int();
    __CPROVER_assume(ego.speed > vehicles[0].speed
       && ego.speed < MAX_SPEED_FPS);
    collision = CheckCollision(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS,
        COLLISION_CURRENT_SECOND);
    __CPROVER_assume(collision == 0);
    ego = MoveEgo(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS);
    collision = CheckCollision(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS,
        COLLISION_ADS_SECOND);
    __CPROVER_assert(collision == 0, "or there will be a collision");
  }
}

#elif (defined CASE_MORE_OTHER_VEHICLES)
#define MAX_CARS 5
#define MAX_LANE_WIDTH 4
int main() {
  struct Vehicle vehicles[MAX_CARS];
  struct Vehicle ego;
  uint8_t collision = 0;
  // Generating ego vehicle.
  ego.lane = nondet_uchar();
  __CPROVER_assume(ego.lane >= 0 && ego.lane < MAX_LANE_WIDTH);
  ego.position = nondet_int();
  __CPROVER_assume(ego.position >= 0 && ego.position < MAX_ROAD_LENGTH);
  ego.speed = nondet_int();
  __CPROVER_assume(ego.speed >= 0 && ego.speed <= MAX_SPEED_FPS);
  // Generating other vehicles.
  for (uint8_t i = 0; i < MAX_CARS; i++) {
    vehicles[i].lane = nondet_uchar();
    __CPROVER_assume(vehicles[i].lane >= 0
        && vehicles[i].lane < MAX_LANE_WIDTH);
    vehicles[i].position = nondet_int();
    __CPROVER_assume(vehicles[i].position >= 0
        && vehicles[i].position <= MAX_ROAD_LENGTH);
    vehicles[i].speed = nondet_int();
    __CPROVER_assume(vehicles[i].speed >= 0
        && vehicles[i].speed <= MAX_SPEED_FPS);
  }
  // Check generated vehicle status.
  collision = CheckCollision(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS,
      COLLISION_CURRENT_SECOND);
  __CPROVER_assume(collision == 0);
  ego = MoveEgo(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS);
  collision = CheckCollision(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS,
      COLLISION_ADS_SECOND);
  __CPROVER_assert(collision == 0, "or there will be a collision");
}

#elif (defined CASE_TEST_COLLISION)
#define MAX_CARS 3
#define MAX_LANE_WIDTH 4
int main() {
  struct Vehicle vehicles[MAX_CARS];
  struct Vehicle ego;
  uint8_t collision = 0;
  // Generating ego vehicle.
  ego.lane = nondet_uchar();
  __CPROVER_assume(ego.lane >= 0 && ego.lane < MAX_LANE_WIDTH);
  ego.position = nondet_int();
  __CPROVER_assume(ego.position >= 0 && ego.position < MAX_ROAD_LENGTH);
  ego.speed = nondet_int();
  __CPROVER_assume(ego.speed >= 0 && ego.speed <= MAX_SPEED_FPS);

