4-14-2016

APEX: Autonomous Vehicle Plan Verification and Execution

Matthew O'Kelly
mokelly@seas.upenn.edu

Houssam Abbas
University of Pennsylvania, habbas@seas.upenn.edu

Sicun Gao
Massachusetts Institute of Technology, scungao@gmail.com

Shin'ichi Shiraishi
Toyota Info Technology Center, sshiraishi@us.toyota-itc.com

Shinpei Kato
Nagoya University, shinpei@is.nagoya-u.ac.jp

See next page for additional authors

Follow this and additional works at: https://repository.upenn.edu/mlab_papers

Part of the [Computer Engineering Commons](#), [Systems and Communications Commons](#), and the [Theory and Algorithms Commons](#)

Recommended Citation (OVERRIDE)

Matthew O'Kelly, Houssam Abbas, Sicun Gao, Shin'ichi Shiraishi, Shinpei Kato, and Rahul Mangharam, "APEX: Autonomous Vehicle Plan Verification and Execution" SAE World Congress, April 2016.

This paper is posted at ScholarlyCommons. https://repository.upenn.edu/mlab_papers/84
For more information, please contact repository@pobox.upenn.edu.

APEX: Autonomous Vehicle Plan Verification and Execution

Abstract

Autonomous vehicles (AVs) have already driven millions of miles on public roads, but even the simplest scenarios have not been certified for safety. Current methodologies for the verification of AV's decision and control systems attempt to divorce the lower level, short-term trajectory planning and trajectory tracking functions from the behavioral rules-based framework that governs mid-term actions. Such analysis is typically predicated on the discretization of the state space and has several limitations. First, it requires that a conservative buffer be added around obstacles such that many feasible plans are classified as unsafe. Second, the discretized controllers modeled in this analysis require several refinement steps before being implementable on an actual AV, and typically do not allow the specification of comfort-related properties on the trajectories. In contrast, consumer-ready AVs use motion planning algorithms that generate smooth trajectories. While viable algorithms exist for the generation of smooth trajectories originating from a single state, analysis should consider that the AV faces state estimation errors and disturbances. Third, verification is restricted to a discretized state space with fixed-size cells; this assumption can artificially limit the set of available trajectories if the discretization is too coarse. Conversely, too fine of a discretization renders the problem intractable for automated analysis. This work presents a new verification tool, APEX, which investigates the combined action of a behavioral planner and state lattice-based motion planner to guarantee a safe vehicle trajectory is chosen. In APEX, decisions made at the behavioral layer can be traced through to the spatio-temporal evolution of the AV and verified. Thus, there is no need to create abstractions of the AV's controllers, and aggressive trajectories required for evasive maneuvers can be accurately investigated.

Keywords

autonomous vehicles, formal verification, reachability

Disciplines

Computer Engineering | Electrical and Computer Engineering | Systems and Communications | Theory and Algorithms

Author(s)

Matthew O'Kelly, Houssam Abbas, Sicun Gao, Shin'ichi Shiraishi, Shinpei Kato, and Rahul Mangharam

APEX: Autonomous Vehicle Plan Verification and Execution

Matthew O’Kelly, Houssam Abbas
University of Pennsylvania

Sicun Gao
MIT

Shin’ichi Shiraishi
Toyota Info Technology Center USA

Shinpei Kato
Nagoya University

Rahul Mangharam
University of Pennsylvania

Copyright © 2015 Society of Automotive Engineers, Inc.

Abstract

Autonomous vehicles (AVs) have already driven millions of miles on public roads, but even the simplest scenarios have not been certified for safety. Current methodologies for the verification of AV’s decision and control systems attempt to divorce the lower level, short-term trajectory planning and trajectory tracking functions from the behavioral rules-based framework that governs mid-term actions. Such analysis is typically predicated on the discretization of the state space and has several limitations. First, it requires that a conservative buffer be added around obstacles such that many feasible plans are classified as unsafe. Second, the discretized controllers modeled in this analysis require several refinement steps before being implementable on an actual AV, and typically do not allow the specification of comfort-related properties on the trajectories. Consumer-ready AVs use motion planning algorithms that generate smooth trajectories. While viable algorithms exist for the generation of smooth trajectories originating from a single state, analysis should consider that the AV faces state estimation errors and disturbances. Verification is restricted to a discretized state space with fixed-size cells; this assumption can artificially limit the set of available trajectories if the discretization is too coarse. Conversely, too fine of a discretization renders the problem intractable for automated analysis. This work presents a new verification tool, APEX, which investigates the combined action of a behavioral planner and state lattice-based motion planner to guarantee a safe vehicle trajectory is chosen. In APEX, decisions made at the behavioral layer can be traced through to the spatio-temporal evolution of the AV and verified. Thus, there is no need to create abstractions of the AV’s controllers, and aggressive trajectories required for evasive maneuvers can be accurately investigated.

Introduction

Advanced Driver Assistance Systems (ADAS) and fully autonomous vehicles (AVs) feature a complex integration between discrete and continuous controllers. The discrete controllers make decisions regarding the vehicle’s next goal (“move to left lane”), while the continuous controllers determine throttle and steering inputs which allow the vehicle to execute the decisions of the discrete controller.

Approximately 1.25 million traffic accidents occur yearly around the world; almost 90% of such accidents are attributable to driver error [1]. ADAS and AV technology has the potential to all but eliminate this burden from society; however, experts are still unsure when such technology will be ready for the consumer market. Thus, as such vehicles come to market there is a pressing need to *bound and minimize* the risks they might pose to other vehicles, pedestrians and infrastructure.

Legal liability is a main consideration in the design of the next generation of vehicles, and could determine the future of ADAS and AVs as a mainstream technology. The question is: who’s liable when the AV (or an ADAS feature) causes an accident or unsafe situation? Currently there is no one answer to the question: on the ADAS side, Tesla Motors has included a lane changing feature; however, it must be initiated by the human driver as a way to make the latter ultimately responsible for the outcomes.

Volvo on the other hand has recently announced it will assume full liability for its autonomous cars’ actions, and warns that “the US risks losing its leading position due to the lack of Federal guidelines for the testing and certification of autonomous vehicles” [2]. The possibility that the manufacturer might ultimately be deemed responsible for the car’s actions highlights the urgent need for a technology that can *automatically and exhaustively certify the impossibility of accidents in various driving scenarios, and under well-defined conditions*.

It is clear that, for now, next generation AVs will remain research projects because there is a lack of confidence that they

can be used in safety critical applications. More specifically, there are currently few tools for plan verification and execution analysis [3]. New, more practical methods, for formal verification and model based design could increase the confidence that highly autonomous systems can be put into service and can potentially reduce both development costs and time to market.