  // Generating other vehicles.
  for (uint8_t i = 0; i < MAX_CARS; i++) {
    vehicles[i].lane = nondet_uchar();
    __CPROVER_assume(vehicles[i].lane >= 0
        && vehicles[i].lane < MAX_LANE_WIDTH);
    vehicles[i].position = nondet_int();
```



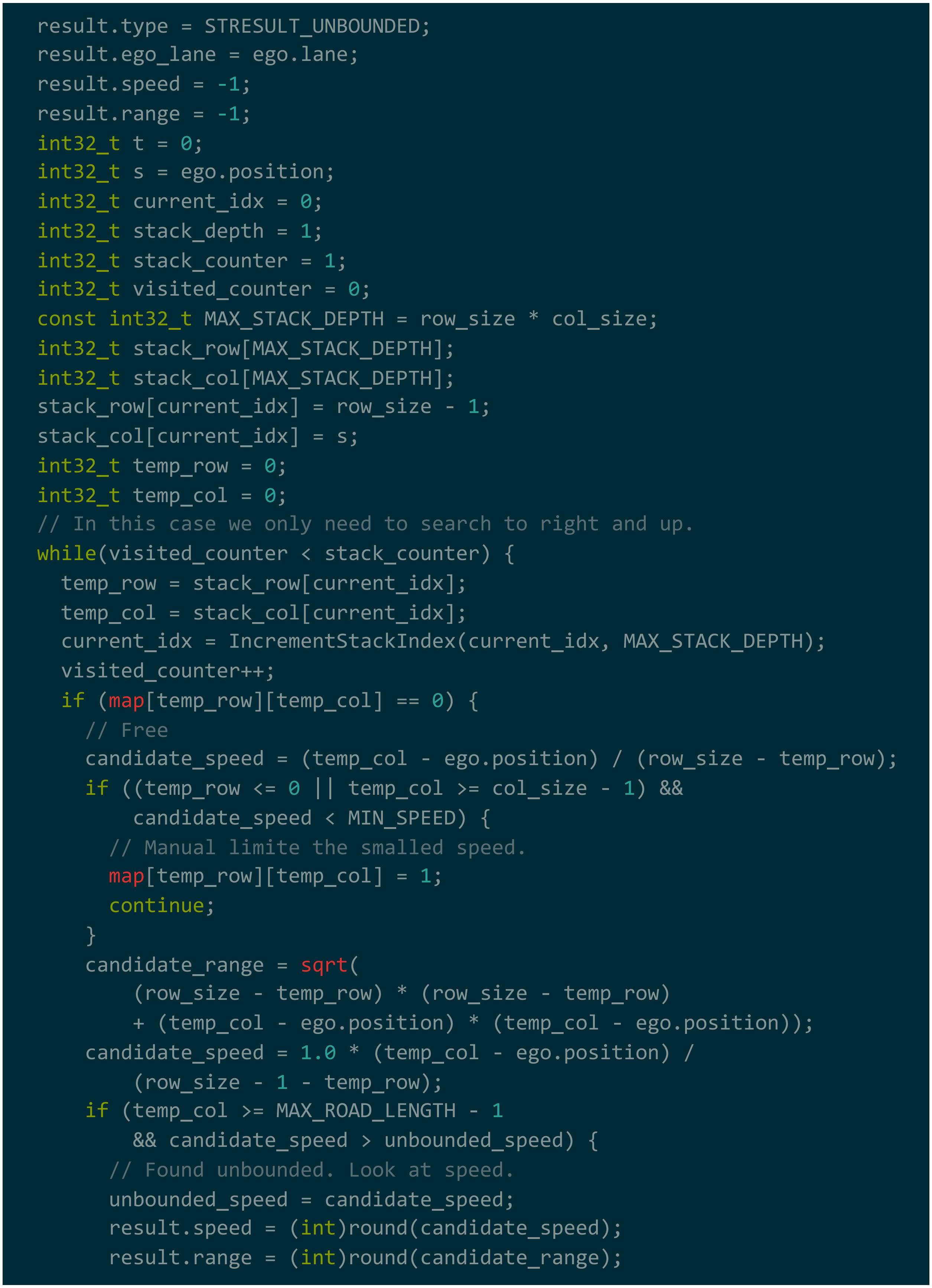
```
    __CPROVER_assume(vehicles[i].position >= 0
        && vehicles[i].position <= MAX_ROAD_LENGTH);
    vehicles[i].speed = nondet_int();
    __CPROVER_assume(vehicles[i].speed >= 0
        && vehicles[i].speed <= MAX_SPEED_FPS);
  }

  // Check generated vehicle status.
  collision = CheckCollision(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS,
      COLLISION_CURRENT_SECOND);
  __CPROVER_assume(collision == 0);
  ego = MoveEgo(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS);
  collision = CheckCollision(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS,
      COLLISION_ADS_SECOND);
  __CPROVER_assert(collision == 0, "or there will be a collision");
}
#endif
```

Appendix II: ADS Complex model code



```
#include <assert.h>
#include <math.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