A successful tool must handle a large variety of scenarios in which the AVs will operate: for example, highway driving, entrance ramps, roundabouts, and stoplights. Furthermore, each scenario has a large number of variations: highway driving involves a varying number of cars and different starting configurations of the cars. These *configurations* define the traffic participants relative positions, their initial speeds, orientations, and their goals. The configuration space can be extremely large, and in fact, is often uncountably infinite, as can be seen by considering that a car's position is a real-valued variable. In this paper, we address two questions: (1) Is it possible to investigate all configurations via symbolic abstraction? (2) If it is, how can we verify that the car's behavior is correct in *all* configurations? Here, correct means that the vehicle is both safe and achieves the mobility goals of the pilot.

The most common approach today to verifying the safety and correctness of the car's controllers is to perform a large number of simulations and tests. Every simulation varies the scenario configuration: in one simulation the cars start $0.7m$ away from each other, in another they start $2m$ apart with equal speeds, in yet a third they start $2m$ apart with the lead vehicle making a left turn. However, simulation *always* leaves a verification gap. To illustrate this gap, we focus on a common scenario: the lane change.

Simulating a lane change

Consider the situations presented in Fig. 1, all of which are variations on a lane change scenario. Every car in the scenario is characterized by its *state* x which includes at least its position, velocity, orientation and yaw rate. In general, additional variables might be used to describe the state of the car at every time instant. In this example, we use a 7-dimensional state. To simulate the scenario, we select initial values for these variables, i.e., an initial state $x(0)$ (which we called configuration). Note for this example that once an initial state is fixed, the system evolves deterministically based on the controllers of each car. The initial state can have any value in a bounded set: e.g., in Fig. 1, the initial position (s_x, s_y) of the ego vehicle is in $[0, 1] \times [0, 1]$, and the velocity v is in $[24, 33]m/s$. There are two distinct sources for uncertainty about the state: first, the ego vehicle will have to perform a lane change under a variety of initial states. Simulating it under only one initial state is clearly insufficient, because we expect the outcome of the scenario when the two cars start $0.5m$ apart to differ from its outcome when the two cars start $5m$ apart. The second source of uncertainty comes from errors in perception such as localization and velocity estimate. Even if we wish to start the simulation in a particular state, inaccuracies in measurements mean that the car's state can not be exactly known. So while the control algorithms assume a given starting state, the car may actually begin from anywhere in a bounded set around that state estimate. Thus it's important to verify that these measurement errors do not cause unsafe situations.

Figure 1: Simulation is not sufficient to fully verify a lane change. After a large number of simulations, the unsafe scenario at the bottom may still not be detected as simulation-based testing is not exhaustive and leaves a verification gap.

The question then becomes: how many simulations should we perform, and *which* simulations should we perform? Ideally, we would simulate all configurations that produce an unsafe outcome, but this can not be guaranteed in general. Even experienced engineers might not think of corner cases, especially given the size of the vehicles configuration space. E.g., in Fig. 1, we show a lane change scenario, which has been simulated a 1000 times, including with varying numbers of vehicles. Yet, it is only the last, non-simulated, situation that reveals the collision: if the ego vehicle starts with a positive orientation and yaw rate, and attempts to change lanes while the other vehicle is slowing down, it could cause a collision. This is because the ego vehicle is unable to exactly follow the reference trajectory which the motion planner determined would be safe.

Randomized testing, where the configurations are sampled from hypercubes of parameters, is not a scalable solution: suppose we decide to sample only 10 points in the range of every state variable. For our 7D model, and with 2 cars, this yields a total of 10^{14} simulations. Say we wish to simulate 10 seconds. Even if a simulation runs in real-time, this still requires $10 \cdot 10^{14}$ seconds = 30 million years to complete.

Thus, while simulations are a useful and intuitive method for getting a quick confidence level in the basic safety of a scenario, they are not sufficient for *guaranteeing* the absence of risk in a given scenario with a bounded state space.

Contributions

Our main contribution is a design-time approach to *formally* verifying the trajectory planning and trajectory tracking stacks of an ADAS/AV as they interact with potentially dynamic participants in a *variety* of driving scenarios. This approach is implemented in a software tool, APEX, and illustrated with examples of a lane change maneuver. The verification approach has two characteristics:

- It is formal: we are *guaranteed* that if APEX determines a scenario to be safe, then it is safe. No amount of simulation can find an unsafe behavior in a scenario verified as correct by APEX.
- It allows the use of an arbitrary trajectory planner, for example, it could be code or an abstraction. That is, there is

no need to model the trajectory planner, which is often very complex software. Moreover, the same trajectory planner can then be run on a real vehicle. In the case study presented in this paper, APEX uses a trajectory planner that has been tested on a real vehicle.

In APEX, the verification engineer can

- Specify the low-level dynamics of the vehicle, including the trajectory tracker. Unlike other approaches and existing tools the dynamics can be nonlinear. The default model in APEX is a 7D bicycle model.
- Provide a motion planner that takes in a starting position and end position and returns a trajectory that links the two points. The motion planner can be *any piece of software*: there are no restrictions on it. The default planner in APEX is a state lattice planner incorporated in ROS and tested on a real vehicle. Figure 2 shows the planner GUI available as part of Autoware [4].
- Specify a sequence of goal positions (or *waypoints*) that the vehicle must visit, or a behavioral planner that computes these waypoints in a reactive manner. The default behavioral planner in APEX is a simple 2-state automaton that decides whether to execute lane following or lane changing. However, we expect that designers will implement many other more complex behavioral planners.
- Specify the uncertainty sets for the ego vehicle and the other agents in the scenario.
- Specify the unsafe conditions to be avoided by the vehicle. APEX supports a rich specification language, Metric Interval Temporal Logic (MITL) for the description of unsafe behaviors [5].

APEX will then verify, in an exhaustive fashion, that the ego vehicle can complete the scenario under the specified uncertainty, or return a specific case where it fails. The engineers can then use this *counter-example* in order to debug the controllers, and better understand how to avoid this failure at design-time.

Figure 2: ROS APEX planning implementation GUI.