#define MIN_SPEED 10
// Max speed is in ft/s
#define MAX_SPEED_FPS 88
// Minimum safe distance is 50ft
#define MIN_DISTANCE_FT 50
// Max road length is 500 ft
#define MAX_ROAD_LENGTH 500
// System check collision in next second
#define COLLISION_CURRENT_SECOND 0
// ADS check sollision after two seconds
#define COLLISION_ADS_SECOND 1
```

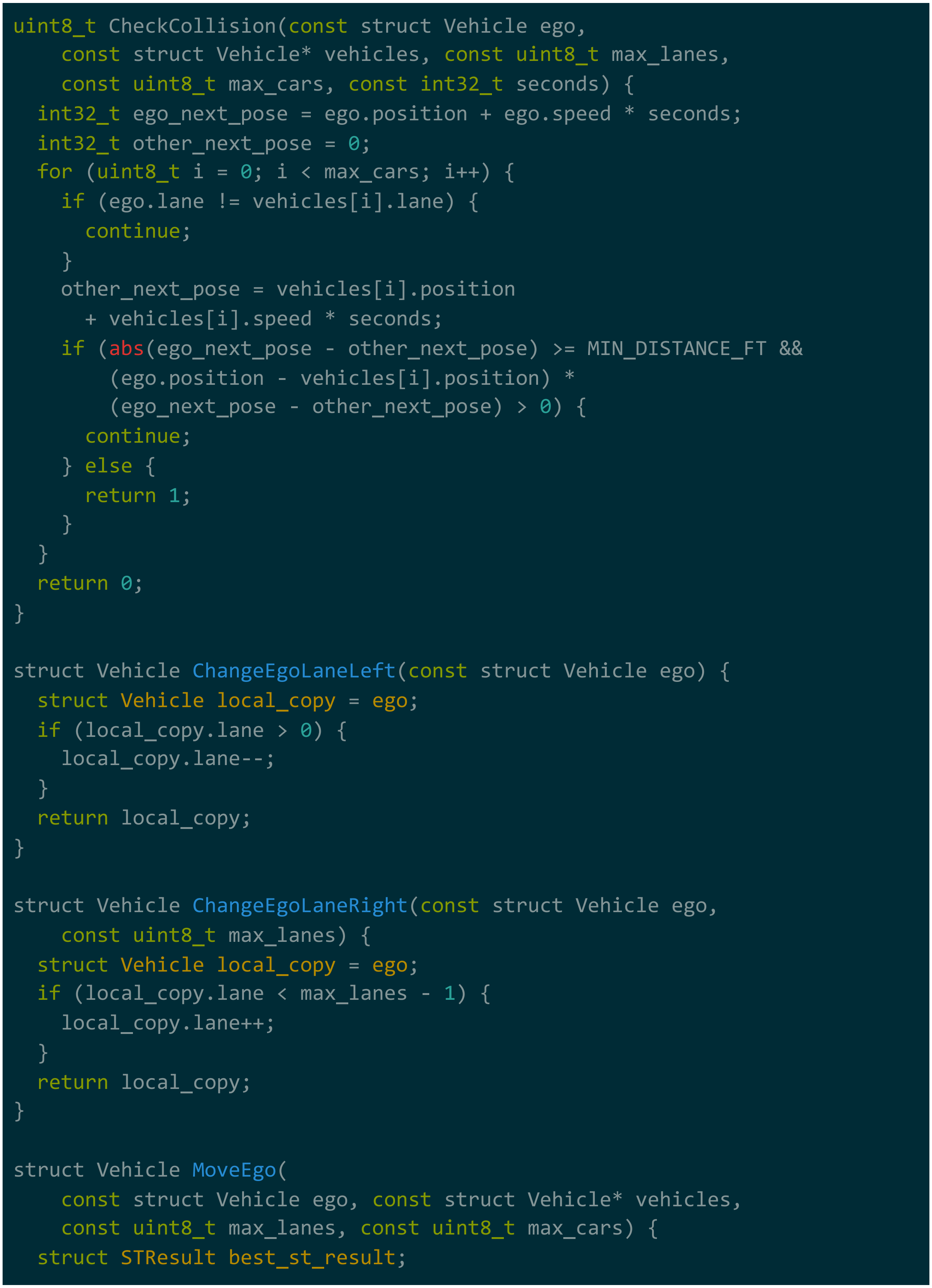
```c
#define CBMC
// #define GNUC

// #define CASE_ONLY_ME
// #define CASE_ONE_SIDE
// #define CASE_BOTH_SIDE
// #define CASE_MORE_OTHER_VEHICLES
#define CASE_TEST_COLLISION

#if (defined CBMC)
int nondet_int();
uint8_t nondet_uchar();
#endif  // defined CBMC

enum STResultType {
  STRESULT_BOUNDED = 0,
  STRESULT_UNBOUNDED = 1
};

struct Vehicle {
  uint8_t lane;
  int32_t speed;
  int32_t position;
};

struct STResult {
  // If safe zone is bounded, then speed_score = speed.
  // If safe zone is unbounded, then speed_score = MAX_SPEED_FPS + speed
  enum STResultType type;
  uint8_t ego_lane;
  int32_t speed;
  int32_t range;
};

uint32_t IncrementStackIndex(
    const uint32_t idx, const uint32_t depth_limit) {
  return idx + 1 >= depth_limit ? 0 : idx + 1;
}

struct STResult CompareSTResult(const struct STResult r1,
    const struct STResult r2) {
  if (r1.type != r2.type) {
    return r1.type > r2.type? r1 : r2;
  }
  if (r1.type == STRESULT_BOUNDED) {
    return r1.range > r2.range? r1 : r2;
  }

  if (r1.type == STRESULT_UNBOUNDED) {
    return r1.speed > r2.speed ? r1 : r2;
  }
}

void OccupyTheMap(uint8_t **map,
    const uint32_t row_size, const uint32_t col_size,
    const struct Vehicle vehicle, int8_t direction) {
  // Direction means occupy left or occupy right
  // Notice that we dont need to occupy every cells,
  // just occupying a boundary is enough.
  // https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm
  const float error = 1.0f / vehicle.speed;
  float acc_error = 0;
  uint32_t s = vehicle.position;
  uint32_t t = 0;
  for (uint32_t s = vehicle.position + direction; s < col_size; s++) {
    if (s >= MAX_ROAD_LENGTH - 1) {
      break;
    } else {
      map[row_size - 1 - t][s] = 1;
    }
    acc_error += error;
    if (acc_error >= 0.5) {
      t++;
      acc_error -= 1.0;
    }
  }
}

struct STResult CalculateSpeedProfile(uint8_t **map,
    const uint32_t row_size, const uint32_t col_size,
    const struct Vehicle ego) {
  float candidate_range = -1;
  float candidate_speed = -1;
  float unbounded_speed = -1;
  float bounded_range = -1;
  struct STResult result;
```



```
result.type = STRESULT_UNBOUNDED;
result.ego_lane = ego.lane;
result.speed = -1;
result.range = -1;
int32_t t = 0;
int32_t s = ego.position;
int32_t current_idx = 0;
int32_t stack_depth = 1;
int32_t stack_counter = 1;
int32_t visited_counter = 0;
const int32_t MAX_STACK_DEPTH = row_size * col_size;
int32_t stack_row[MAX_STACK_DEPTH];
int32_t stack_col[MAX_STACK_DEPTH];
stack_row[current_idx] = row_size - 1;
stack_col[current_idx] = s;
int32_t temp_row = 0;
int32_t temp_col = 0;
// In this case we only need to search to right and up.
while(visited_counter < stack_counter) {
  temp_row = stack_row[current_idx];
  temp_col = stack_col[current_idx];
  current_idx = IncrementStackIndex(current_idx, MAX_STACK_DEPTH);
  visited_counter++;
  if (map[temp_row][temp_col] == 0) {
    // Free
    candidate_speed = (temp_col - ego.position) / (row_size - temp_row);
    if ((temp_row <= 0 || temp_col >= col_size - 1) &&
        candidate_speed < MIN_SPEED) {
      // Manual limite the smalled speed.
      map[temp_row][temp_col] = 1;
      continue;
    }
    candidate_range = sqrt(
        (row_size - temp_row) * (row_size - temp_row)
        + (temp_col - ego.position) * (temp_col - ego.position));
    candidate_speed = 1.0 * (temp_col - ego.position) /
        (row_size - 1 - temp_row);
    if (temp_col >= MAX_ROAD_LENGTH - 1
        && candidate_speed > unbounded_speed) {
      // Found unbounded. Look at speed.
      unbounded_speed = candidate_speed;
      result.speed = (int)round(candidate_speed);
      result.range = (int)round(candidate_range);
```