One real-world example of an AV software bug related to plan execution was highlighted by the first ever crash between AVs at the Urban Challenge [6]. At the time of the accident, participants noted that there are no known “formal methods that would allow definitive statements about the completeness or correctness of a vehicle interacting with a static environment, much less a dynamic one” [7]. It is beyond the scope of this

Classical three layer planning architecture

Figure 3: The three layer architecture presented by Gat is widely accepted as a standard means of implementing planning and control for an autonomous vehicle.

paper to review the numerous developments in verification and synthesis technology; we note attempts exist to reason about the safety of autonomous vehicles in static environments via synthesis [8], but such methods cannot currently scale to realistic systems and are extremely conservative. In response the authors of [8] propose a receding horizon framework, but still rely on coarse grid-based abstractions. Others have sought to verify Adaptive Cruise Control Algorithms (ACC) which severely restrict scenarios in which the car may operate (no lane changes) [9]. Finally, some research which eschews discretization in favor of continuous linearized dynamics focuses on moving the verification task online [10].

APEX description and usage

Planning and Control for Autonomous Vehicles

In order to motivate the need for the APEX approach, we first outline the architecture of a typical ADAS/AV control system. It is *not* necessary that a vehicle use this *particular architecture* in order to be verified under APEX, but it motivates the key issues involved in obtaining a proof of safety. In the three-layer architecture paradigm [11] which we demonstrate, the planning and control of the vehicle is hierarchical in nature. Each successive layer performs a task over a shorter time horizon. Fig. 3 details this approach to AV architecture.

At the top level a mission planner is given a mobility goal. Such a goal is typically expressed as a (current location, destination) pair. Given this pair the mission planner finds an optimal (or feasible) route through the road network.

In the next layer, the behavioral planner makes local decisions about how to navigate the road network. For example, if the mission planner informs the behavioral planner that at the next intersection it will need to turn left, the behavioral planner will use a set of rules to determine that the ego vehicle must be in the left lane. It then provides a sequence of waypoints, or intermediary destinations, to the lower-level local planner.

Finally, the local planner, or *trajectory planner*, produces a trajectory that connects the vehicle’s current pose to the target pose at the next waypoint. Here ‘pose’ refers to the combined position, heading and velocity of the vehicle. Specifically, given

Figure 4: Reachability Analysis is a means of formal verification which shows that for a given initial set and a specified unsafe set, no trajectory of the system can reach the unsafe set

Figure 5: One step in the APEX tool: the local planner generates a trajectory, which is automatically input into the mission definition file and verified using dReach

a goal pose relative to the vehicle’s current pose, the local planner computes a set of candidate smooth trajectories that can lead to the goal pose or near it, then selects a single trajectory and sends it to the vehicle. The vehicle itself includes a PID controller (or some other controller) in order to track the selected trajectory.

Reachability

APEX applies recent techniques in reachability analysis [12] to the verification of hybrid systems. Hybrid systems provide a precise and formal means of describing models which have both continuous state variables and discrete operating modes, and are an appropriate model for autonomous vehicles as will be shown. Reachability analysis asks the following question: given an initial set of states of the system, and an unsafe set of states that the system should not visit, is it possible for the system to reach the unsafe set if it starts in the initial set? See Fig. 4. A reachability tool either answers the question negatively (the system is safe), or it provides a particular execution of the system that enters the unsafe set. Such an execution is known as a *counter-example*.

The APEX Approach

APEX answers the following question by re-formulating it as a reachability problem: given some initial uncertainty about the state of the AV, constraints on the configuration of the environ-

ment, and a desired behavior of the AV (like mobility goal and traffic laws), is it ever possible for the AV to violate the desired behavior? Fig. 5 summarizes a single execution of the verification engine. For each trajectory selected by the planner (highlighted in red), APEX calls dReach [13], a reachability analysis tool for nonlinear hybrid systems.

We emphasize that the verification process is *offline* - the vehicle does *not* run APEX while it is driving. At each decision point encountered by the behavioral planner there may be multiple executions of the verification engine depending on the design of the behavioral planner. Fig. 6 describes how the execution of the controller online relates to the *offline* verification process. In contrast to the simulation based approach outlined in the introduction, the result of the APEX approach is that we have converted a brute force search over real intervals into a finite series of tractable bounded reachability problems over a finite verification horizon.

Tool Input

APEX is a *command line tool* for verification of autonomous vehicle missions written in Python and C++. The input to the verification process is a *mission definition file*. The *mission definition file* defines the sequence of waypoints or road links which the vehicle will traverse in order to achieve a mobility goal.

The *mission definition file* describes the following:

- *The collection of agents in the scenario*, consisting of the ego vehicle and other cars in the scenario. The agents are described via ODEs that describe the evolution of their state with time, and their behavioral planners, which give the next waypoints for each vehicle. All agents operate in an ontology specific to the mission, in this case the world model consists of a geometric description of a road network.
- *Set of initial states* for each state variable of every vehicle.
- *The constraints that the AV should satisfy*, such as traffic laws and the unsafe conditions that ego vehicle must avoid. These are described in MITL.
- *The goal of the ego vehicle*, also expressed in MITL.

The mission definition file is part of a *mission definition script*. The latter manages the execution of the behavioral planner and trajectory generator. Each (state, goal) pair that is encountered on the mission generates at least one trajectory which must be verified. The *mission definition script* automatically updates a *scenario verification instance*. The *scenario verification instance* is a dReach (.drh) file which combines the results of the plan execution with the dynamical model of the vehicle and a low-level trajectory tracking controller. The *agent definition file* contains the dynamical model of the vehicle and the tracking controller is also written using the syntax of dReach, it may be manually edited in order to match mission specific vehicle models. We provide an example of the syntax of the composed *scenario verification instance* in Fig. 7.

Together, the constraints of the environment ξ and ego vehicle goal and constraints ϕ constitute the *specification* of the mission. The mission is a success if every execution of the system (i.e., every simulation) satisfies the specification.

Figure 6: Stages of APEX verification and their correspondence to control execution. Top left: at $t = 0$, mode is Lane Follow (LF) and the vehicle follows the current lane. Top right: at $t = 1$, mode is Lane Change (LC) and vehicle starts a lane change maneuver. Bottom left: offline, APEX verifies that in mode LF, the vehicle can track the trajectory. Bottom right: offline, APEX verifies both possible executions, a lane change and a lane following.

Tool Output

Each *scenario verification instance* can return either SAFE or δ -UNSAFE. SAFE means that for all possible executions of the system we can not reach an unsafe state. δ -UNSAFE means that there exists an execution of the system which comes within a δ of the unsafe region, and possibly enters it. If the system is δ -UNSAFE the tool will return a counter-example describing a tube around a concrete trajectory whose intersection with the unsafe region is not empty. Users of the APEX tool should be aware that selecting too large of a precision value (δ) may result in δ -UNSAFE results which are false positives, but any declaration of SAFE is guaranteed to be correct.