```
        result.type = STRESULT_UNBOUNDED;
      } else if (temp_col < MAX_ROAD_LENGTH - 1 &&
          candidate_range > bounded_range &&
          result.type == STRESULT_BOUNDED) {
        // new bounded frontier. look at travel range.
        bounded_range = candidate_range;
        result.speed = (int)round(candidate_speed);
        result.range = (int)round(candidate_range);
      }
      // Search for right cell and upper cell;
      if (temp_col + 1 < col_size) {
        stack_row[stack_depth] = temp_row;
        stack_col[stack_depth] = temp_col + 1;
        stack_depth = IncrementStackIndex(stack_depth, MAX_STACK_DEPTH);
        stack_counter++;
      }
      if (temp_row - 1 >= 0) {
        stack_row[stack_depth] = temp_row - 1;
        stack_col[stack_depth] = temp_col;
        stack_depth = IncrementStackIndex(stack_depth, MAX_STACK_DEPTH);
        stack_counter++;
      }
      // visited cell.
      map[temp_row][temp_col] = 1;
    }
  }
  printf("CalculateSpeedProfile finihsed, speed: %d, range: %d\n",
      result.speed, result.range);
  return result;
}

struct STResult STGraphSolver(const struct Vehicle ego,
    const struct Vehicle* vehicles,
    const uint8_t max_lanes,
    const uint8_t max_cars) {
  // During each STGraphSolver we only calculate the result
  // for the current lane.
  // Init STResult. speed == -1 means no result.
  // Creating occupency map;
  // Indexing order is map[row][col];
  // Row aligns to y-axis(time)
  // Col aligns to x-axis(travel distance)
  uint8_t **occupency_map;
```

```c
  const uint32_t ROW_SIZE = MAX_ROAD_LENGTH / MIN_SPEED;
  const uint32_t COL_SIZE = MAX_ROAD_LENGTH;
  occupency_map = (uint8_t**)malloc(ROW_SIZE * sizeof(uint8_t*));
  for (uint32_t i = 0; i < ROW_SIZE; i++) {
    occupency_map[i] = (uint8_t*)calloc(COL_SIZE, sizeof(uint8_t));
  }
  // Filling the cell.
  struct Vehicle dummy_ego = ego;
  dummy_ego.speed = MAX_SPEED_FPS;
  OccupyTheMap(occupency_map, ROW_SIZE, COL_SIZE, dummy_ego, 1);
  for (uint8_t i = 0; i < max_cars; i++) {
    if (vehicles[i].lane == ego.lane) {
      if (vehicles[i].position <= ego.position) {
        // Have 1 second safety range.
        struct Vehicle dummy_car = vehicles[i];
        dummy_car.position += dummy_car.speed;
        OccupyTheMap(occupency_map, ROW_SIZE, COL_SIZE, dummy_car, -1);
      } else {
        // Have 1 second safety range.
        struct Vehicle dummy_car = vehicles[i];
        dummy_car.position -= dummy_car.speed;
        OccupyTheMap(occupency_map, ROW_SIZE, COL_SIZE, dummy_car, 1);
      }
    }
  }
  // printf("Map size: [%d, %d]\n", ROW_SIZE, COL_SIZE);
  // for (uint32_t c = 0; c < COL_SIZE; c++) {
  //   for (uint32_t r = 0; r < ROW_SIZE; r++) {
  //     printf("%d ", occupency_map[ROW_SIZE - 1 - r][c]);
  //   }
  //  printf("\n");
  // }
  struct STResult result = CalculateSpeedProfile(
      occupency_map, ROW_SIZE, COL_SIZE, ego);
  // Find the result.
  // free
  for (uint32_t i = 0; i < ROW_SIZE; i++) {
    free(occupency_map[i]);
  }
  free(occupency_map);
  return result;
}
```



```c
uint8_t CheckCollision(const struct Vehicle ego,
    const struct Vehicle* vehicles, const uint8_t max_lanes,
    const uint8_t max_cars, const int32_t seconds) {
  int32_t ego_next_pose = ego.position + ego.speed * seconds;
  int32_t other_next_pose = 0;
  for (uint8_t i = 0; i < max_cars; i++) {
    if (ego.lane != vehicles[i].lane) {
      continue;
    }
    other_next_pose = vehicles[i].position
      + vehicles[i].speed * seconds;
    if (abs(ego_next_pose - other_next_pose) >= MIN_DISTANCE_FT &&
        (ego.position - vehicles[i].position) *
        (ego_next_pose - other_next_pose) > 0) {
      continue;
    } else {
      return 1;
    }
  }
  return 0;
}

struct Vehicle ChangeEgoLaneLeft(const struct Vehicle ego) {
  struct Vehicle local_copy = ego;
  if (local_copy.lane > 0) {
    local_copy.lane--;
  }
  return local_copy;
}

struct Vehicle ChangeEgoLaneRight(const struct Vehicle ego,
    const uint8_t max_lanes) {
  struct Vehicle local_copy = ego;
  if (local_copy.lane < max_lanes - 1) {
    local_copy.lane++;
  }
  return local_copy;
}

struct Vehicle MoveEgo(
    const struct Vehicle ego, const struct Vehicle* vehicles,
    const uint8_t max_lanes, const uint8_t max_cars) {
  struct STResult best_st_result;
  struct STResult st_result;
  struct Vehicle dummy_ego = ego;
  // Current lane.
  best_st_result = STGraphSolver(ego, vehicles, max_lanes, max_cars);
  dummy_ego = ChangeEgoLaneLeft(ego);
  st_result = STGraphSolver(dummy_ego, vehicles, max_lanes, max_cars);
  if (st_result.speed > best_st_result.speed) {
    best_st_result = st_result;
  }
  best_st_result = CompareSTResult(best_st_result, st_result);
  dummy_ego = ChangeEgoLaneRight(ego, max_lanes);
  st_result = STGraphSolver(dummy_ego, vehicles, max_lanes, max_cars);
  best_st_result = CompareSTResult(best_st_result, st_result);
  dummy_ego = ego;
  dummy_ego.speed = best_st_result.speed;
  return dummy_ego;
}

#if (defined CASE_ONLY_ME) && (defined CBMC)
#define MAX_CARS 0
#define MAX_LANE_WIDTH 4
int main() {
  struct Vehicle vehicles[MAX_CARS];
  struct Vehicle ego;
  uint8_t collision = 0;

  // Generating ego vehicle.
  ego.lane = nondet_uchar();
  __CPROVER_assume(ego.lane >= 0 && ego.lane < MAX_LANE_WIDTH);
  ego.position = 0;
  ego.speed = nondet_int();
  __CPROVER_assume(ego.speed >= MIN_SPEED
      && ego.speed <= MAX_SPEED_FPS);
  // Check generated vehicle status.
  collision = CheckCollision(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS,
      COLLISION_CURRENT_SECOND);
  __CPROVER_assume(collision == 0);
  ego = MoveEgo(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS);
  collision = CheckCollision(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS,
      COLLISION_ADS_SECOND);
  __CPROVER_assert(collision == 0, "or there will be a collision");
}
```

```
#elif (defined CASE_ONE_SIDE) && (defined CBMC)
#define OTHER_MIN_POSITION 100
#define MAX_CARS 5
#define MAX_LANE_WIDTH 4
int main() {
  struct Vehicle vehicles[MAX_CARS];
  struct Vehicle ego;
  uint8_t collision = 0;