Building an Autonomous Vehicle Agent

To run APEX, we need to capture the AV dynamics, the low level tracking controller, and the planning stack which generates the trajectories for the vehicle to follow.

Modeling

The first step towards verification is a model of the AV. APEX uses the formalism of nonlinear hybrid systems to describe the AV and other vehicles. The trajectory tracking controller and AV can be described using ordinary differential equations. The

discrete nature of the behavioral control layer dictates that we much capture a system with mixed continuous-discrete dynamics. We provide a list of symbols used in Table 1.

Ego Vehicle Model

APEX uses a non-linear 7 degree of freedom bicycle model [14] in order to describe the ego-vehicle. Higher order models can be supported in the future, and of course the parameters of the base model can be customized in order to match specific vehicles. See Fig. 8. The input to such a model is steering angle velocity and linear velocity, the output is vehicle state as a function of time.

The state vector describing the vehicle is described in equations (1)-(7). The variable β is the slip angle at the center of mass, ψ is the heading angle, $\dot{\psi}$ is the yaw rate, v is the velocity, s_x and s_y are the x and y positions, and δ is the angle of the front wheel. In the formulation of [6], the inputs to the system are a_x , the longitudinal acceleration, and v_w the rotational speed of the steering angle.

$$x_v = (\beta, \Psi, \dot{\Psi}, v, s_x, s_y, \delta) \quad (1)$$

```

// Each verification instance is automatically updated to
// to reflect the new planned trajectory.
#define s (22.681867)
#define kappa_0 (0.000000)
#define kappa_1 (0.033194)
// ...

// Below are constants regarding the vehicle and controller.
// These do not vary between trajectories, manually edited
#define m 2273.0
// ...

// Below are automatically generated parameters, helper
// definitions, etc.
#define w 3.7

// Cubic spline inputs
#define a (kappa_0)
// ...

// -----

// Below are the automatically generated vehicle dynamics
// equations

mode 1;

invt:
// X position must be greater than or equal to 0
(sx>=0);
// Velocity must be greater than or equal to 0
(v>=0);

flow:
/* Target Vehicle Dynamics*/
d/dt[tau]=1;
// Compute rate of change of the slip angle
d/dt[Beta]=((c1/(m*v*v))-1.0)*Psi_dot + Cf*delta/(m*v) - (Cf+Cr)
*Beta/(m*v);
//...

// Controller Equations -- Generated automatically
d/dt[kappa_d] = b*v_d + 2*c*v_d*v_d*tau + 3*e*v_d*v_d*v_d*tau*
tau;
// ...

jump:

// -----

// Set of Initial Conditions
init:
@1 (and (sx>=0) (sx<=0.5) (sy>=0) (sy<=0.1) (v>=10.8) (v<=11.1))
// ...
// Set of Goal Conditions
goal:
@1 (or (and (sy<w) (tau>2) (tau<3)) (and (sy<w) (sx>sx_env)));

```

Figure 7: Scenario verification instance generated by APEX

The state equations for the system as described in [15] are:

$$\dot{\beta} = \left(\frac{C_r l_r - C_f l_f}{m v^2} \right) \dot{\psi} + \left(\frac{C_f}{m v} \right) \delta - \left(\frac{C_f + C_r}{m v} \right) \beta \quad (2)$$

$$\ddot{\psi} = \left(\frac{C_r l_r - C_f l_f}{I_z} \right) \beta - \left(\frac{C_f l_f^2 - C_r l_r^2}{I_z} \right) \left(\frac{\dot{\psi}}{v} \right) + \left(\frac{C_f l_f}{I_z} \right) \delta \quad (3)$$

Figure 8: Nonlinear bicycle model describing the statespace for the APEX approach to vehicle dynamics

Table 1: Symbols for Vehicle Model

Symbol List		
Symbol	Units	Description
x_v	-	Verification State Vector
x_{sl}	-	Lattice Planning State Vector
x_p	-	Vehicle Pose
x_g	-	Goal Pose
x_f	-	Predicted Vehicle Pose
p	-	Cubic Spline Parameter Vector
t_f	s	Prediction Horizon
m	kg	Vehicle Mass
l_r	m	Rear Wheelbase
l_f	m	Front Wheelbase
I_z	kg m ²	Moment of Inertia
C_f	N/rad	Front Cornering Stiffness
C_r	N/rad	Rear Cornering Stiffness
β	rad	Slip Angle
Ψ	rad	Heading Angle
v	m/s	Velocity
s_x	m	Position, x
s_y	m	Position, y
δ	rad	Steering Angle
ϵ_x	m	Tracking Error, x
ϵ_y	m	Tracking Error, y
v_w	rad/s	Steering Angle Velocity
a_x	m/s ²	Longitudinal Acceleration
κ	rad/m	Curvature
s_f	m	Arc Length

$$\dot{v} = a_x \quad (4)$$

$$\dot{s}_x = v \cos(\beta + \psi) \quad (5)$$

$$\dot{s}_y = v \sin(\beta + \psi) \quad (6)$$

$$\dot{\delta} = v_w \quad (7)$$

Vehicle Parameters

The parameters C_f, C_r and l_f, l_r describe respectively the cornering stiffness and distances from the center of gravity to the axles respectively; the subscripts f, r denote whether the parameter is defined for the front or rear of the vehicle. The moment of inertia, I_z and the vehicle mass, m are experimentally determined constants [16]. The kinematic bicycle model considers the two front wheels and two rear wheels of the vehicle to move in unison, with steering provided by the front wheels only. Furthermore, Each abstracted wheel is located along the center of the vehicle's body. Table 2 contains the validated vehicle parameters as given in [15]. It is possible to obtain such parameters and replace these constants in order to investigate specific vehicle characteristics.

Table 2: Parameters of Example Ego Vehicle [15]

Vehicle Parameters					
$m(\text{kg})$	$I_z(\text{kg} \cdot \text{m}^2)$	$C_f(\text{N/rad})$	$C_r(\text{N/rad})$	$l_f(\text{m})$	$l_r(\text{m})$
2273	4423	10.8e4	10.8e4	1.292	1.515

Tracking Controller

A simple trajectory tracking controller is included with the APEX vehicle model. Trajectory tracking controllers guide a vehicle along a geometrically defined cubic spline by applying steering and longitudinal acceleration inputs. A successful path tracking algorithm maintains vehicle stability and attempts to minimize the error between the desired trajectory and actual trajectory. The parameters computed for this controller when implemented and validated on a typical crossover SUV [15] are presented in Table 3.

Table 3: Controller Parameters [15]

Controller Parameters					
k_1	k_2	k_3	k_4	k_5	k_6
2	12	4	2	1	1.515

Using the approach in [16] and [15] the control inputs for longitudinal acceleration (pressing the accelerator) and steering angle velocity (turning the steering wheel) can be computed as v_w and a_x respectively.

$$v_w = k_1(\cos(\Psi_d)(s_{y,d} - s_y - w_y) - \sin(\Psi_d)(s_{x,d} - s_x - w_x)) + k_2(\Psi_d - \Psi - w_\Psi) + k_3(\dot{\Psi}_d - \dot{\Psi} - w_{\dot{\Psi}}) - k_4(\delta - w_\delta) \quad (8)$$

$$a_x = k_5(\cos(\Psi_d)(s_{x,d} - s_x - w_x) + \sin(\Psi_d)(s_{y,d} - s_y - w_y)) + k_6(v_d - v - w_v) \quad (9)$$

We note that we cannot use traditional linear systems techniques or sum of squares optimizations to directly find a Lyapunov function for this system because of the obvious non-linearity and non-polynomial form of the governing ordinary differential equations. Instead we will seek to show stability and safety properties using reachability and model checking analysis.

Planning

In APEX we provide a validated planning stack which can be run on a real vehicle. The planning strategy is hierarchical and includes: mission planning, behavioral planning, and local planning. In this section we will focus on the local planner because it is the layer which connects directly to the tracking controller for the vehicle. The local planner is used to generate smooth trajectories which a non-holonomic dynamically constrained vehicle is capable of following. Our planning stack utilizes the methods outlined in [17] commonly known as state-lattice planning with cubic spline trajectory generation.

Each execution of the planner requires as an input the current state of the vehicle and a goal state as defined by the behavioral planner. We note that we will call the vehicle state x_{sl} because it does not necessarily have to be the same as the model used for verification (although it can be). The planner must run online, in real-time, therefore, lower order models are often substituted here. In this implementation we define x_{sl} as:

$$x_{sl} = (s_x, s_y, v, \Psi, \kappa) \quad (10)$$

Where s_x and s_y are the x and y positions of the center of mass, v is the velocity, Ψ is the heading angle, and κ is the

Figure 9: Details of a local planning algorithm and used by AVs employing state lattice planning

curvature. We note that the state equations involve an additional constant, L which is the wheelbase of the vehicle. Where the state equations are described as:

$$\dot{x} = v * \cos(\Psi) \quad (11)$$

$$\dot{y} = v * \sin(\Psi) \quad (12)$$

$$\dot{\theta} = \kappa * v \quad (13)$$

$$\dot{\kappa} = \frac{\dot{\Psi}}{L} \quad (14)$$

The local planner's objective is then to find a feasible trajectory from the initial state defined by the tuple x_{sl} to a goal pose x_p defined as:

$$x_p = (s_x, s_y, \Psi) \quad (15)$$

In this formulation we limit trajectories to a specific class of parameterized curves known as cubic splines. A cubic spline is defined as a function of arc length:

$$\kappa(s) = \kappa_0 + a\kappa_1 s + b\kappa_2 s^2 + c\kappa_3 s^3 \quad (16)$$

Note that there are four free parameters (a, b, c, s_f) and our goal posture has four state variables. Thus, a cubic spline is a minimal polynomial that can be assured to produce a trajectory from the current position to the goal position (if it is kinematically feasible). For any particular state, goal pair there are two steps necessary to compute the parameters. First, it is necessary to produce an initial guess. There are several approaches available such as using a neural network, lookup table, or a simple heuristic. In this case we adapt a heuristic from Nagy and Kelly [18] such that it is compatible with a stable parameter formulation presented by McNaughton [17]. The stable reparameterization is defined as:

$$\kappa(0) = p_0 \quad (17)$$

$$\kappa(s_f/3) = p_1 \quad (18)$$

$$\kappa(2s_f/3) = p_2 \quad (19)$$

$$\kappa(s_f) = p_3 \quad (20)$$

Figure 10: Output of an execution (10 Hz) of the trajectory generator, a single trajectory will be chosen from this set.

Where the parameters (a, b, c, s_f) can now be expressed as:

$$a(p) = p_0 \quad (21)$$

$$b(p) = -\frac{11p_0 - 18p_1 + 9p_2 - 2p_3}{2s_f} \quad (22)$$

$$c(p) = \frac{9 * (2p_0 - 5p_1 + 4p_2 - p_3)}{2s_f^2} \quad (23)$$

$$d(p) = -\frac{9(p_0 - 3p_1 + 3p_2 - p_3)}{2s_f^3} \quad (24)$$

Which results in the following initialization heuristic:

$$p_0 = \kappa_0 = \kappa_i \quad (25)$$

$$p_1 = \kappa_1 = \frac{1}{49}(8b(s_f - s_i) - 26\kappa_0 - \kappa_3) \quad (26)$$

$$p_2 = \kappa_2 = \frac{1}{4}(\kappa_3 - 2\kappa_0 + 5\kappa_1) \quad (27)$$

$$p_3 = \kappa_3 = \kappa_f \quad (28)$$

Finally, with an initial guess in hand, and a stable re-parameterization the local planner can solve a simple gradient descent problem to drive the vehicle to the goal posture.

Thus, we can now compute a set of parameterized trajectories which may each be evaluated to test for safety and optimality. A description of these aspects of the planner may be found in [17] and such a cost function can obviously be modified based on the goals of the design team. We note that our algorithm implementation is parallelized using OpenMP such that multiple trajectories (with goals regularly sampled around the initial goal) may be evaluated simultaneously. Furthermore, with small changes we can also support quintic splines which expand the variety of possible maneuvers and are more suitable for high speed driving. Figure 10 shows an example of a trajectory generation instance.

Specification

Formal verification requires both a system model and a specification. This means that the project stakeholders must provide an exact definition of the desirable system properties. Furthermore, it is often the case that such properties are expressed as occurring only under certain conditions. For convenience we provide the symbols used to describe the vehicle specification in Table 4.

Table 4: Symbols for Specifications

Symbol List		
Symbol	Units	Description
k	-	Search Depth
ϕ	-	Ego Vehicle Spec
ξ	-	Environment Spec
LC	Boolean	Lane Change Request
LO	Boolean	Lane Occupied
v_{ego}	m/s	Velocity of Ego Vehicle
$s_{x_{ego}}$	m	Position of Ego Vehicle, x
$s_{y_{ego}}$	m	Position of Ego Vehicle, y
v_{limit}	m/s	Speed Limit
$s_{x_{ref}}$	m	Centerline Reference, x
$s_{y_{ref}}$	m	Centerline Reference, y
$w(s_{x_{ref}}, s_{y_{ref}})$	m	Lane Width
B	m	Buffer
r	m	Collision Radius
t	s	Current Timestep
t_{max}	s	Max Timestep
$\square$	-	Always
$\rightarrow$	-	Implies
$\neg$	-	Not
$\wedge$	-	And
$\vee$	-	Or

An example specification follows: the ego vehicle should drive in the selected lane at the speed limit *unless* a stop sign is encountered. We note that the traffic laws of a given region provide a partial, but informal definition of many of the high level specifications which the ego vehicle should adhere to.