  // Generating ego vehicle.
  ego.lane = nondet_uchar();
  __CPROVER_assume(ego.lane >= 0 && ego.lane < MAX_LANE_WIDTH);
  ego.position = 0;
  ego.speed = nondet_int();
  __CPROVER_assume(ego.speed >= MIN_SPEED
      && ego.speed <= MAX_SPEED_FPS);
  // Generating other vehicles.
  for (uint8_t i = 0; i < MAX_CARS; i++) {
    vehicles[i].lane = ego.lane;
    vehicles[i].position = nondet_int();
    __CPROVER_assume(vehicles[i].position >= OTHER_MIN_POSITION
        && vehicles[i].position <= MAX_ROAD_LENGTH);
    vehicles[i].speed = nondet_int();
    __CPROVER_assume(vehicles[i].speed >= MIN_SPEED
        && vehicles[i].speed <= MAX_SPEED_FPS);
  }
  // Check generated vehicle status.
  collision = CheckCollision(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS,
      COLLISION_CURRENT_SECOND);
  __CPROVER_assume(collision == 0);
  ego = MoveEgo(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS);
  collision = CheckCollision(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS,
      COLLISION_ADS_SECOND);
  __CPROVER_assert(collision == 0, "or there will be a collision");
}

#elif (defined CASE_BOTH_SIDE) && (defined CBMC)
#define MAX_CARS 2
#define MAX_LANE_WIDTH 4
int main() {
  struct Vehicle vehicles[MAX_CARS];
  struct Vehicle ego;
  uint8_t collision = 0;
```

```c
  ego.lane = 0;
  ego.speed = 50;
  ego.position = 100;
  while(1) {
    // Vehicle[0] is behind ego vehicle
    vehicles[0].lane = 0;
    vehicles[0].position = nondet_int();
    __CPROVER_assume(ego.position < ego.position - 50
        && ego.position < MAX_ROAD_LENGTH);
    vehicles[0].speed = nondet_int();
    __CPROVER_assume(ego.speed >= MIN_SPEED
        && ego.speed < MAX_SPEED_FPS);
    // vehicle[1] is in front of ego vehicle
    vehicles[1].lane = 0;
    vehicles[1].position = nondet_int();
    __CPROVER_assume(ego.position > ego.position + 50
        && ego.position < MAX_ROAD_LENGTH);
    vehicles[1].speed = nondet_int();
    __CPROVER_assume(ego.speed > vehicles[0].speed
       && ego.speed < MAX_SPEED_FPS);
    collision = CheckCollision(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS,
        COLLISION_CURRENT_SECOND);
    __CPROVER_assume(collision == 0);
    ego = MoveEgo(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS);
    collision = CheckCollision(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS,
        COLLISION_ADS_SECOND);
    __CPROVER_assert(collision == 0, "or there will be a collision");
  }
}

#elif (defined CASE_MORE_OTHER_VEHICLES) && (defined CBMC)
#define MAX_CARS 5
#define MAX_LANE_WIDTH 4
int main() {
  struct Vehicle vehicles[MAX_CARS];
  struct Vehicle ego;
  uint8_t collision = 0;
  // Generating ego vehicle.
  ego.lane = nondet_uchar();
  __CPROVER_assume(ego.lane >= 0 && ego.lane < MAX_LANE_WIDTH);
  ego.position = nondet_int();
  __CPROVER_assume(ego.position >= 0 && ego.position < MAX_ROAD_LENGTH);
  ego.speed = nondet_int();
  __CPROVER_assume(ego.speed >= MIN_SPEED && ego.speed <= MAX_SPEED_FPS);
  // Generating other vehicles.
  for (uint8_t i = 0; i < MAX_CARS; i++) {
    vehicles[i].lane = nondet_uchar();
    __CPROVER_assume(vehicles[i].lane >= 0
        && vehicles[i].lane < MAX_LANE_WIDTH);
    vehicles[i].position = nondet_int();
    __CPROVER_assume(vehicles[i].position >= 0
        && vehicles[i].position <= MAX_ROAD_LENGTH);
    vehicles[i].speed = nondet_int();
    __CPROVER_assume(vehicles[i].speed >= MIN_SPEED
        && vehicles[i].speed <= MAX_SPEED_FPS);
  }
  // Check generated vehicle status.
  collision = CheckCollision(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS,
      COLLISION_CURRENT_SECOND);
  __CPROVER_assume(collision == 0);
  ego = MoveEgo(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS);
  collision = CheckCollision(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS,
      COLLISION_ADS_SECOND);
  __CPROVER_assert(collision == 0, "or there will be a collision");
}