Ego Vehicle Specification

The specification for the ego vehicle has two components: safety properties and liveness properties. A specification for the ego vehicle in the case study follows:

- The ego vehicle travels at a velocity less than or equal to the speed limit

$$\square (v_{ego} \leq v_{limit}) \quad (29)$$

- The ego vehicle does not drive backwards

$$\square (v_{ego} \geq 0) \quad (30)$$

- The ego vehicle does not collide with any of the n other objects in the environment

$$\square \left(\sqrt{(s_{x_{ego}} - s_{x_{env_i}})^2 + (s_{y_{ego}} - s_{y_{env_i}})^2} \geq r \right) \quad \forall i = 1 \dots n \quad (31)$$

- If a timed lane change request is invoked, the ego vehicle completes the lane change on time.

$$\square (LC \rightarrow (s_{y_{ego}} > w) \wedge (t \leq t_{max})) \quad (32)$$

Environment Specification

The other vehicles operating within a scenario present both an interesting challenge and a primary motivation for formal verification. It is clear that it is impossible to know the intentions of

the agents operating such vehicles; their execution represents a significant source of non-determinism. In fact, a more complex model of such agents which includes details such as steering angle or tire friction will not enable less conservative results, for it is the control input not the plant that remains the largest unknown. Thus, we conclude that: *for verifying the autonomous agent, only the perceptible behavior of other agents is important, not their internal structure.*

Still it remains clear that *the behavior of other agents must be part of the scenario description.* As such we present a safety case which assumes that other agents will follow a certain minimal set of driving rules. For brevity we will reference the following specification as ξ in the case studies.

- Acceleration ceases when some maximum velocity is reached.

$$\Box (v_{env} \geq v_{max} \rightarrow a = 0) \quad (33)$$

- Other agents must drive in the proper direction according to their lane.

$$\Box (v_{env} \geq 0) \quad (34)$$

- The accelerations of other agents are within those rates achievable by maximum engine power

$$\Box (a_{env} \leq a_{max}) \quad (35)$$

- Other agents maintain their lanes unless explicitly specified not to.

$$\Box (\neg LC \rightarrow (y_{min} \leq s_{y_{env}}) \wedge (y_{max} \geq s_{y_{env}})) \quad (36)$$

- Lane changes by other agents are only permitted if the alternate lane is unoccupied or unless a degenerate scenario is being modeled.

$$\Box (LO \rightarrow \neg LC) \quad (37)$$

APEX internals and theory

APEX maintains an internal representation of the scenario as a *hybrid system*. The components of this hybrid system are:

- The behavioral planners of all vehicles involved, $\mathcal{B}_1, \dots, \mathcal{B}_m$. Fig. 11 shows the behavioral planner we used in the case study for a lane change. A behavioral planner is a finite state system. We will refer to each state of a behavioral planner as a *mode*.
- For every vehicle, the continuous dynamics involved in each of the modes of its behavioral planner. In general, different modes may require different dynamics: e.g. a Collision Avoidance mode which is invoked when a collision is imminent requires more stability control than a turn at a low speed. The continuous dynamics are given in terms of Ordinary Differential Equations (ODEs) $\dot{x}_i = f_i(x_i)$, where $x_i \in R^n$ is the continuous state of the i^{th} agent.
- For each vehicle, transition conditions between the modes of the behavioral planner $\mathcal{B}_i$ are expressed in terms of the state vector x_i . The planner transitions between two modes

q and q' only if a *guard condition* $G_{q,q'}$ is satisfied. In general, the guard condition for $\mathcal{B}_i$ is expressed as a set in the state space of *all the agents*, since transitions will occur based on, for example, how close two vehicles are to each other. Specifically, let $x = (x_1, \dots, x_n)$ combine the states x_i of the individual vehicles. So $x \in R^{n \cdot m}$. Then there's a transition between two states q and q' of $\mathcal{B}_i$ only if $x \in G_{qq'} \subset R^{n \cdot m}$. For example, there's a LF-to-LC transition only if the two cars are closer than 10m and the following car is faster than the leading car. In this case $G_{LF,LC} = \{x \mid \|x_1 - x_2\| \leq 10 \wedge v_2 > v_1\}$.

Together, these make up a hybrid system, so-called because it combines discrete dynamics in the behavioral planner with continuous dynamics in each mode. We will refer to the n hybrid systems of the n agents in the scenario as $H_1, \dots, H_m$. The *state of the scenario* x is simply $x = (x_1, \dots, x_n)$.

APEX does not keep an internal representation of the motion planner. Rather, as explained in earlier sections, APEX issues calls to the motion planner in the course of the verification, and obtains a trajectory from it.

APEX also needs to maintain a description of the scenario specification. This specification is provided by the user and can be any formula in first-order logic over the set of modes and states of all agents. See the *Case Study*. For example the following is a possible specification:

$$Mode_1 = LC \rightarrow |\dot{\psi}| \leq b$$

The following sections describe how APEX verifies a property of the scenario using this internal representation.

Execution tree and formal model

Let $\mathcal{B}$ be a behavioral planner of a given vehicle. The formal model of the behavioral planner is a finite transition system $\mathcal{B} = (Q, q_0, \Sigma, \rightarrow)$ where Q is the finite set of modes, q_0 is the initial mode, Σ is a set of output labels, and $\rightarrow \subset Q \times \Sigma \times Q$ is the labeled transition relation of the system. We write $q \xrightarrow{\sigma} q'$ for $(q, \sigma, q') \in \rightarrow$. Fig. 11 shows the behavioral planner that is used by APEX by default for modeling a lane change controller. It can be described as $\mathcal{B} = (\{LC, LF\}, LF, R^n, \{(LF, LF), (LF, LC), (LC, LF)\})$. In mode LF, the vehicle's goal is to follow the current lane. In mode LC, the vehicle's goal is to change lanes. In general, a mode represents a decision by the controller, a *behavior* that the vehicle should follow. With every transition between modes, the behavioral planner outputs a vector x_B in R^n : this is the destination that the vehicle must reach. The planner transitions between modes when certain *guard conditions* are satisfied.