#elif (defined CASE_TEST_COLLISION) && (defined CBMC)
#define MAX_CARS 3
#define MAX_LANE_WIDTH 4
int main() {
  struct Vehicle vehicles[MAX_CARS];
  struct Vehicle ego;
  uint8_t collision = 0;
  // Generating ego vehicle.
  ego.lane = 0;
  ego.position = 150;
  ego.speed = 30;

  // Generating other vehicles.
  vehicles[0].lane = 0;
  vehicles[0].position = 90;
  vehicles[0].speed = 60;
  vehicles[1].lane = 1;
  vehicles[1].position = 150;
  vehicles[1].speed = 30;
  vehicles[2].lane = 10;
  vehicles[2].position = 210;
  vehicles[2].speed = 10;

  // Check generated vehicle status.
  collision = CheckCollision(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS,
      COLLISION_CURRENT_SECOND);
  __CPROVER_assume(collision == 0);
  ego = MoveEgo(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS);
  collision = CheckCollision(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS,
      COLLISION_ADS_SECOND);
  __CPROVER_assert(collision == 0, "or there will be a collision");
}

#elif (defined GNUC)
#define MAX_CARS 2
#define MAX_LANE_WIDTH 4
int main() {
  struct Vehicle vehicles[MAX_CARS];
  struct Vehicle ego;
  uint8_t collision = 0;
  // Generating ego vehicle.
  ego.lane = 0;
  ego.position = 150;
  ego.speed = 30;

  // Generating other vehicles.
  vehicles[0].lane = 0;
  vehicles[0].position = 90;
  vehicles[0].speed = 60;
  vehicles[1].lane = 1;
  vehicles[1].position = 150;
  vehicles[1].speed = 30;
  vehicles[2].lane = 10;
  vehicles[2].position = 210;
  vehicles[2].speed = 10;

  // Check generated vehicle status.
  collision = CheckCollision(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS,
      COLLISION_CURRENT_SECOND);
  ego = MoveEgo(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS);
  collision = CheckCollision(ego, vehicles, MAX_LANE_WIDTH, MAX_CARS,
      COLLISION_ADS_SECOND);
  assert(collision == 0);
```

```
}
#endif
```

References

Blum, Avrim L., and Merrick L. Furst. “Fast Planning Through Planning Graph Analysis.” *Artificial Intelligence*, vol. 90, no. 281, 1997, p. 300. *graphplan.pdf*, https://www.cs.cmu.edu/~avrim/Papers/graphplan.pdf.

GM Cruise. “Cruise.” *https://www.getcruise.com/*.

LaValle, Steven M M. *Planning Algorithms*. Cambridge University Press, 2006. *Planning Algorithms*, http://planning.cs.uiuc.edu/booka4.pdf.

Nister, David. “The Safety Force Field.” *Nvidia*, vol. -, no. -, 2019, -. *The Safety Force Field*, https://www.nvidia.com/content/dam/en-zz/Solutions/self-driving-cars/safety-force-field/the-safety-force-field.pdf.

Shalev-shwartz, Shai. “On a Formal Model of Safe and Scalable Self-driving Cars.” *Mobileye*, vol. -, no. -, 2017, -. *Arxiv*, https://arxiv.org/pdf/1708.06374.pdf.

Uber. “Uber ATG.” *https://www.uber.com/us/en/atg/technology/*.

Waymo. “Waymo.” *https://waymo.com/*.