The behavioral planner advances in discrete time. The discrete time advances, for example, with every update of the vehicle's sensors. Thus $\mathcal{B}$ makes a decision on what to do everytime its information about the environment is updated. The planner may decide to maintain the current decision, i.e., stay in the same mode, if that mode has a self-loop. Mode LF has a self-loop in Fig. 11. Let $\Delta t > 0$ be the update period. Since every scenario is time-limited, and every transition takes fixed non-zero time Δt , there is a natural limit D on the number of decisions, or transitions, that can be taken in any given scenario.

In the first step of the verification process, APEX builds an execution tree: the root of the tree is the initial mode q_0 , and every branch of the tree represents one possible sequence of decisions, i.e. one possible execution of $\mathcal{B}$. See Fig. 13 for the execution tree of the behavioral planner of Fig. 11. Since the number of transitions is bounded by D in a given scenario, this tree has a depth at most D .

With the execution tree built, APEX must next verify that the sequence of decisions taken by the behavioral planner can be implemented by the low-level controllers. E.g., let (LF, LF, LC) be a sequence of decisions of depth 3. In every occurrence of LF, APEX must check that the vehicle can indeed follow the lane, and in every occurrence of LC, APEX must verify that the vehicle can indeed change lanes. In the next section, we define what it means to ‘follow the lane’ and ‘change lanes’ via the motion planner.

Calling the motion planner

After building the execution tree, APEX starts executing every branch, starting at the root, which is the initial mode q_0 . The initial set of continuous states is X_0 . A transition is taken if the initial set intersects its guard. Since X_0 may intersect more than one guard, then more than one transition are possible. APEX explores all transitions (all branches) in the execution tree. In each mode APEX enters, $\mathcal{B}$ will output a destination x_B . Formally, x_B is a scenario state, but in what remains, it is simpler to think of it as the position that the ego vehicle must reach.

APEX then calls the motion planner to obtain the trajectory that the vehicle will follow. Since the current state is only known as a set X_A , APEX sets the starting point of the trajectory to be the center x_A of X_A . The motion planner then returns a trajectory starting at x_A and ending in a neighborhood of x_B . The neighborhood shape and size are known to APEX and are part of the motion planner’s description. Let that neighborhood be X_B . Note that APEX does not place any restrictions on the motion planner’s operation and calls it as a black box. Therefore, the actual motion planner that is used on the real car can be used in the verification of the system. In this way the verification results are directly applicable to the actual deployed software.

Verifying each trajectory

Once a trajectory is generated connecting $x_A \in X_A$ to the neighborhood X_B of x_B , it remains to verify that the ego vehicle will always reach X_B within a specified amount of time T , regardless of where it starts in X_A . To verify that the specification is satisfied, APEX builds a *reachability problem*. This reachability problem is characterized by the following:

- The system: in this case, the system consists of the scenario hybrid system.
- The target set: this is the set that the system should reach. In this case the state of the ego vehicle x_1 should reach X_B , and there are no target sets for the other agents.
- The unsafe set: this is the set that the scenario hybrid system must *not* reach at any point in time. In this case, the ego vehicle must not get closer than d_{min} to any other agent in the scenario.

Figure 11: An automaton describing a simplistic behavior planner for lane changes

- A time bound: the target set must be reached within a certain amount of time T .

We call the above a bounded reachability problem. To solve this problem, APEX passes it to dReach [12], a reachability analysis tool for nonlinear hybrid systems. dReach answers the question: is there a trajectory of the vehicle starting in X_A that will violate the constraints? (e.g. will not reach the target set X_B or will get too close to another vehicle). dReach returns one of two answers. If the answer dReach returns is SAFE, then it is *guaranteed* that *no* behavior of the ego vehicle will violate the constraints. It should be stressed that this is a mathematical guarantee: no amount of simulation in this case will reveal a violation, because dReach guarantees that no such violation exists. If dReach answers δ -UNSAFE, then this means that there exists a behavior of the ego vehicle which, when perturbed by an amount $\delta > 0$, violates the constraints. See Fig. 4. The parameter δ can be set by the user. It suffices to choose δ small enough so δ -SAT means the system is not robust since a small perturbation of size δ could cause it to violate the constraints.

Case Study

We briefly introduce and expand on the concept of driving scenarios to help reason about inherently diverse situations and requirements which an autonomous vehicle might face.

An unsafe lane change scenario

The following example describes a lane change scenario in the context of a mission and mobility goals. In this description we imply a valid local planning solution, and seek to verify that all possible individual trajectories which are selected in the *execution* of the plan are safe. First, in *Scenario 1* we will demonstrate a dangerous condition that could have been missed under testing or simulation. Next, in *Scenario 2* we will show how a refinement in the requirements on the perception system or a refinement in the behavioral controller can lead to a provably safe maneuver. Finally, in *Scenario 3* we demonstrate how a change in manufacturer specification can be accurately assessed for safety. To perform verification, we employ dReach version 3.15.10.02 on a Mac OSX laptop with Intel(R) Core i7(R) 2.60GHz CPU and 16 GB memory, and the results are provided in Table 5.

Scenario 1 (A simple lane change and goal) As shown in Fig. 12, the ego vehicle is driving in the right lane of a uni-directional two lane road network. Another car is driving in front of the ego vehicle at a lower speed. We include

Behavioral controller

Given any deterministic finite automaton it is possible to express as a *computational logic tree*. Such a tree is rooted in a single state, is infinite in size, and represents a branching notion of time; that is each state (moment in time) may split into multiple possible future worlds. As we will explain in the following sections, such a representation is at the heart of the APEX approach and verification occurs over a bounded search depth on such a computation tree.

We present the initialization of the scenario and the results of the verification. Table 5 contains the initialization of each parameter.

Verification and Result

Finally, for the lane change case, we define an additional constraint set R_{unsafe} as well as a goal set representing the maximum allowable deviation from the goal state. R_{unsafe} expresses that the system fails if it still hasn't changed lanes within 2 sec or it collides with the car ahead of it.

$$((s_y < w) \wedge (t > 2)) \vee ((s_y < w) \wedge (sx - \epsilon > s_{x_{env}})) \quad (38)$$

Then, using APEX we *attempt* to show that there is no execution of the system which can enter R_{unsafe} . How-

Figure 12: A lane change scenario that could have been missed in testing due to nonintuitive and uncountably infinite set of initial conditions. This scenario is unsafe for certain inter-vehicle buffer spacing and reachability analysis determines the minimum spacing to achieve a safe lane change.

D = search depth

Figure 13: An automaton describing a simplistic behavior planner for Lane Following (LF) and Lane Changes (LC)

Table 5: Verification Results

Symbol	Scenario 1	Scenario 2	Scenario 3
w	3.7	3.7	3.7
B	15	20	20
δ	0.1	0.1	0.1
v_{ego}	[10.8, 11.1]	[10.9, 11]	[10.9, 11]
$s_{x_{ego}}$	[0.0, 0.5]	[0.0, 0.5]	[0.0, 0.5]
$s_{y_{ego}}$	[0.0, 0.1]	[0.0, 0.05]	[0.0, 0.05]
Ψ	[0.0, 0.1]	[0.0, 0.1]	[0.0, 0.2]
Search Depth	2	2	2
Verification Time (s)	30.821	373.924	36.166
Result	δ -UNSAFE	SAFE	δ -UNSAFE

ever, because the system is incorrectly designed dReach returns δ -UNSAFE.

A safe lane change scenario

Using the information and counterexample from the previous scenario it is easy to see that the behavior controller must be corrected in order to guarantee safety of the lane change scenario.

Scenario 2 (A more conservative behavioral controller)

We begin with Scenario 1. In order to ensure the forward safety of the vehicle we propose a small modification to the behavioral controller of the vehicle, and furthermore require that the ego vehicle’s localization system return estimates with less uncertainty. Namely, we first increase the size of variable buffer, so that the ego vehicle is forced to initiate a lane change maneuver earlier. Secondly, we decrease the size of the initial sets. Speed v now starts anywhere in $[10.9, 11]$ and s_y starts in $[0.0, 0.05]$.

With these changes, dReal returns SAFE, meaning that no trajectory of the system violates the constraints.

A supplier issues a specification change

Given that a safe controller has been found a supplier wishes to know if they may reduce the accuracy of several key sensors associated with localization of the ego vehicle. Such a specification change is known to add significant uncertainty to the estimate of the ego vehicle’s heading angle during the planning phase.

Scenario 3 (Large perception errors) We begin with Scenario 2. In order to reflect the change in supplier specification

we update the localization system return estimates to reflect greater uncertainty. Namely, we increase the size of the initial set for ego vehicle heading such that Ψ starts in $[0.0, 0.2]$.

The result of this modification is again δ -UNSAFE, because the ego vehicle clips the rear bumper of the environmental vehicle while executing the lane change maneuver. Again the engineer in charge of the project may use the new information to refine the controller design or reject the suppliers specification change. In this way formal verification efforts can be a useful tool in determining the *requirements* which sensors and perception systems must meet given a particular control algorithm.

Conclusion

APEX is a tool for formally verifying the trajectory planning and tracking stacks of ADAS/AV cars. It can perform formal verification on realistic autonomous vehicle planning stacks. In this paper we demonstrate a case study which formally verifies a lane change maneuver. Future work will incorporate more complex behavioral controllers for other scenarios, including synthesized planners, and will add a GUI to the tool as well as a means of visualizing complex counterexamples.

REFERENCES

- [1] M. Mitchell Waldrop. Autonomous vehicles: No drivers required. *Nature*, 518(7537): 20–23, feb 2015. doi: 10.1038/518020a. URL <http://dx.doi.org/10.1038/518020a>.
- [2] Volvo press release: US urged to establish nationwide federal guidelines for autonomous driving. 2015.
- [3] Laura R Humphrey, Eric M Wolff, and Ufuk Topcu. Formal specification and synthesis of mission plans for unmanned aerial vehicles. In *AAAI Spring Symposium Series*, 2014.
- [4] Shinpei Kato, Eijiro Takeuchi, Yoshio Ishiguro, Yoshiki Ninomiya, Kazuya Takeda, and Tsuyoshi Hamada. An open approach to autonomous vehicles. *Micro, IEEE*, 35(6):60–68, Nov 2015. ISSN 0272-1732. doi: 10.1109/MM.2015.133.
- [5] Rajeev Alur, Tomas Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. In *Symposium on Principles of Distributed Computing*, pages 139–152, 1991.
- [6] Luke Fletcher, Seth Teller, Edwin Olson, David Moore, Yoshiaki Kuwata, Jonathan How, John Leonard, Isaac Miller, Mark Campbell, Dan Huttenlocher, et al. The mit-cornell collision and why it happened. *Journal of Field Robotics*, 25(10):775–807, 2008.
- [7] Chris Urmson, Joshua Anhalt, Drew Bagnell, Christopher Baker, Robert Bittner, MN Clark, John Dolan, Dave Duggins, Tugrul Galatali, Chris Geyer, et al. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, 25(8):425–466, 2008.
- [8] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M. Murray. Receding horizon control for temporal logic specifications. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '10, pages 101–110, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-955-8. doi: 10.1145/1755952.1755968. URL <http://doi.acm.org/10.1145/1755952.1755968>.
- [9] Petter Nilsson, Omar Hussien, Ayca Balkan, Yuxiao Chen, Aaron D Ames, Jessy W Grizzle, Necmiye Ozay, Huei Peng, and Paulo Tabuada. Correct-by-construction adaptive cruise control: Two approaches.
- [10] Matthias Althoff and John M Dolan. Reachability computation of low-order models for the safety verification of high-order road vehicle models. In *American Control Conference (ACC), 2012*, pages 3559–3566. IEEE, 2012.
- [11] Erann Gat. On three-layer architectures. In *Artificial Intelligence and Mobile Robots*. MIT Press, 1998.
- [12] Soonho Kong, Sicun Gao, Wei Chen, and Edmund M. Clarke. dreach: Delta-reachability analysis for hybrid systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015*, pages 200–205, 2015.
- [13] Sicun Gao, Soonho Kong, Wei Chen, and Edmund Clarke. Delta-complete analysis for bounded reachability of hybrid systems. *arXiv preprint arXiv:1404.7171*, 2014.
- [14] Rajesh Rajamani. *Vehicle dynamics and control*. Springer Science & Business Media, 2011.
- [15] M. Althoff and J.M. Dolan. Online verification of automated road vehicles using reachability analysis. *Robotics, IEEE Transactions on*, 30(4):903–918, Aug 2014. ISSN 1552-3098. doi: 10.1109/TRO.2014.2312453.
- [16] Jarrod M Snider. Automatic steering methods for autonomous automobile path tracking. *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RITR-09-08*, 2009.
- [17] Matthew McNaughton. *Parallel Algorithms for Real-time Motion Planning*. PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, July 2011.
- [18] Bryan Nagy and Alonzo Kelly. Trajectory generation for car-like robots using cubic curvature polynomials. *Field and Service Robots*, 11, 2001.